

# Synchronous Collaboration in Ubiquitous Computing Environments

Conceptual Model and Software Infrastructure  
for Roomware Components

Vom Fachbereich Informatik  
der Technischen Universität Darmstadt  
zur Erlangung des akademischen Grades eines  
Doktor-Ingenieurs (Dr.-Ing.)  
genehmigte

**Dissertation**

von  
Diplom-Informatiker

**Peter Tandler**

geb. Seitz  
aus Herdecke

Referent: Prof. Dr. Erich J. Neuhold  
Koreferent: Prof. Dr. Brad A. Myers  
Tag der Einreichung: 01. März 2004  
Tag der mündlichen Prüfung: 30. April 2004

Darmstadt 2004

D17  
Darmstädter Dissertation

---

Version 1.3.4, Final.

© 2004 Peter Tandler

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/2.0/de/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

## Abstract

Ubiquitous computing environments offer a wide range of devices in many different shapes and sizes, creating new possibilities for interaction. In the context of meetings and teamwork situations, it is desirable to take advantage of their different properties for synchronous collaboration. Besides providing an adapted user interface, this requires the software to be designed for *synchronous access* to shared information using *heterogeneous devices* with *different interaction* characteristics. The handling of these requirements poses challenges for software developers. As this field is still emerging and no mature models, tools, and standards are at hand, developers have to create their own solutions from scratch.

The goal of this thesis is to provide guidance and support for developers of *synchronous groupware applications for ubiquitous computing environments*. They have to be enabled to develop applications more efficiently and with the flexibility and extensibility that is required for ubiquitous computing. The development effort can be reduced effectively if support for developers is provided at several levels. Developers need assistance when creating models of the applications to be developed, when choosing an appropriate architecture, when creating the design, and finally when implementing. This implies that an architecture-driven, model-based development approach should be followed.

While the implementation of a single synchronous UbiComp application still requires research, the development of appropriate *development support* is even more challenging. Common properties of ubiquitous computing applications have to be identified. Future developments and extensions have to be predicted. Requirements of different research areas have to be fulfilled. Addressing these aspects, the goal of this dissertation is accomplished by providing extensions to the state of the art at four levels:

- (1) A *conceptual model* of synchronous UbiComp applications defines a high-level structure for applications that ensures reusability and extensibility of developed software components. It identifies separation of concerns, degree of coupling and sharing, and level of abstraction as the three main design dimensions of these applications. The conceptual model provides two key contributions to the state of the art. First, it proposes the strict *separation of user interface and interaction* concerns orthogonal to the level of abstraction that is not found in current HCI models. This is a crucial extension of HCI models that is required in the context of ubiquitous computing. Second, it introduces a new view on the concept of *sharing*. By applying the CSCW concept of sharing in the context of ubiquitous computing, sharing user interface, interaction, and environment state becomes relevant. Thereby,

the concept of sharing as known from CSCW can be extended to function as a guiding principle for UbiComp application design. This novel design approach helps ensuring the extensibility and flexibility that is required in ubiquitous computing.

- (2) A flexible *software architecture* identifies essential abstractions that support the development of synchronous applications in “roomware” environments. Roomware refers to the integration of room elements with information technology, such as interactive tables, walls, or chairs. Roomware environments represent one form of ubiquitous computing environment. They are used in this thesis as an application context for the conceptual model. The software architecture refines the conceptual model to meet the needs of roomware environments.
- (3) An object-oriented *application framework* that has been designed and implemented provides a reusable design and reusable software components. Furthermore, extensibility is supported by explicit mechanisms that are provided to allow adaptability for variable aspects of applications. Thus, the application framework helps developers with the design and implementation.
- (4) To show how model, architecture, and framework can be applied, the design of sample *roomware applications* is explained. To demonstrate the extensibility, several *new forms of interaction* that are required for roomware environments are implemented. The developed applications and interaction forms are used in i-LAND, the roomware environment at Fraunhofer IPSI. Besides being a contribution on their own, the developed applications and new forms of interaction provide evidence that the conceptual model effectively supports developers in meeting the requirements of roomware environments. They show that the model helps reduce the implementation effort when accompanied by appropriate software development tools such as the application framework.

The *conceptual model*, *software architecture*, and *application framework* presented in this thesis relieve software developers from the burden of handling all details of multiple interaction forms, and of many critical issues when dealing with synchronous collaboration. By these means, the developer can concentrate on the task at hand designing software at an appropriately high abstraction level, and thus create applications with a higher quality that are flexibly extensible.

## Kurzfassung

Die stetig zunehmende Allgegenwart von Computern ermöglicht neue Formen der Interaktion und Zusammenarbeit. In der Umgebung steht eine Vielzahl von Geräten zur Verfügung, die verschiedene Formen, Größen und Interaktionsmöglichkeiten aufweisen. Vor allem für Meetings und Teamwork ist es wünschenswert, dass die verschiedenen Eigenschaften aller verfügbaren Geräte effektiv ausgenutzt werden können, um synchrone Zusammenarbeit zu ermöglichen. Beispielsweise muss eine Benutzungsschnittstelle bereitgestellt werden, die an die verschiedenen Geräte angepasst ist. Genauso wichtig ist es aber, dass die Software so entworfen wird, dass *synchroner* Zugriff auf gemeinsame Informationen möglich wird, obwohl *unterschiedliche Geräte mit differierenden Interaktionsmöglichkeiten* verwendet werden. Die resultierenden Anforderungen zu erfüllen, stellt eine Herausforderung für Softwareentwickler dar. Das Forschungsgebiet „Ubiquitous Computing“ (UbiComp) befasst sich mit den Anforderungen, die durch die Allgegenwart von Computern entstehen. Da dieses Forschungsfeld noch in der Entwicklung begriffen ist und daher weder bewährte Modelle und Werkzeuge noch Standards zur Verfügung stehen, müssen Softwareentwickler eigene Lösungen von Grund auf neu entwickeln.

Ziel dieser Arbeit ist es daher, Softwareentwickler von *synchronen Groupware Anwendungen in UbiComp-Umgebungen* Hilfestellung und Unterstützung zu bieten. Es soll ermöglicht werden, diese Anwendungen effizient und mit der für das „Ubiquitous Computing“ nötigen Flexibilität und Erweiterbarkeit zu entwickeln. Der gesamte Entwicklungsaufwand kann effektiv reduziert werden, indem Entwicklern Unterstützung auf mehreren Ebenen geboten wird: Entwickler benötigen Hilfe bei der Erstellung von Anwendungsmodellen, bei der Wahl einer passenden Architektur, bei der Erstellung des Designs und schließlich bei der Implementierung. Diese Herangehensweise legt einen architektur-zentrierten und modell-basierten Entwicklungsprozess zugrunde.

Heute erfordert die Entwicklung einer einzigen synchronen UbiComp-Anwendung noch Forschung; geeignete Unterstützung für deren Entwickler bereitzustellen ist daher eine noch größere Herausforderung. Hierzu müssen gemeinsame Eigenschaften von UbiComp-Anwendungen identifiziert, zukünftige Entwicklungen und Erweiterungen vorhergesehen und Anforderungen aus verschiedenen Forschungsgebieten beachtet und erfüllt werden. Das Ziel dieser Dissertation wird – unter Berücksichtigung der genannten Aspekte – erreicht, indem auf vier Ebenen ein neuer Beitrag zu dem bisherigen Stand der Technik geleistet wird:

- (1) Ein *konzeptionelles Modell* von synchronen UbiComp-Anwendungen beschreibt die grobe Struktur dieser Anwendungen, um Wiederverwendbarkeit und Erweiterbarkeit von entwickelten Software Komponenten sicherzustellen. Es identifiziert Aufgabenbereich, Kopplungsgrad und Abstraktionsebene als die drei wesentlichen Entwurfsdimensionen.
- (2) Eine flexible *Software-Architektur* identifiziert die essentiellen Abstraktionen, mithilfe derer synchrone Anwendungen für „Roomware“-Umgebungen entworfen werden können. „Roomware“ bezeichnet die Integration von Raumelementen mit Informationstechnik, wie zum Beispiel interaktive Tische, Wände oder Stühle. *Roomware*-Umgebungen sind dabei ein konkretes Beispiel für *UbiComp*-Umgebungen. Sie werden in dieser Arbeit als Anwendungskontext für das konzeptuelle Modell verwendet; die entwickelte Software-Architektur verfeinert das konzeptuelle Modell in Bezug auf die Bedürfnisse von *Roomware*-Umgebungen.
- (3) Ein objekt-orientiertes *Anwendungsframework* wurde entwickelt, das zum einen ein wiederverwendbares Design und wiederverwendbare Software-Komponenten bereitstellt. Zum anderen wird die Erweiterbarkeit durch explizite Mechanismen ermöglicht, welche die Adaptierbarkeit bezüglich der variablen Aspekte der Anwendungen sicherstellen. Hierdurch hilft das Anwendungsframework Entwicklern bei Design und Implementierung.
- (4) Um zu zeigen, wie das konzeptuelle Modell, die Architektur und das Framework angewandt werden können, wird das Design von Beispielanwendungen für *Roomware* erläutert. Um die Erweiterbarkeit zu demonstrieren, wurden mehrere neue Interaktionsformen entwickelt, die in *Roomware*-Umgebungen benötigt werden. Die entwickelten Anwendungen und Interaktionsformen werden in *i-LAND*, der *Roomware*-Umgebung am Fraunhofer IPSI, eingesetzt.

Das konzeptuelle Modell, die Software-Architektur und das Anwendungsframework, die in dieser Arbeit entwickelt wurden, befreien Softwareentwickler davon, sich um die Details verschiedener Interaktionsformen und um viele kritische Aspekte synchroner Kooperation kümmern zu müssen. Dies ermöglicht Entwicklern, sich auf ihre Kernaufgabe zu konzentrieren, indem sie Software auf einer angemessenen Abstraktionsebene entwerfen und somit flexibel erweiterbare Anwendungen höherer Qualität erstellen können.

»Siehst du, Momo«, sagte er dann zum Beispiel, »es ist so: Manchmal hat man eine sehr lange Straße vor sich. Man denkt, die ist so schrecklich lang; das kann man niemals schaffen, denkt man.«

Er blickte eine Weile schweigend vor sich hin, dann fuhr er fort: »Und dann fängt man an, sich zu eilen. Und man eilt sich immer mehr. Jedes Mal, wenn man aufblickt, sieht man, dass es gar nicht weniger wird, was noch vor einem liegt. Und man strengt sich noch mehr an, man kriegt es mit der Angst, und zum Schluss ist man ganz außer Puste und kann nicht mehr. Und die Straße liegt immer noch vor einem. So darf man es nicht machen.«

Er dachte eine Zeit nach. Dann sprach er weiter: »Man darf nie an die ganze Straße auf einmal denken, verstehst du? Man muss nur an den nächsten Schritt denken, an den nächsten Atemzug, an den nächsten Besenstrich. Und immer wieder nur an den nächsten.«

Wieder hielt er inne und überlegte, ehe er hinzufügte: »Dann macht es Freude; das ist wichtig, dann macht man seine Sache gut. Und so soll es sein.«

Beppo Straßenkehrer in Momo (Ende, 1973)





# Contents

Abstract .....	iii
Kurzfassung .....	v
Contents .....	ix
1. Introduction .....	1
1.1. Challenges for Software Development .....	2
1.2. Context of the Dissertation: i-LAND and Roomware .....	5
1.3. Goal of the Dissertation .....	5
1.4. Contributions of the Dissertation .....	6
1.5. Outline of the Dissertation .....	10
<hr/>	
Part I. The BEACH Conceptual Model: Guidance for Ubiquitous Computing Developers .....	13
2. Requirements of Ubiquitous Computing Applications .....	15
2.1. The i-LAND Roomware Project .....	15
2.2. Application Scenarios of Synchronous Collaboration within Roomware Environments .....	18
2.3. HCI: User Interface and Interaction .....	20
2.4. UbiComp: Ubiquitous Computing Environments .....	23
2.5. CSCW: Synchronous Collaboration .....	27
2.6. SW: Software Engineering Requirements .....	30
2.7. Summary of Requirements .....	31
3. Related Work .....	33
3.1. Object-Oriented Frameworks and Architectures .....	34
3.2. Software Models for Human-Computer Interaction .....	37
3.3. Software Models for Ubiquitous Computing .....	44
3.4. Software Models for Computer-Supported Cooperative Work .....	48
3.5. Conclusions: What to Use, What is out of Focus & What is missing .....	55
4. A Conceptual Model for UbiComp Applications .....	59
4.1. Design Dimensions .....	60

4.2.	The BEACH Conceptual Model .....	61
4.3.	First Dimension: Separating Basic Concerns .....	62
4.4.	Second Dimension: Coupling and Sharing .....	68
4.5.	Third Dimension: Conceptual Levels of Abstraction.....	75
4.6.	Comparison with Related Work.....	77
4.7.	Discussion of the Conceptual Model .....	83
<hr/>		
Part II.	The BEACH Software Architecture and Framework: Support for Implementing Roomware Applications.....	85
5.	Software Architecture for Roomware Applications.....	87
5.1.	Architecture Overview .....	88
5.2.	Model Layer: Domain-Independent Abstractions for Roomware Components ....	89
5.3.	Generic Layer: Informal Collaboration Support for Roomware Environments .....	90
5.4.	Task Layer: A Platform for Roomware Applications and Extensions .....	92
5.5.	Core Layer: Specialized Infrastructure for Roomware Components .....	93
5.6.	Discussion of the BEACH architecture.....	97
6.	Software Infrastructure for Roomware Environments.....	101
6.1.	Design of the Core Layer .....	101
6.2.	Design of the Core Interaction Model Support .....	109
6.3.	Design of the Model Layer .....	112
6.4.	Design of the Data Model .....	113
6.5.	Design of the Application Model .....	115
6.6.	Design of the Environment Model.....	116
6.7.	Design of the User Interface Model .....	117
6.8.	Design of the Interaction Model .....	119
6.9.	Discussion: Properties of the BEACH Model Framework .....	126
7.	Generic Support for Collaboration in Roomware Environments .....	129
7.1.	Generic Environment Model: Roomware Components .....	130
7.2.	Generic Data and Application Model: Document Elements .....	131
7.3.	Generic User Interface Model: Roomware User Interface .....	135
7.4.	Generic Interaction Model: Visual Output for Roomware Components.....	139
7.5.	Generic Interaction Model: User Input to Roomware Components.....	143
7.6.	Example Device Configurations .....	148
7.7.	Discussion: Properties of the BEACH Generic Collaboration Framework .....	151
<hr/>		
Part III.	Applications for Roomware Environments: Validating the Usability of the BEACH Model and Framework .....	153
8.	Extending BEACH for New Forms of Interaction .....	155
8.1.	Passage Mechanism: Interaction with Physical Objects .....	156
8.2.	ConnecTTables: Dynamic Reconfiguration .....	163
8.3.	Integrating Audio Feedback: Non-Speech Augmentation.....	170
8.4.	Discussion: Experiences Extending the BEACH Framework.....	175
9.	Tools for Collaborative Roomware Environments.....	179
9.1.	MagNets: Extending Document Model and Interaction Behavior .....	180
9.2.	PalmBeach: Implementing Models on a Different Platform.....	184

9.3. Discussion: Experiences Building Roomware Applications.....	188
10. Conclusions & Future Work.....	191
10.1. Achievements of the Dissertation.....	191
10.2. Evaluation: Comparison against Requirements.....	195
10.3. Open Questions & Future Research Topics.....	199
Acknowledgements.....	203
<hr/>	
Appendix.....	205
Appendix A Notations Used.....	207
Appendix B Abbreviations.....	211
Appendix C Glossary.....	213
Appendix D Development Tools Used.....	221
Figures.....	223
Tables.....	231
Examples.....	233
References.....	235



# I. Introduction

---

The introduction examines properties of ubiquitous computing (UbiComp), human-computer interaction (HCI), and computer-supported cooperative work (CSCW), which have been identified as research areas that must be considered when supporting synchronous collaboration in ubiquitous computing environments. The challenges for software development that arise when combining requirements of these areas are illustrated. The roomware concept is introduced, constituting the context and primary application domain of this thesis. The goal of the dissertation is defined to provide support for developers of synchronous applications in ubiquitous computing environments. This dissertation contributes to the state of the art at four different levels by providing a conceptual model of synchronous applications in ubiquitous computing environments, a software architecture, an application framework, and roomware applications that also serve as a proof-of-concept.

---

Ubiquitous computing environments offer a wide range of devices in many different shapes and sizes (Weiser, 1993). Today, the desktop computer as a device is present in every office and many homes. Portable information appliances, such as personal digital assistants, enjoy an increasing popularity. In the future as it can be foreseen today, collaboration between users and environments with multiple interconnected devices will determine work and everyday activities to a large degree. The vision is that heterogeneous devices will complement each other to provide a consistent usage experience. User interfaces will take advantage of the different properties of the devices. The devices will be closely connected and integrated with the environment and context in which people use them. People will be able to exchange and synchronously share information and functionality among all devices. The devices will indeed seamlessly work together to support fluent collaboration. This vision was first pursued by Mark Weiser, who coined the term *ubiquitous computing* (Weiser, 1991; Weiser, 1996). Related approaches are *pervasive computing* (Lyytinen and Yoo, 2002), *intelligent environments* (Coen *et al.*, 1999; Shafer *et al.*, 1998), or *reactive environments* (Cooperstock *et al.*, 1997).

These approaches have in common the goal of transforming and transcending human–computer interaction resulting rather in direct human–information interaction and human–human cooperation, thus making computers disappear in two ways (Streitz, 2001; Ambient Agoras, 2001). Firstly, the *physical disappearance* of computer devices comes about by integrating computers into the environment, for instance by making the computer-based parts small enough so that they fit in the hand, interweave with clothing, or attach to the body. Secondly, the *mental disappearance* of computers occurs when they become invisible to the user’s *mental eyes*. The important point here is that humans do not perceive the devices as computers anymore, but as embedded elements of augmented artifacts within their environment.

### 1.1. Challenges for Software Development

Today, the development of interactive software is determined to a large degree by the requirements and properties of the single-user desktop PC. The envisioned ubiquitous computing situation, in contrast, differs from the current usage scenarios of desktop PCs in several ways (see fig. 1-1). Firstly, new interaction devices require *new user interface concepts*. Secondly, applications can profit by *adapting to the current environment* and the interaction capabilities it provides. Thirdly, *synchronous collaboration* is essential, taking into account that collaboration with heterogeneous devices must be handled. This section gives a brief introduction of these aspects. A detailed discussion of the requirements addressed is given in chapter 2.

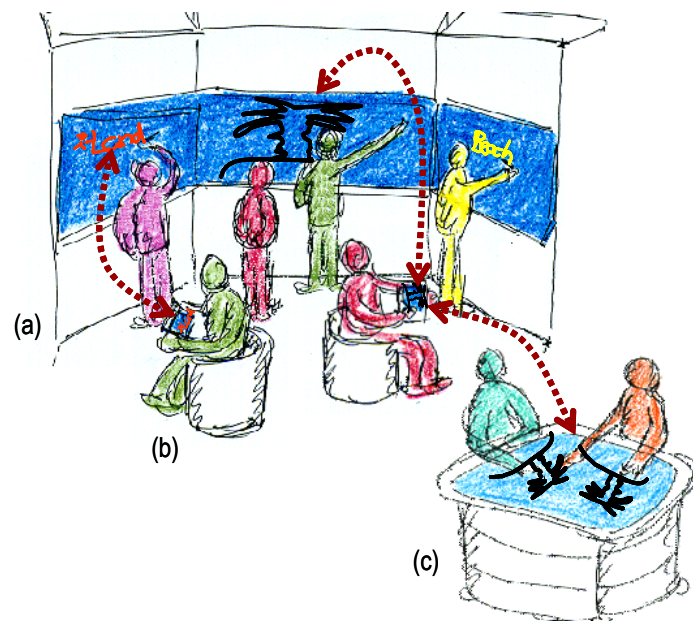


Figure 1-1. In ubiquitous computing environments, heterogeneous devices are used to support synchronous collaboration: (a) an interactive wall with a large visual interaction area, (b) interactive chairs or handhelds with small displays, (c) a horizontal tabletop surface. The drawing is based on a vision scribble we created in early 1997.

The devices available in ubiquitous computing environments have different properties that enable—and need—new possibilities for interaction. Instead of being equipped with mouse and keyboard, other forms of interaction are used for input, such as pen, finger, voice, or gestures. For output, some devices offer displays that range from small to very large wall-size, or that are mounted horizontally. Other devices employ different modalities to replace or augment visual presentation. Some examples are shown in figure 1-1. All these interaction capabilities must be handled by application software. To ensure intuitive and efficient interaction, *new user interface and interaction concepts* must be supported.

The main characteristic of a ubiquitous computing environment is the presence of multiple devices. For interaction, software can utilize the different properties of these devices to complement and augment each other. This requires that the software is *aware of the current environment* of the device it is running on, in order to be able to detect other devices and their properties. Then, a small handheld could detect a nearby interactive wall and use this display to show information that is needed by several collaborating people (see fig. 1-1a–b).

Today, most desktop applications are designed for a single user; they cannot handle collaboration. A forte of ubiquitous computing environments will be the support for *synchronous collaboration*. To allow *truly* synchronous work, the software must be capable of handling multiple users that can simultaneously manipulate a shared document. This requires a software infrastructure that is designed for handling multiple concurrent users—unlike today’s common operating systems.

The devices used for collaboration in ubiquitous computing environments are likely to have different interaction capabilities. Current software tools that support synchronous collaboration assume that all participants use computers with similar interaction capabilities. This assumption does not hold for ubiquitous computing environments. The software must be able to reduce the degree of coupling in such a way that the presentation of shared information can be adapted to the devices used. An extreme example is a synchronously coupled presentation of a visual presentation on a large interactive wall with a voice presentation on a mobile cell phone.

All these features pose new challenges for the development of ubiquitous computing applications.

Due to the described nature of collaborative ubiquitous computing environments, software systems have to cater for all aspects of interaction and collaboration in a heterogeneous environment. The outlined properties lead to requirements addressed by related and intersecting research areas (fig. 1-2) that software systems have to fulfill:

- *Human-Computer Interaction* (HCI) deals with user interfaces and interaction techniques.
- *Ubiquitous computing* (UbiComp) explores dynamic environments with heterogeneous devices.
- *Computer-Supported Cooperative Work* (CSCW) develops techniques to handle synchronous interaction with distributed computers.
- *Software design techniques* (SW) are needed to ensure extensibility and reusability.

Admittedly, this is a simplified view of the research areas, but points only to their relevant contributions within the context of this thesis.

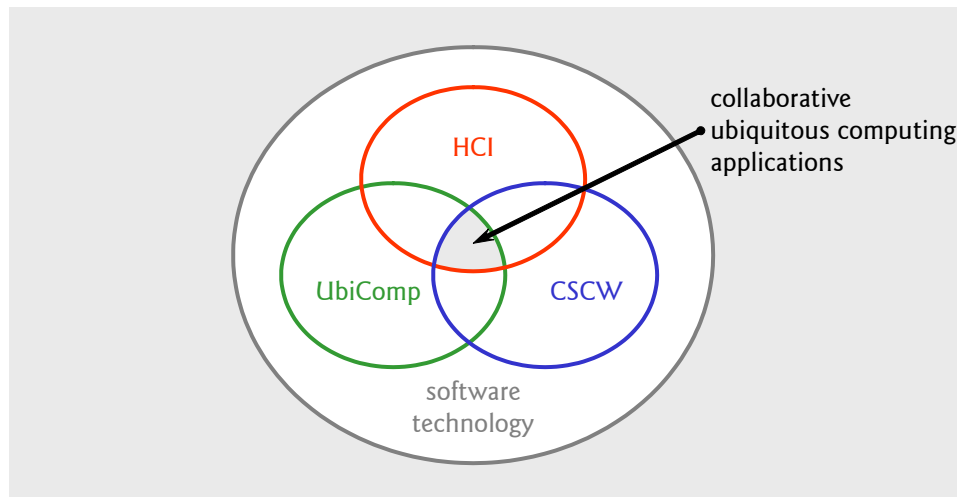


Figure I-2. Related research areas for the design of collaborative ubiquitous computing applications

Whereas several well-established models, frameworks, and tools exist to aid application design for a traditional PC (e.g. Myers, 2003), UbiComp application developers cannot draw upon such tools. They are in the same situation as that of pioneering computer scientists when PC use first began to spread.

Current operating systems, for instance, provide no support for handling this heterogeneity. Synchronous collaboration can be handled by several computer-supported cooperative work frameworks, groupware systems, or middleware infrastructures, but these systems have little or no support for heterogeneous devices. There are research prototypes aimed at managing devices with different interaction capabilities, but these mainly deal with interfaces for, and discovery of, simple services while they lack support for tight collaboration.

All existing tools cover only parts of the necessary functionality. However, they cannot simply be combined, as different approaches often make incompatible assumptions. Frameworks, for example, reverse the flow of control (Fayad, 1999), making it very costly, if not impossible to combine different frameworks (Bosch *et al.*, 1999).

Because of these shortcomings, there is a need for a *software infrastructure* designed to handle heterogeneous environments, to support adequate interaction forms and user interface concepts, as well as offering capabilities for synchronous collaboration. As this kind of infrastructure is built on top of current operating systems, which handle the interaction with the specific hardware, it is referred to as a *middleware* operating system (Román *et al.*, 2001b).

To structure architectures of software systems, it has proven helpful to rely on conceptual models, reference models, and architectural styles that allow reuse of successful designs (Bass *et al.*, 1999). Traditional application models do not provide enough guidance for developers to create software systems for synchronous collaboration in ubiquitous computing environments; software developers must consider further aspects not relevant for software running on a desktop PC. Several researchers have already identified this problem. Yet, there are only few emerging models for ubiquitous computing. Myers *et al.* (2000) states the need for changes in user interface software tools. Garlan (Garlan, 2000; Sousa and Garlan, 2002) and Banavar and Bernstein (2002) found that software architectures for ubiquitous computing systems will have to be more flexible than they are today to meet the challenges that ubiquitous computing poses on software architecture. Abowd (1999) says that building ubiquitous computing applications raises software engineering problems in toolkit design, software structuring for separation of concerns, and component interaction. Winograd (2001) and Dey *et al.* (2001) complain about a lack of conceptual models and tools. Also, there is no straightforward way of combining models coming from different research areas.



Until now, the burden of carefully selecting appropriate models from all relevant areas has been placed on the software developer. He or she has to ensure that the chosen combination does not cause unexpected effects. A conceptual model covering all aspects of synchronous collaboration in ubiquitous computing environments would take this burden off the software developer and provide the structure in which to create appropriate software tools and techniques. Such a unified conceptual model that covers all aspects of interaction and collaboration in a heterogeneous environment has to fulfill requirements brought up by the relevant research areas shown in figure 1-2.

## 1.2. Context of the Dissertation: i-LAND and Roomware

Over the last eight years at IPSI, the Fraunhofer Integrated Publication and Information Systems Institute<sup>1</sup> in Darmstadt (Germany), we have been working in the i-LAND project on supporting synchronous collaboration with roomware components (Streitz *et al.*, 1997; Streitz *et al.*, 1999; Streitz *et al.*, 2001; Streitz *et al.*, 2002; Tandler *et al.*, 2002b; Tandler, 2004). “Roomware” is a term we coined (Streitz *et al.*, 1997) to refer to the integration of room elements with information technology, such as interactive tables, walls, or chairs. Figure 1-1 shows some of the ‘vision scribbles’ of i-LAND created in spring 1997. It presents roomware components we envisioned for i-LAND, an “interactive landscape for creativity and innovation”. The roomware components we built as part of the i-LAND project are tailored to support co-located teamwork. The term “roomware” has come to be used as a general characterization of this approach.<sup>2</sup> The roomware components are presented in detail in section 2.1.

The work presented in this thesis was initially triggered by the need to create a software infrastructure for the roomware environment. This led to the development of a software prototype called BEACH, the “Basic Environment for Active Collaboration with Hypermedia”.<sup>3</sup> BEACH provides the software infrastructure for roomware environments supporting synchronous collaboration with many different devices. It offers a user interface that also fits to the needs of devices that have no mouse or keyboard and require new forms of human-computer and team-computer interaction. To allow synchronous collaboration, BEACH is based on shared models, making information accessible through multiple interaction devices concurrently.

During its development, BEACH was restructured and refactored (Opdyke and Johnson, 1990; Roberts *et al.*, 1997; Jacobsen, 2000) several times. It became obvious that a *conceptual model* was needed to guide developers of ubiquitous computing applications. This led us to the work presented here. Parts of BEACH emerged as a software framework with an architecture that is structured according to the conceptual model for synchronous ubiquitous computing applications as developed in this thesis.

## 1.3. Goal of the Dissertation

The goal of this thesis is to provide guidance and support for developers of *synchronous groupware applications for ubiquitous computing environments*. They have to be enabled to develop applications more efficiently and with the flexibility and extensibility that is required for ubiquitous computing. Since the implementation of one single synchronous UbiComp application still requires research, it is evident that creating appropriate *development support* is even more

---

<sup>1</sup> Until summer 2001, IPSI was part of GMD Forschungszentrum Informationstechnik GmbH, the German National Research Center for Information Technology that was merged with the Fraunhofer Gesellschaft.

<sup>2</sup> “Roomware” is a registered trademark of Fraunhofer Gesellschaft (formerly of GMD).

<sup>3</sup> Concerning the creation of the BEACH acronym, please refer to (Seitz, 1997, section 5.1; Myers, 1991; Streitz *et al.*, 1999; Streitz *et al.*, 1994; Schuckmann *et al.*, 1996).

challenging. Common properties of ubiquitous computing applications have to be identified. Future developments and extensions have to be predicted. Requirements of different research areas have to be fulfilled.

The development effort can be reduced effectively if support for developers is provided at several levels. Developers need assistance when creating models of the applications to be developed, when choosing an appropriate architecture, when creating the design, and finally when implementing. This implies an architecture-driven, model-based development approach, such as described by Jacobson *et al.* (1992).

### I.4. Contributions of the Dissertation

This thesis presents a conceptual model, a software architecture, and an application framework that supports developers of synchronous ubiquitous computing software. Using the model, architecture, and framework, UbiComp developers can create applications much more easily; the applications are structured such that the reusability and extensibility as required in ubiquitous computing environments is ensured. Development effort is significantly reduced by providing a framework that handles critical requirements and offers predefined components for common functionality. It enables developers to concentrate on the task-specific parts of the implementation by providing high-level abstractions, leading to higher software quality. As a proof-of-concept, several roomware applications that support new forms of interaction have been realized.

To reduce the complexity of UbiComp software development, design and development issues are broken down to four levels. At each of the levels, this thesis contributes to the state of the art.

- (1) At the *conceptual level*, guidance for UbiComp developers has to be given by providing a conceptual model of synchronous ubiquitous computing applications that helps define their basic structure. In this thesis, the term “conceptual model” is used to describe the very high-level structure of applications of a particular domain.
- (2) Software architectures define the high-level structure for software systems. By analyzing successful software architectures in a given problem domain, structures have to be identified that are common across the whole problem domain. Therefore, at the *architectural level*, a software architecture for synchronous software applications of roomware environments has to be defined that can be reused by developers.
- (3) Software frameworks allow the reuse of implemented software architectures, offering specific support for extensibility. At the *design and implementation level*, a software application framework has to be created that provides a reusable design and reusable components that ease the implementation of roomware applications.
- (4) At the *application level*, new forms of interaction and applications for roomware environments must be developed using the conceptual model, architecture, and framework. The applications to be developed serve as a proof-of-concept and illustrate how the conceptual model, architecture, and framework are applied.

This thesis applies the conceptual model (1) to develop a *software infrastructure* supporting synchronous collaboration in *roomware environments*. The term “software infrastructure” is used here to describe everything that is needed to start building and executing applications within their destination environment. This includes the *software architecture* (2) and an implementation of this architecture, e.g. as an *application framework* (3), that can be used as a platform to easily develop applications (4) for a roomware environment (see fig. 1-3).

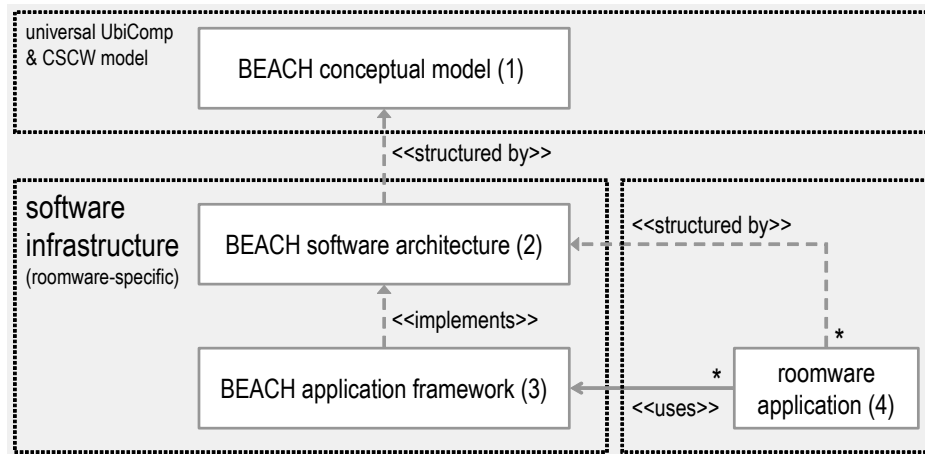


Figure 1-3. Relationship between conceptual model, architecture, framework, and infrastructure that are developed as part of this thesis. The conceptual model (1) provides the structure for the software architecture (2). The application framework (3) implements the architecture. Roomware applications (4) use the application framework. The numbers refer to the four levels of contributions.

#### 1.4.1 Conceptual Model

In order to narrow the gap between needed and available conceptual models and architectural guidance, this thesis presents the generic *BEACH conceptual model* covering all aspects of software systems supporting synchronous collaboration in ubiquitous computing environments. The model provides flexibility and extensibility for different devices. It helps identify reusable components of software systems. As the requirements for the software go beyond what is needed for traditional single-user PCs with mouse-and-keyboard interaction, it covers new interaction techniques and devices. Additionally, it supports distributed systems that collaborate synchronously. It fulfills requirements brought up by all mentioned relevant research areas (fig. 1-2). The conceptual model provides the guidance to structure the software architecture.

A *unified conceptual model* is important, since future development will cross the borders of the individual research areas. This relieves the application developer from trying to merge incompatible concepts that have been created for different purposes.

The BEACH model is structured according to *three design dimensions*:

- Separation of concerns,
- Coupling and sharing, and
- Level of abstraction.

The first dimension, separation of concerns, separates *five basic concerns*, represented as models: data, application, user interface, environment, and interaction models. Second, coupling and sharing is concerned with the *degree of coupling*, and which parts of the models are shared. The third dimension of the conceptual model is the level of abstraction. *Four levels of abstraction* are distinguished: task, generic, model, and core level.

To provide *guidance for developers* of software for roomware environments, the conceptual model helps define the high-level structure of software systems. Software developers can reuse this structure when designing software systems for ubiquitous computing environments. In this way, they can draw on proven experiences, ensuring extensibility and reusability of their software.

A clear *separation of concerns* makes it possible to adapt different aspects of a software system independently. This is important if, e.g., two devices are used to support collaboration where

## 1. Introduction

each device requires a different user interface. Therefore, the BEACH model introduces a clear separation of user interface and interaction orthogonal to the level of abstraction, which has not yet been proposed in related models. This separation is required to enable tightly-coupled collaboration using different interaction styles. For instance, a form can be visually presented by a projection, and then a keyboard or voice recognition can be used to enter text in this form, and eye tracking or hand gestures can control the input focus in the form. This way, multiple interaction modalities can be combined. This helps meet the requirement of enabling appropriate interaction in ubiquitous computing environments.

The structure that is imposed on software systems by the basic concerns is horizontally organized. As the concerns have a defined dependency that can be sequentialized, it is possible to make a *vertical cut* through the architecture of systems built according to the BEACH model. This is a key factor for improving maintainability, extensibility, and reusability of software systems in general.

Similarly, the levels of abstractions define a vertical structure for software systems, in contrast to the horizontal structure defined by the concerns. As each level builds on the lower levels only, these levels allow a *horizontal cut* through the architecture of software systems built according to the BEACH model.

The BEACH conceptual model is the first of the three main contributions of this thesis. The model provides a flexible and extensible structure for synchronous collaboration with different devices and helps identify reusable parts of a software system. It is discussed in detail in chapter 4.

### 1.4.2 Software Architecture

The conceptual model is applied to structure the *architecture* of the software infrastructure. The BEACH *software architecture* tailors the generic structure defined by the conceptual model to match the concrete needs of applications supporting synchronous collaboration in roomware environments. It can therefore be used as an example of how the conceptual model can be applied in practice. The BEACH architecture defines the software components that are necessary to use roomware components. It defines the *key abstractions* at several abstraction levels, according to the BEACH conceptual model. The identified abstractions reduce the complexity for software developers, as they hide the underlying realization and thus allow thinking at the appropriate abstraction level. This allows software developers also of non-roomware ubiquitous computing applications to reuse and adapt the BEACH software architecture, instead of having to create a new one from scratch.

The BEACH software architecture is the second of the three main contributions of this thesis. It defines important abstractions for synchronous collaboration in roomware environments to reduce the complexity for software developers. It is presented in chapter 5.

### 1.4.3 Application Framework

At the implementation level, the BEACH model and architecture have been used to design and implement an *object-oriented application framework*, in order to ease the construction of applications for roomware environments. It serves as proof-of-concept to show how the model and architecture can be successfully applied. In order to be able to design an application framework, existing applications have to be analyzed to acquire comprehensive knowledge about *common and variable aspects* of ubiquitous computing applications. The abstractions defined by the BEACH architecture serve as the basis for common parts and variable aspects. The common parts are provided by the framework as *reusable software components*; the variable aspects are

implemented as *hooks*<sup>4</sup> to enable extensibility. The design of the framework therefore contributes the abstracted knowledge about common and variable aspects of ubiquitous computing applications that has been acquired by analyzing existing applications and ongoing research efforts. This knowledge is valuable for software developers of ubiquitous computing applications.

The BEACH framework constitutes the software infrastructure for roomware environments, providing services and reusable software components. Software developers can use the framework for the implementation of roomware applications. To be able to draw on a number of existing components speeds up the development process and enhances software quality (Fayad, 1999).

To implement the BEACH framework itself, it was possible to reuse existing software components. To realize the *shared-object space* that handles the aspects related to coupling and sharing, the open-source framework COAST has been selected (Schümmer *et al.*, 2000). In addition, it proved to be feasible to adapt and extend the VisualWorks *user interface framework* (ParcPlace-Digitalk, Inc., 1995) such that it meets the requirements of roomware applications.

However, the implementation of the software infrastructure as an application framework implies certain restrictions, which might be relevant in a different setting, other than a roomware environment. To clarify the goal of this thesis, the focus of the developed software architecture and application framework is described after the discussion of related work, in section 3.5.2. In this way, this thesis can be contrasted against the state of the art.

The design of the BEACH application framework, which is the third main contribution of this thesis, is explained in chapters 6 and 7. Using this framework, developers can implement applications that acknowledge the specific properties of roomware environments much more efficiently. The design of the framework reflects abstracted knowledge about common and variable aspects of ubiquitous computing applications.

#### 1.4.4 Roomware Applications

At the application level, the thesis contributes *roomware applications* that implement new forms of interaction and provide specific functionality to support collaboration in roomware environments. To validate the approach proposed by this thesis, the applications have been constructed guided by the BEACH conceptual model, based on the BEACH software architecture, and using the BEACH application framework. They illustrate how different interaction techniques and new functionality can be implemented using the framework.

The development of the roomware applications was carried out in collaboration with other members of the AMBIENTE team and students from the Darmstadt University of Technology. Up to now, more than 15 software developers have used the BEACH model and framework to create 12 tools and extensions. Contributions of others are explicitly noted in the chapters 8 and 9.

To enable users to interact in a natural way, the users' *physical* actions—to the extent that they can be captured by computers—can serve as input for software. This way, physical objects can become part of the user interface. For this kind of interaction, a generic interaction model has been designed, and two applications that use physical input have been implemented.

The example of combined visual and acoustic presentation is used to illustrate how *new interaction modalities* can be added in order to augment other forms of interaction. This proves that the BEACH model ensures extensibility for new forms of interaction.

---

<sup>4</sup> Hooks are those parts of a framework that are designed to be extended (Froehlich *et al.*, 1997; Pree, 1999).

A suite of tools has been developed to support creative work within roomware environments. These cover all phases of creativity sessions and enable seamless transitions. The functionality provided by the BEACH application framework has been used and extended with new types of document elements and appropriate interaction forms. Handheld devices can be used for asynchronous work. They also illustrate how the BEACH model can be applied on different computing platforms.

In ubiquitous computing environments, it is important that hardware and software are integrated with the environment in which they operate, to provide a coherent whole. For instance, the software must be aware of other devices nearby. A tool has been developed within the BEACH framework that visualizes this information for the user by showing the presence of other roomware components. In this way, *synchronous collaboration* among roomware components can be initiated.

By analyzing these applications, it can be shown that the BEACH conceptual model helps create a clearly structured, reusable and extensible software design. In fact, we experienced that developers with little experience in software design and architecture—with the help of BEACH—created applications that were able to support synchronous collaboration and that could be easily extended for new forms of interaction.

Comparing one of the applications with a previous implementation not based on the BEACH model and framework provided evidence that both model and framework help software developers to significantly reduce the necessary implementation effort. In fact, the implementation effort measured by lines of code was reduced to less than one third while the robustness to failures of components was improved significantly. This example is discussed in section 8.1.

Roomware applications that are developed with the BEACH conceptual model, architecture, and framework serve as a proof-of-concept. They provide new interaction techniques and functionality for synchronous collaboration in roomware environments. The new forms of interaction that have been implemented are presented in chapter 8, new tools for roomware environments in chapter 9.

In order to provide the basis for the overall discussion, the thesis starts with a thorough analysis of the requirements coming from the research areas that influence the work within roomware environments. The identified requirements are not only relevant for roomware environments, but they also describe important aspects of all kinds of ubiquitous computing software systems.

### 1.5. Outline of the Dissertation

---

↓ Part I:  
The BEACH  
Conceptual Model

This thesis consists of four parts. The first part starts with an analysis of the requirements for the software infrastructure of ubiquitous computing environments (chapter 2) by examining existing applications for computer-supported cooperative work, co-located teamwork, and ubiquitous computing environments. Subsequently, existing solutions are analyzed with respect to the extent which they can fulfill the requirements (chapter 3). Based on the requirements and the insights gained while examining related work, the conceptual model is defined and its applicability is discussed (chapter 4).

---

↓ Part II:  
Architecture and  
Framework

The second part describes an application of the BEACH conceptual model. Chapter 5 defines the BEACH architecture, tailoring the conceptual model for the concrete needs of the roomware components developed by the i-LAND project. Then, the BEACH architecture is implemented as two software frameworks. The BEACH Model framework covers the implementation of the core and model layer (chapter 6). The BEACH Generic Collaboration framework implements the generic layer of the BEACH architecture (chapter 7). Both frameworks are described in a bottom-up fashion.

---

↓ Part III:  
Applications for  
Roomware

The third part of this thesis describes applications of the BEACH conceptual model and software framework. The applications validate the usability of model and framework and provide evi-

dence that they ease application development. In addition, they illustrate how their features can be applied. Chapter 8 presents examples where BEACH has been extended to support new forms of interaction. Chapter 9 describes some tools that have been developed on top of the BEACH framework. Chapter 10 presents conclusions, also discussing open questions and directions for future work.





## Part I. The BEACH Conceptual Model: Guidance for Ubiquitous Computing Developers

---

Part I of this thesis provides support for ubiquitous computing developers. It starts in chapter 2 with analyzing the requirements for the software infrastructure of ubiquitous computing environments. This is achieved by examining existing applications for computer-supported cooperative work, co-located teamwork, and ubiquitous computing environments. Chapter 3 investigates to what extent related work fulfills the identified requirements. The conducted analysis reveals that existing models and systems cover the requirements of their original domain only; they fail to meet requirements that originate from other research areas. Especially, it becomes apparent that CSCW models and frameworks fail to cope with the requirements of ubiquitous computing, while UbiComp systems offer no adequate support for synchronous collaboration. To fill this gap, chapter 4 presents a conceptual model for synchronous ubiquitous computing applications that meets all addressed requirements. Its applicability is discussed, also comparing it to related approaches. This comparison demonstrates that the presented model comprises—and exceeds—what can be expressed by the current state of the art.

---



## 2. Requirements of Ubiquitous Computing Applications

---

This chapter analyzes the requirements of the software infrastructure of ubiquitous computing environments. It starts with a presentation of the roomware components developed in the i-LAND project and describes important application scenarios. This description constitutes the background for the requirement analysis, pointing to an initial set of relevant aspects. The subsequent sections present work within the areas of human-computer interaction, ubiquitous computing, computer-supported cooperative work, and software engineering to broaden the view on the problem domain beyond the roomware developed by the i-LAND project. The given samples are examined and structured in order to derive requirements of synchronous ubiquitous computing applications. Finally, the identified requirements are summarized in the last section of this chapter.

---

In order to develop a conceptual model for ubiquitous computing applications supporting synchronous collaboration, this chapter analyzes general properties and requirements of such applications. To get a broad view, several examples from all relevant research areas are scrutinized to find common aspects. Properties of the roomware components are presented before the specific aspects of the related areas are researched. Synchronous roomware applications are examples of applications placed at the intersection of HCI, CSCW, and UbiComp (fig. 1-2). Analyzing the roomware components is also relevant to design an appropriate software infrastructure, which is taken as a sample application for the conceptual model.

### 2.1. The i-LAND Roomware Project

This section presents the roomware components that were developed as part of the i-LAND project at Fraunhofer IPSI. As the BEACH architecture and framework constitute the software infrastructure for these components, the design and functionality of the roomware components

had a major influence on the requirements. Their properties are also relevant in a broader context of ubiquitous computing environments.

Until now, two generations of roomware components have been developed. The first generation comprised a large interactive wall, called DynaWall, and the initial designs of an interactive table and interactive chairs, called InteracTable and CommChairs. In 1999, the second generation of roomware components was developed together with partners from industry as part of the R&D consortium “Future Office Dynamics” (FOD, 2002), and presented to the public in late 1999. Apart from redesigns of the InteracTable and the CommChair, two new components were added, the ConnecTable and the InterWall, a mobile information appliance with a large vertical display. In the meantime, two roomware components, the InteracTable and the CommBoard (a redesigned version of the DynaWall using different technology), have become commercially available (Wilkhahn, 2002).

↓ Roomware  
overview

The first three roomware components created as part of the first generation are presented in this section: the DynaWall, the CommChair, and the InteracTable. The ConnecTable is described in chapter 8, as it was developed after the BEACH framework was already implemented. The ConnecTable was used to test the flexibility and extensibility of the BEACH architecture and framework. The InterWall is presented elsewhere (FOD, 2002; Englisch, 1999), as their interaction capabilities are very similar to those of the other components and require no special software features. The description of roomware components also includes forward references to the requirements analyzed and categorized later.

### 2.1.1 DynaWall

The DynaWall is a large interactive wall with a display size of 4.50 m width and 1.10 m height. It consists of three touch-sensitive back-projection units<sup>5</sup> that are integrated in one wall of the room. The DynaWall is shown in figure 2-1. Each segment is driven by a separate PC that receives input from the touch-sensitive surface and has an LCD projector attached as a display. Without any special software support, the DynaWall works just like three computers with large displays placed next to each other. The DynaWall is similar to the interactive wall constructed at Stanford (Stanford University, 2000).<sup>6</sup>

In order to combine the segments to form a homogeneous interaction area, the display output of the three computers has to be synchronized. The output must be generated in such a way that objects being moved off one side of a display will appear on the other side of the adjacent display. This is a requirement that originated directly from the design of the DynaWall (see req. U-2, on page 24 below).

Other issues arise from the size of the DynaWall. Large interactive surfaces raise the need for new forms of interaction, as it is, e.g., not possible (or convenient) to easily reach all objects on the wall (Winograd and Guimbreti re, 1999; Swaminathan and Sato, 1997; Pier and Landay, 1992). To re-arrange objects directly on a large visual surface while standing in front of it, it has been proposed to enable throwing of objects from one user to another (Gei bler, 1998; Streitz *et al.*, 2002).

---

<sup>5</sup> The DynaWall at IPSI uses three dismantled SMART boards (SMART Technologies Inc., 2002).

<sup>6</sup> The interactive wall at Stanford was constructed some years after the DynaWall was built.



Figure 2-1. Two people working at the DynaWall

Apart from that, interaction with pen or finger can also benefit from an adapted interaction style. This leads to the requirement to support different forms of interaction (see req. H-1 below).

### 2.1.2 CommChair

The CommChair is a mobile chair, which has an embedded computer with a pen-sensitive display attached at one side (see figure 2-2). To ensure maximum flexibility and mobility, each chair is provided with a wireless network and an independent power supply.

While a CommChair can be used as a comfortable single-user device, it develops its full power when used collaboratively. In this situation, it can be used to share information among a small group of coworkers in order to support discussions (see req. C-1 on page 28). Alternatively, CommChairs can be used to access both personal and public information (Greenberg *et al.*, 1999; O'Hara *et al.*, 2002) when used together with a public interaction device such as the DynaWall (see also figure 2-4). In this case, the collaboration is more complicated, as the involved devices differ significantly with respect to available display area. This requires allowing collaboration with heterogeneous devices (see req. UC-1 on page 30).

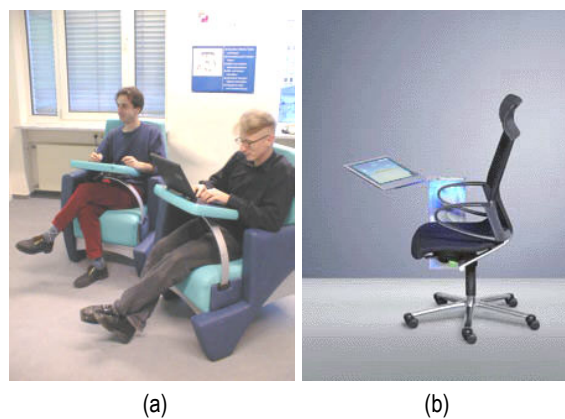


Figure 2-2. (a) The initial two prototypes of CommChairs developed by IPSI. (b) The redesigned version of the CommChair being part of the second generation of roomware components that have been created by IPSI and Wilkhahn/Wiege within the “Future Office Dynamics” R&D consortium.

### 2.1.3 InteracTable

The InteracTable is designed to support informal discussions of small groups (figure 2-3). It is an interactive table that can be used by up to six people standing around it. It has a touch-

## 2. Requirements of Ubiquitous Computing Applications

sensitive display in a horizontal position, allowing people standing around the table to look at the display from any side.

However, this has implications for the interaction (Streitz *et al.*, 2001; Kruger *et al.*, 2003; Scott *et al.*, 2003; Tandler *et al.*, 2002a; Hancock *et al.*, 2002; Shen *et al.*, 2002).

While vertical displays have, for example, a defined top, bottom, left, and right, a horizontal interaction area has no predefined orientation. Furthermore, multiple users are often working at a table concurrently. Therefore, traditional user interface concepts and interaction techniques need to be augmented (see req. H-2, UH-1, C-3).

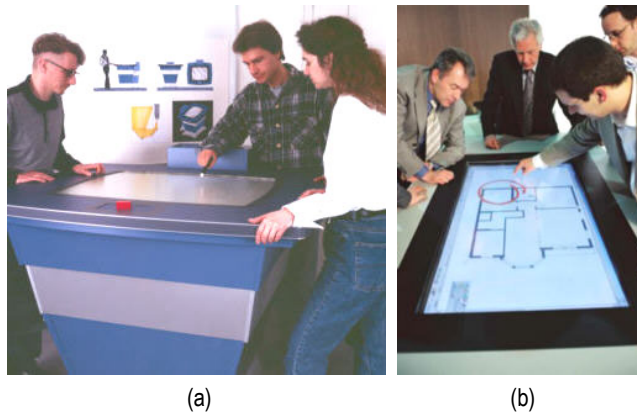


Figure 2-3. (a) First prototype of the InteracTable, developed by IPSI. (b) Second version of the InteracTable, which is commercially available. It allows collaboration of up to six co-located people.

### 2.2. Application Scenarios of Synchronous Collaboration within Roomware Environments

This section illustrates typical work situations in a roomware environment by presenting three scenarios. The situations described in the scenarios are used in section 7.6 to show how these can be realized with the proposed architecture.

The scenarios illustrate a meeting of a small group. A marketing team is working on the advertising strategy for a new product. The leader of the team has prepared an agenda for the meeting in advance.

#### 2.2.1 Group Discussions with the DynaWall and CommChairs

At the beginning of the meeting, all participants assemble in the meeting room equipped with roomware components. One side of the room is completely covered with a DynaWall. The leader walks up to the DynaWall to show the prepared agenda. The others take seats in one of the available CommChairs.

After a brief introduction, the group starts a discussion about the open issues of the strategy. As each member uses a CommChair, everyone can take private notes. In addition, every CommChair has access to the information displayed at the DynaWall. This enables everyone to directly interact with the shared information. All ideas can be collected in a shared document; everyone can write down ideas, draw illustrations of parts of the advertising, or write annotations on contributions made by others. Moreover, material that was collected in preparation for the meeting can be moved from the personal workspace to the shared document. As the team works in a tight collaboration mode, all modifications made by a team member in a CommChair are immediately displayed at the DynaWall.

For the users in CommChairs, it is helpful that they can easily look at the DynaWall. Since the display of the CommChair is much smaller, they can use the DynaWall to get an overview of all created material. The CommChair is used to select the focus area for their contribution.

Finally, the team decides that all major issues have been identified. They agree to split into subgroups. This facilitates working on different issues in parallel. They distribute the material necessary to work on the selected issues to different sub-workspaces, which are created by simple pen gestures at the DynaWall.

### 2.2.2 Tight Collaboration between CommChairs and the DynaWall

One subgroup continues working at the DynaWall. A user at the DynaWall continues a discussion about one of the selected issues together with another user still sitting in a CommChair (see figure 2-4). They need a lot of space to sketch sequences for TV spots, so they frequently have to switch between different workspaces. As they work loosely coupled with respect to the other subgroups, their navigation commands do not affect the other group's currently visible workspace.



Figure 2-4. Tight collaboration, using a CommChair in front of a DynaWall. The user in the chair can remotely interact with information displayed at the wall by using the display attached to the chair. In addition, he can access his personal workspace.

In this situation, several subgroups share the same room, but they are able to work *independently*. Still, if two groups access the same information object, they share the same object. For example, if the subgroups need the list of identified issues, in order to add further items that came up in their subgroup discussion, they share the same copy of the list and can immediately see the modifications made by others. This is helpful in eliminating the need for merging different versions of the same object, and to create awareness about the other groups' activities in an unobtrusive manner.

### 2.2.3 Cooperative Work at an InteracTable

Two other people forming another sub-group move over to an InteracTable separated by a movable partitioning wall. They use the table for brainstorming about possible customer profiles. They write down their ideas and sketches on small digital brainstorming cards. Standing around the table, they shuffle new cards over to each other to increase creativity by supplying new associations. Since they look from different positions at the table, the cards automatically rotate when thrown to someone else. After a while, they decide that they have enough material and start incorporating their ideas into a drawing. To be able to look at the emerging drawing simultaneously, one team member opens a *second view* on the drawing and rotates it towards her. This way, they both can watch and work with the same workspace, but each with the preferred orientation (fig. 2-5).



Figure 2-5. To support horizontal displays, BEACH lets different users rotate documents to a preferred orientation. For collaboration, a second view of a document can be opened that remains synchronized with the original.

### ↓ Chapter preview

The next sections derive requirements for the software infrastructure for roomware environments that can be identified by analyzing the relevant research areas (fig. 1-2) to broaden the scope. This facilitates the identification of requirements not relevant to the currently existing roomware components. The following is an updated version of the requirements published in (Tandler, 2001b).

The requirements are organized by the research area to which they are related. To identify the requirements, the following abbreviations are used:

- “H” – HCI: User Interface and Interaction requirements (section 2.3)
- “U” – UbiComp: Ubiquitous Computing Environments (section 2.4)
- “C” – CSCW: Applications for Synchronous Collaboration (section 2.5)
- “S” – SW: Software engineering requirements (section 2.6)

### 2.3. HCI: User Interface and Interaction

In the area of computer-human interaction, several new user interface concepts are being developed that go beyond the traditional WIMP (windows, icons, menus, pointing) metaphor (Beaudouin-Lafon, 2000). This section presents some important developments that influence the development of ubiquitous computing environments. The discussed developments are analyzed to form requirements for the software infrastructure for roomware environments. They are grouped in two categories.

- *New forms of interaction* present new interaction techniques and devices.
- *New user interface concepts and elements* describe new concepts that are originally designed for traditional desktop PCs, but seem to be very helpful in the context of roomware components.

#### 2.3.1 New Forms of Interaction

New devices with different properties require new forms of interaction. This section concentrates on the consequences of pen and gesture input, as well as non-visual interaction.

With the availability of new input devices besides mouse and keyboard, new interaction techniques started to develop. These aim to providing a more efficient, more direct, transparent, and natural interaction (Baudel and Beaudouin-Lafon, 1993; Abowd, 1999; Abowd and My-natt, 2000).



In the context of roomware environments, these new interaction techniques are relevant as within meeting situations little distraction should occur. In particular, complicated interaction should be avoided (Prante *et al.*, 2002).

Within informal meetings, pen input to public whiteboards facilitates more natural and adequate interaction (Pedersen *et al.*, 1993; Haake *et al.*, 1994; Meyer and Bederson, 1998; Shipman and Marshall, 1999). Additionally, pen input has the benefit of having one level of indirection less in the interaction compared to mouse-input, as the pen is operated directly at the position where the output is rendered. (In contrast, the mouse is operated on a horizontal surface besides the display.) In the Tivoli system, pen-input is implicitly structured to combine the informal character with the benefits of structured data types (Moran *et al.*, 1995).

To provide natural interaction, it is appropriate to use pen or hand gestures (Bolt, 1980; Baudel and Beaudouin-Lafon, 1993; Henry *et al.*, 1991; Rekimoto and Matsushita, 1997). When gestures are interpreted by software in order to invoke commands, the software must be aware that input can be ambiguous and detection error prone (Mankoff *et al.*, 2000b; Mankoff *et al.*, 2000a).

In some situations, it might be more appropriate to use speech as input or audio output (Müller-Tomfelde and Steiner, 2001; Mynatt *et al.*, 1998; Coen, 1998).

While a common interface can be provided for different visualizations at a rather low level describing the visual appearance, another approach has to be taken for other modalities. One such idea is the separation of the models for the abstract and the physical user interface (Thevenin and Coutaz, 1999). Depending on the available output device, it might be possible to generate the physical user interface automatically from generic elements (Myers *et al.*, 2000, p. 13). In general, it is helpful if all different interaction models use a common interface for the underlying functionality.

Facing all these different forms of interaction that apply to different situations, it is important for ubiquitous computing software to be open for different styles of interaction and to be extensible for future developments (Myers *et al.*, 2000, p. 15 f). However, current operating systems and platforms only offer direct support for “traditional” interaction techniques and devices. The ubiquitous computing software must allow the integration of other device drivers in an extensible manner (Abowd, 1999, p. 82).

The idea of different forms of interaction is extended by Myers *et al.* (2002). They define the term “flexi-modal” as the ability to simultaneously and flexibly mix different interaction modalities.

Different interaction styles, such as pen, speech, or gestures, require the introduction of new interaction models. For example, pen input cannot be dispatched to one single point at a display, as it might affect a wide area on the screen. Speech or gesture recognition requires different processing levels. For each of these types of input, abstractions must be defined that can be easily mapped to the invocation of functionality (Abowd, 1999, p. 81). For mouse, keyboard, and pen input, events are a useful abstraction, but for other types of input, other concepts might be more appropriate (Myers *et al.*, 2000, p. 24). Other interaction techniques, such as hardware buttons often found in PDAs (personal digital assistants), offer a very similar functionality compared to software button widgets.

### *Requirement H-1: Different Forms of Interaction*<sup>7</sup>

Ubiquitous computing software must be able to handle different forms of interaction. It must be possible to simultaneously and flexibly mix interaction forms. Different forms of interaction have different characteristics requiring different abstractions and different handling of input and output.

#### 2.3.2 New User Interface Concepts

New devices and interaction techniques offer new possibilities for the construction of user interfaces. While the classical user interface style with windows, menus, and toolbars was designed for a traditional desktop PC with mouse and keyboard, new user interface concepts have been developed that consider the properties of new interaction devices.

Having a pen as an input device (see previous section) offers the possibility to employ strokes drawn with a pen as part of the interaction with the user interface. For example, pen-gestures can supplement a traditional user interface. In order to reflect the properties of the pen as an input device, new user interface elements can be designed. Consequently, researchers have developed various types of icons and menus.

For example, *Marking Menus* (Kurtenbach and Buxton, 1994) extend pie menus (Callahan *et al.*, 1988) for usage with pens. In addition to selecting a menu item, a stroke can be drawn to the direction of the menu item without waiting for the menu to open. This is a short cut that skips drawing the menu.

The *FlowMenu* (Guimbretière and Winograd, 2000) combines the previously explained idea with mechanisms developed for pen-based entry of text (Perlin, 1998; Mankoff and Abowd, 1998). They allow selecting a command together with sub-commands or arguments by drawing a pen-gesture instead of navigating through a hierarchical menu.

Another variation is *Gedrics* (Geißler, 1995; Geißler, 2001). *Gedrics* are gesture-driven extensions of icons that respond to pen gestures. Users can either select a command from a popup-menu or draw the corresponding pen gesture over the *Gedric*.

The devices found in roomware environments differ in terms of their characteristics, e.g. display size (Weiser, 1991; Myers *et al.*, 2000). The requirements imposed by the size of the display led to the development of different user interface concepts. The following presents examples for new concepts for managing display space.

Large interactive displays are an example of how the physical properties of the devices impose requirements on the user interface. Classical window-interfaces typically provide menus at the top of the window. However, on a large display, this can be inconvenient or too high to reach for short users (Pier and Landay, 1992).

Using large interactive display surfaces, overlapping windows can obscure important pieces of information. Therefore, researchers experimented with non-overlapping segments. *Flatland* defines segments that are automatically shrunk if space is needed (Mynatt, 1999b; Mynatt, 1999a).

In *Tivoli* (Moran *et al.*, 1997; Moran *et al.*, 1995), a single surface is used. The surface can be structured by “boundaries” and “regions”. Boundaries can be manipulated directly by the user, while regions are implicitly computed for every selection.

*Translucent patches* (Kramer, 1994) extend the traditional window concept. Translucency inhibits that underlying information is obscured. In addition, the patches can have freeform shapes.

---

<sup>7</sup> Requirements related to human-computer interaction are denoted with a capital “H”.

*Pad++* (Bederson *et al.*, 1996) uses the concept of infinite zooming to overcome the restriction of a finite available surface. *Fisheye views* or *multi-scale interfaces* have a variable zoom factor. The part of a document having the focus is magnified, while less important parts are shrunk (Robertson and Mackinlay, 1993; Furnas and Bederson, 1995). Guimbretière *et al.* (2001) applied this idea to space management of interactive walls. *ZoomScapes* have an associated scaling value associated with each point on the surface. This allows objects to be put aside that are currently out of focus and, therefore, do not need to be visible at a high resolution.

Different concepts for user interfaces must be defined for very small displays, e.g. personal digital assistants. Prante *et al.* (2002) describe the “navigation stack”: spatially arranged information objects can be viewed in two different modes, which show the spatial relationship or the details, respectively. The navigation stack helps to switch quickly between different objects.

Due to the differences of the interaction capabilities of different devices, it is not sufficient to support different forms of interaction (req. H-1) that use the same user interface. Moreover, a particular user interface concept is not feasible for all forms of interaction and all devices (Myers *et al.*, 2000, p. 16). For example, in handheld computers, parts of the user interface will be built into the hardware itself, such as the physical buttons and switches. These buttons allow only one interaction, namely to press them physically. Consequently, physical buttons that are part of a user interface cannot be used with other interaction forms.

To reuse application functionality for different devices requiring different user interfaces, a conceptual model for ubiquitous computing applications must separate the user interface from application issues that encapsulate the application logic and are independent of the user interface. This is necessary for providing an appropriate user interface for every device available. Depending on the number of different devices, the appropriate user interface can be manually implemented by the developer for every supported class of devices. Other researchers are developing device-independent descriptions of user interfaces that allow the dynamic creation or selection of user interfaces (Ponnekanti *et al.*, 2001; Banavar *et al.*, 2000).

#### *Requirement H-2: Different User Interface Concepts*

For ubiquitous computing software, it is essential to separate the application functionality from the user interface. This is to ensure that based on the available interaction devices an appropriate user interface can be supplied to access the functionality. To increase reuse, user interfaces should be designed to enable different forms of interaction.

## 2.4. UbiComp: Ubiquitous Computing Environments

Ubiquitous computing, as foreseen by Mark Weiser (1991), is still a vision. Computing devices are not yet able to cooperate in an unobtrusive way. Although named “ubiquitous”, ubiquitous computing is not yet available *everywhere*. When speaking of ubiquitous computing *environments*, this often refers to *small* environments only, e.g. a room or building within which devices have access to their network resources. To be *truly* ubiquitous, a ubiquitous computing environment has to be scaleable to cover large areas. Environments have to be able to exchange information and devices. However, the currently existing ubiquitous computing environments will grow. Researchers will develop techniques and protocols to exchange devices and information among each other. Finally, the developed technology has to be standardized by industry to ensure wide-scale interoperability.

This section first analyzes properties of ubiquitous computing environments. Subsequently, issues of human-computer interaction within UbiComp environments are discussed.

↓ Section outline

## 2. Requirements of Ubiquitous Computing Applications

### 2.4.1 Properties of UbiComp Environments

Ubiquitous computing environments have three main properties (Weiser, 1991) that constitute requirements for software systems for ubiquitous computing environments:

- A ubiquitous computing environment consists of *multiple heterogeneous networked devices*.
- As most devices are mobile, they are likely to be carried around, raising the need of *dynamic configuration*.
- The devices are not treated “in isolation”; they are always seen related with their current *environment*.

These properties directly bring up the following requirements.

#### Multiple and Heterogeneous Devices

The most prominent property of a ubiquitous computing environment is the ubiquitous presence of devices with embedded computing facilities (Weiser, 1991). These devices come in many different sizes and shapes depending on the tasks that they are designed for (Weiser, 1993). Ubiquitous computing software must therefore be able to deal with the device heterogeneity (Garlan, 2000; Brummit *et al.*, 2000; Sousa and Garlan, 2002). This includes a communication infrastructure for message exchange and for providing access to shared data (Myers *et al.*, 2000; Esler *et al.*, 1999; Kon *et al.*, 2002).

#### *Requirement U-1: Multiple and Heterogeneous Devices*<sup>8</sup>

Ubiquitous computing software must be able to deal with environments containing multiple heterogeneous devices.

#### Multiple-Computer Devices

Some devices that are perceived by a user as a single element might actually consist of many individual hardware components. Some roomware components (e.g. the DynaWall, see fig. 2-1) are composed of several segments, each of them running on a separate PC. Due to hardware limitations,<sup>9</sup> this configuration allows each segment to receive pen input by one user only at a time, thereby supporting several users working simultaneously. To give the user the impression of a homogeneous interaction area, the segments must therefore be coupled via software. This facilitates multiple users collaborating on the same visual interaction area despite the limitations of hardware or physical space.

#### *Requirement U-2: Multiple-Computer Devices*

Ubiquitous computing software must support devices that have multiple embedded computers. It must be possible to coordinate and couple the software running on these computers.

#### Context and Environmental Awareness

The devices within a ubiquitous computing environment are not treated in isolation; instead, they are often perceived within the context of their environment. Software being aware of its context can act depending on the state of the surrounding environment (Schmidt *et al.*, 1999; Dey, 2000; Schilit, 1995; Mills and Scholtz, 2001). Therefore, it is also important for the system to have a “deeper understanding of the physical space” (Brummit *et al.*, 2000).

Example 2-1:  
Context

*Common examples of relevant context information are the current location of devices (Weiser, 1991), specific users, and the kind of device on which a software application is actually running (Chen and*

<sup>8</sup> Requirements related to ubiquitous computing are denoted with a capital “U”.

<sup>9</sup> When designing and implementing BEACH, each SMART Board (SMART Technologies Inc., 2002) could only recognize a single pen position at a time.

Kotz, 2000; Abowd and Mynatt, 2000). Apart from the physical environment, other contextual information—such as the current task or project, or presence of co-workers—could influence the behavior of the software, as far as this information is available to the application (Sousa and Garlan, 2002; Schmidt, 2000; Ambient Agoras, 2003; Thevenin and Coutaz, 1999).

The software infrastructure must therefore maintain a *representation of the current context*. In order to be able to update this representation, an interface to sensors collecting context information distributed all over the environment is needed. Similar to what was described for different input devices, the data collected by the sensors will normally need to be pre-processed in order to generate information at a level of abstraction useful for the application (Salber *et al.*, 1999).

If context changes are detected, mechanisms must exist to notify the application about the changes. This way, resource awareness can be realized, which is one of the requirements for ubiquitous computing environments identified in (Garlan and Schmerl, 2001). In fact, resource awareness extends to *environmental awareness* as it includes not only physical resources but also the other kinds of context information, named above.

**Requirement U-3: Context and Environmental Awareness**

Ubiquitous computing software is integrated with its environment. It must draw on context information to provide an adapted behavior and tailored functionality.

**Dynamic Changes**

Ubiquitous computing environments are highly dynamic. During meetings it often occurs that several independent problems have to be solved in parallel. In such situations, a team usually splits into several subgroups, each of which tries to solve one of those problems. After a given time, the team regroups and all solutions are presented. This scenario shows that different kinds of collaboration modes must be supported within a ubiquitous computing environment. Each of these will require a different configuration of available devices.

In (Sousa and Garlan, 2002), dynamic changes of users entering and leaving an environment, of the environment, of the task, and of the context (like privacy preferences or the current user activity such as sitting or driving) are distinguished. Configurations change dynamically due to devices, which are brought in and taken away (Coen *et al.*, 1999; Shafer *et al.*, 2001). Handling the dynamic reconfiguration was identified by Garlan (2000) as a key issue for software architectures of UbiComp environments.

The dynamics of work practices must therefore be reflected by the design of the software. The design should be flexible enough to give a team the necessary freedom to work efficiently.

**Requirement U-4: Dynamic Configuration**

As ubiquitous computing environments are highly dynamic, software must deal with dynamic change as well. This implies that it must be possible to dynamically adapt the software configuration.

**2.4.2 Human-Computer Interaction Issues in UbiComp Environments**

Ubiquitous computing environments impose new challenges on human-computer interaction. The requirements that are found by analyzing interaction issues in the context of ubiquitous computing are also part of the requirements for roomware environments.

**Adapted Presentation for Different Devices**

Due to the different form-factors of interaction devices in a ubiquitous computing environment, displays appear in a wide range of different sizes and with different orientations (req. U-1). For differently-sized devices, different scaling factors, a different representation, or a different selection of objects must be used (Russell and Weiser, 1998).

Hence, we need to consider whether the user needs an overview of the entire document, or just the part of the document that is being edited. The more the devices differ, the harder it is to use the same user interface for all devices (Abowd, 1999, p. 81; Myers *et al.*, 2000, p. 16). For different devices, other interface metaphors and concepts become more appropriate.

Example 2-2:  
Interactive table

*A specific problem is to display information on an interactive table. The output does not necessarily have a common top–bottom / left–right orientation for all users working at an InteracTable (see scenario in section 2.2 and implementation in section 7.6.3), as different users can look at the surface from different positions (Streitz *et al.*, 1999). At a normal table, people would simply rotate a sheet of paper to show it around (Kruger *et al.*, 2003). At an interactive table, it is desirable that the same be possible. Furthermore, users should be able to keep a view of this object oriented towards them. This way, each user can look at the object with the preferred orientation (Hancock *et al.*, 2002; Shen *et al.*, 2002; Bruijn and Spence, 2001).*

### Requirement UH-1: Adapted Presentation

Ubiquitous computing software must be able to adapt its presentation to the characteristics of the device it is currently running on and the interaction possibilities that are available.

### Multiple-Device Interaction & Cross-Device User Interfaces

In a ubiquitous computing environment, a user usually has access to more than one computer (Myers *et al.*, 2000). Within a meeting, a user might leave an interactive chair and walk up to a large public presentation surface. Here, the software must be able to re-detect and quickly re-assign the used device to give the user access to private information, for instance to the prepared material for a presentation.

Continuing this scenario brings up a different case of multiple-device interaction: the user giving the presentation might have access to another device in parallel to the public display. To view her private annotations in addition to her slides, she uses an electronic lectern or a PDA (Myers, 2001). This means that she uses several devices simultaneously with the same information displayed on both devices—but within a different context that influences the resulting view (different size, different level of detail, private annotations). This relates to the adapted presentation (req. UH-1) where the context is defined by the used devices in contrast to the usage of the device (see also the scenarios in section 2.2).

There are many examples in literature where a PDA-like device is used concurrently with a digital whiteboard, a table, or PC (Fox *et al.*, 2000; Myers *et al.*, 1998b; Myers *et al.*, 2002; Rekimoto, 1998b; Rekimoto, 1998a). There, a PDA is used to have access to additional information or functionality without consuming space of the main display. These are examples of multi-machine or *cross-device user interfaces* (Myers, 2001). In general, interaction devices that are present in the environment can be dynamically employed to extend the interaction capabilities of a mobile device (Pierce and Mahaney, 2004).

Pick-and-drop (Rekimoto, 1997) is an interaction technique for transferring data between different computers. Hyperdragging (Rekimoto and Saitoh, 1999) enables easy exchange of information between laptops and interactive tables or walls.

On displays that are placed in public spaces, the PDA can be used for visualizing private information (Greenberg *et al.*, 1999). Likewise, the PDA can be used to access only the functionality that is currently relevant for its user. In these cases, both devices show different information and offer a different functionality.

As part of the Pebbles project (Carnegie Mellon University, 2000), a technique called “snarfing” has been developed to use a PDA to remotely control another device (Myers *et al.*, 2001). This is done by displaying parts of the user interface on the PDA. A different approach is

taken in (Ayatsuka *et al.*, 2000), where the PDA is used as a 6-degree-of-freedom<sup>10</sup> input device for an interactive surface.

**Requirement UH-2: Multiple-Device User Interface and Interaction**

Ubiquitous computing software can benefit from the presence of multiple devices, if multiple devices can be involved in interaction and the user interface can cross the boundaries of a single device. The software must enable that a distributed user interface benefits from the different properties of all available devices.

Interacting with Physical Objects

Since the configuration of physical objects in, for example, a meeting room depends on the current work mode of a team, changes made to “real” objects can be used to trigger actions of software (Ishii and Ullmer, 1997; Shafer *et al.*, 2001). In particular, it is useful to reflect adaptations made by users to the setting of devices, due to changes of the current collaboration mode.

There are situations in which a state change of the software is relevant to the maintenance of the consistency of the “real” and the “virtual” parts of the world (see also req. U-3). Rekimoto and Saitoh (1999) have augmented a table to allow links between physical and virtual objects. Digital information can be dragged off laptops that are placed on the table to physical objects. At MIT, several software systems with tangible user interface have been created (Ullmer *et al.*, 1998; Patten *et al.*, 2001; Pangaro *et al.*, 2002). Cooperstock *et al.* (1997) describe the reactive room, a computer-augmented video-conferencing environment with automatic behaviors and physical interaction elements. Designer’s outpost is an augmented interactive wall, which accepts physical post-it notes to provide input (Klemmer *et al.*, 2001).

For all these examples, the software includes physical objects as part of the user interface.

**Requirement UH-3: Physical Interaction**

Ubiquitous computing software has to support physical objects as user interface elements (in a generalized sense). It must be able to track user actions with physical objects and use this to trigger software functionality.

## 2.5. CSCW: Synchronous Collaboration

The field of computer-supported cooperative work is concerned with building *groupware*, i.e. “computer-based systems that support groups of people engaged in a *common task* (or goal) and that provide an interface to a *shared environment*” (Ellis *et al.*, 1991). The term “groupware” was initially coined by Peter and Trudy Johnson-Lenz in 1981 (Johnson-Lenz and Johnson-Lenz, 1994).

Groupware applications can be classified in terms of time and space: they can support people working together at the same time or at a different time, and also, being in the same (physical) place or in different places (Ellis *et al.*, 1991). This thesis concentrates on supporting work at the same time *and* in the same place, called *synchronous co-located* collaboration. However, the BEACH model and framework can be used for distributed collaboration as well.

This section analyzes aspects of synchronous groupware. These are used to identify the requirements related to synchronous collaboration—but put into context of ubiquitous computing. In this respect, two issues of CSCW applications are important: meeting support and single display groupware. These are discussed in the context of UbiComp environments.

↓ Section outline

<sup>10</sup> The six degrees are motion in three dimensions and rotation in three dimensions.

## 2. Requirements of Ubiquitous Computing Applications

### 2.5.1 Meeting Support

A prominent application area of groupware tools is meeting support. Meeting support systems can be divided into a number of categories, ranging from remote-conferencing to electronic meeting systems and process support tools. A general model for classifying the functionality of groupware systems is the “clover model” (Calvary *et al.*, 1997; Graham and Grundy, 1999). It distinguishes three “spaces” of functionality: the production, coordination, and communication space. The focus of this thesis is the support of the production space. Within meeting rooms, many coordination tasks can be carried out by social protocols. Likewise, *computer* support for communication and awareness is not required for people working together in the same room.

An example for tools enabling distributed meetings is TeamRooms (Roseman and Greenberg, 1996a). Further examples are discussed in (Ellis *et al.*, 1991). The benefit of the “room” metaphor used in TeamRooms is that it can be used by both individuals and groups, as well as in asynchronous and synchronous work modes (Greenberg and Roseman, 2003).

Other systems focus on meetings where several participants are located in the same room. In this case, all participants often have access to a private workstation (Stefik *et al.*, 1987a; Nunamaker *et al.*, 1991). Since all participants are co-located, awareness and communication features become less important than in the distributed case. These kinds of systems can be called “co-present groupware” (Stewart *et al.*, 1999).

Depending on the type of meeting, whether it is more formal with strict process and floor control or whether it is an informal meeting, the used groupware systems need different features (Greenberg, 1991). GroupSystems (Nunamaker *et al.*, 1995), for example, is able of handling formal processes for meetings with a large number of participants. Tivoli (Pedersen *et al.*, 1993; Moran *et al.*, 1998b; Moran *et al.*, 1998a) on the other hand, is an example for informal meeting support. Other systems such as DOLPHIN (Streitz *et al.*, 1994; Haake *et al.*, 1994) or the Colab suite of tools (Stefik *et al.*, 1987a; Stefik *et al.*, 1987b) aim to allow various working styles.

Independent of whether or not all participants reside within the same room, multiple devices are used to support direct participation of people. This means that the software needs to deal with the requirements imposed by distributed systems. This includes issues such as session management, coupling control, and concurrency control (Schuckmann *et al.*, 1999; Greenberg and Roseman, 1999). Concurrency control is of particular importance for highly dynamic synchronous collaboration occurring in meeting situations. When people are working concurrently within the same shared workspace, software infrastructure and applications must be designed to make people’s actions unlikely to interfere.

#### *Requirement C-1: Multi-Device Collaboration<sup>11</sup>*

Many ubiquitous computing applications have to support collaboration. Involving several devices, these applications are necessarily distributed systems that have to cope with CSCW issues such as synchronization, concurrency, and consistency. To enable synchronous collaboration, information must be shared among devices.

The goal of this thesis is not the development of a groupware infrastructure. It rather investigates the applicability and influence of available groupware technology in the context of ubiquitous computing. This requirement is therefore rather to be understood as a pointer to groupware in general; a detailed discussion of groupware requirements can be found in groupware literature (Graham and Grundy, 1999; Schuckmann *et al.*, 1999; Roseman and Greenberg, 1992).

---

<sup>11</sup> Requirements related to computer-supported cooperative work are denoted with a capital “C”.



### Coupling and Collaboration Mode

As mentioned above, meetings often undergo changing collaboration modes. Changes of the collaboration mode may be induced by changing human behavior and do not need to be reflected in the model retained by the software. In other cases, however, it is quite helpful to reflect these changes in the software. If the software is aware of the current collaboration mode, it can adapt its behavior accordingly.

The mode of collaboration is modeled in groupware as the degree of coupling or “coupling mode” (Patterson *et al.*, 1990). Dewan defines *coupling* as the means by which interface components share interaction state across different users (Dewan and Choudhard, 1991). Dewan and Choudhary (1992) state that flexible coupling mechanisms are an important requirement. In (Haake and Wilson, 1992; Berlage and Genau, 1993) different modes of collaboration are identified: individual work, loosely coupled work, and tightly-coupled work. Tightly-coupled collaboration is critical during highly-interactive exchanges between people (Tatar *et al.*, 1991).

#### *Requirement C-2: Flexible Coupling and Modeled Collaboration Mode*

Collaborative software must provide support for implementing different coupling modes and allow flexible changes in the degree of coupling.

### 2.5.2 Single Display Groupware

Some devices such as interactive tables or walls offer another challenge for the software: a number of people can use and interact simultaneously with a single device. This is often called Single Display Groupware (SDG) (Stewart *et al.*, 1999).

One of the first SDG applications was MMM (Multi-Device, Multi-User, Multi-Editor) (Bier and Freeman, 1991), which allowed the use of up to three mice concurrently to edit rectangles and text. Tivoli (Pedersen *et al.*, 1993) supported up to three pens on the original version of the Xerox Liveboard (Bruce and Elrod, 1992). KidPad is a collaborative drawing tool designed for children (Druin *et al.*, 1997), (Stewart *et al.*, 1998). All these examples have in common that the software has to handle concurrent event sequences, such as drawing a stroke or dragging a window (Hourcade and Bederson, 1999).

While all these systems use mice or pens connected to a PC, the Pebbles project (Carnegie Mellon University, 2000) explored how PDAs can be used to provide concurrent input to a shared PC (Myers *et al.*, 1998b; Myers, 2001). This is already a step toward a ubiquitous computing environment, as multiple devices are used in collaboration (see also req. UH-2).

In addition to SDG, in ubiquitous computing environments multiple users at one display can collaborate with multiple users at other displays. This leads to an n-to-m relation among collaborating users and devices (Winograd, 2001a).

Apart from these technical issues, the user interface should be designed to allow interactions of multiple users without interference (Stewart *et al.*, 1999; Zanella and Greenberg, 2001).

#### *Requirement C-3: Multiple-User Devices*

Software running on a device capable of receiving input by multiple users must be able to receive events from several input-streams, to recognize input from different users, and to track several concurrent event sequences.

### 2.5.3 Collaboration in UbiComp Environments

The Pebbles project (mentioned above) presents an example of support for collaboration where several devices are used concurrently. However, the PDAs are used as input devices for a “master” PC only. This configuration can be extended to a scenario where all participating devices are equivalent, providing functionality at the same level of abstraction.

With the SharedNotes system (Greenberg *et al.*, 1999), it is possible to create personal notes on a PDA. In a meeting, these notes can be published and discussed on a public display. Another example is Rekimoto's M-Pad (Rekimoto, 1998b). It enables several people to use a PDA to transfer data to an interactive display using the pick-and-drop interaction style (Rekimoto, 1997). These examples can be considered as the first step from single display groupware to the integration with ubiquitous computing.

Marsic (2001) describes a system that supports synchronous collaboration with heterogeneous devices. The same instance of information objects can be concurrently edited on a high-end workstation with a virtual reality display and a PDA, using different visualization for each device. Recently, Myers *et al.* (2002) described another system that enables to share information synchronously among public displays and handheld devices.

The examples mentioned above show that within the context of ubiquitous computing, groupware applications cannot assume that they run on similar hardware. Instead, the software must be designed in a way that enables collaboration using heterogeneous devices (Shafer *et al.*, 2001). This implies that standard methods of shared editing cannot be used. For example, WYSIWIS ("What-You-See-Is-What-I-See" (Stefik *et al.*, 1987a; Stefik *et al.*, 1987b)) would require that all collaborating users have coupled workspaces of exactly the same size in pixels. This is not possible in a heterogeneous environment if the devices cover a display size in the range of very small to very large—or even do not have a display at all (Olsen *et al.*, 2000b; Olsen, 1998).

Therefore, the software must allow tightly-coupled components to use different view properties; it must also ensure that the users get a representation that fits to the current working mode. A user in a CommChair working on a shared workspace together with a user at a DynaWall (see section 2.2) will need both an overview representation of the whole workspace content and a zoomed view to work with. If the CommChair is located directly in front of the DynaWall so that the user has an overview, the overview representation displayed at the CommChair can be shrunk or omitted, as it may be needed for navigation only.

### *Requirement UC-1: Collaboration with Heterogeneous Devices*

Ubiquitous computing software must be able to handle collaboration if heterogeneous devices are involved. It must be possible to view and modify the same information with different devices and forms of interaction.

## 2.6. SW: Software Engineering Requirements

This section discusses requirements that arise from general software engineering. The requirements listed above are related to a specific research area, whereas the requirements presented in this section apply to every software application. However, they are of major importance so that they are explicitly mentioned. Other important non-functional requirements such as portability and scalability are not mentioned here, as they are implicitly included in requirement U-1. Supporting the heterogeneous devices found in ubiquitous computing implies that (a) the software must run on different hardware platforms, and (b) it has to cope with multiple devices, which leads to scalability issues.

Software functionality can be divided into levels of abstraction. Since this is important for improving the structure of the entire software system (Dijkstra, 1968), it is also significant for the presentation of a system's functionality to the user, in order to help in creating a uniform user interface. Therefore, the requirements presented in this section can be viewed from two perspectives: the user and the system perspective.

### 2.6.1 Functionality Common for Whole Application Domains

In every application domain, some functionality is common to a wide range of application scenarios. This functionality should be implemented in a reusable way. Examples are generic

document elements, such as workspaces, text, scribbles, and common tools like document browsers.

One possibility to realize reusability is to design software components in a composable way. Composability has been identified as a major requirement for state-of-the-art groupware environments (ter Hofte, 1998).

While reusability in general is still a big challenge in software engineering (Garlan *et al.*, 1995; Krueger, 1992), it is especially difficult in the context of ubiquitous computing. The device heterogeneity of ubiquitous computing environments implies coping with various platforms, resource constraints, and interaction styles, which makes reuse even harder (Garlan, 2000).

*Requirement S-1: Generic Functionality—Reusability*<sup>12</sup>

The software architecture has to be designed in such a way that it eases the reuse of generic and application-domain-specific functionality. Thereby, the heterogeneity of ubiquitous computing environments has to be taken into account.

### 2.6.2 Special Functionality for a Single Task

Aside from support for generic meeting tasks special-purpose functionality can improve the efficiency of a number of tasks (Nunamaker *et al.*, 1995, p. 167). An interview study that Streitz *et al.* (1998) carried out found that creative teams have several recurring tasks. Important examples for typical group tasks that are to be supported are creative sessions, presentations, meeting moderation, and project or task management. Consequently, the software should offer dedicated help for a selected set of such tasks, which should be extensible to meet future needs (ter Hofte, 1998; Moran *et al.*, 1998b).

One possibility to be able to provide tailored support is a module concept that allows extending the generic functionality. Of course, this has to be possible without the need to change existing code and without interference with other modules.

Depending on the devices used, different tasks need different tools that exploit the properties of the available devices. Therefore, extensibility in the context of ubiquitous computing can be seen from two directions. (1) New functionality must be developed in a way that is tailored to available devices. Oppositely, (2) newly developed devices might enable improved ways of interacting with existing functionality.

*Requirement S-2: Tailorable Functionality—Extensibility*

The software architecture has to ensure that the software system is extensible. To support ubiquitous computing, extensibility must reflect both functionality and interaction devices.

## 2.7. Summary of Requirements

Table 2-1 summarizes the requirements for synchronous collaboration in ubiquitous computing environments that are relevant in the context of this thesis. Looking at the number of requirements of each area it can be noted that the focus is more on ubiquitous computing (eight requirements) than it is on HCI (five requirements) and CSCW (four requirements). One reason is that ubiquitous computing is a rather new research area compared to HCI and CSCW, making it necessary to provide requirements that are more detailed. This emphasizes the developments that are necessary compared to the state of the art. The combination of HCI and CSCW is not considered, as this combination is of such a fundamental importance for nearly all CSCW systems that it has been well elaborated in the field of CSCW.

<sup>12</sup> Requirements related to software engineering are denoted with a capital “S”.

## 2. Requirements of Ubiquitous Computing Applications

Research Area	Req.	Requirement Name
HCI	H-1	Different Forms of Interaction
	H-2	Different User Interface Concepts
UbiComp	U-1	Multiple and Heterogeneous Devices
	U-2	Multiple-Computer Devices
	U-3	Context and Environmental Awareness
	U-4	Dynamic Configuration
UbiComp & HCI	UH-1	Adapted Presentation
	UH-2	Multiple-Device User Interface and Interaction
	UH-3	Physical Interaction
CSCW	C-1	Multi-Device Collaboration
	C-2	Flexible Coupling and Modeled Collaboration Mode
	C-3	Multiple-User Devices
UbiComp & CSCW	UC-1	Collaboration with Heterogeneous Devices
SW	S-1	Generic Functionality—Reusability
	S-2	Tailorable Functionality—Extensibility

Table 2-1. Summary of requirements for the software infrastructure of ubiquitous computing environments

The requirements identified here apply to synchronous ubiquitous computing applications in general. The BEACH conceptual model proposed in this thesis (see chapter 4) must ensure that applications meet the requirements if they are constructed according to the model. However, some requirements are at a rather low level of abstraction. They cannot be completely captured by the conceptual model. Instead, the BEACH architecture (see chapter 5), which applies the conceptual model in the context of roomware components, is an example of how they can be fulfilled. This applies to requirements U-4, UH-1, C-2, and C-3.

In order to provide support for roomware components, their properties had to be analyzed. Based on these properties, architectural decisions can be made for the BEACH architecture, where the requirements and the BEACH conceptual model allow several possibilities:

- *visual* and *pen-based* interaction (req. H-1, H-2)
- standard *computation capabilities* in terms of memory resources and processing speed (req. U-1, UC-1)
- a *permanent network connection* among each other (req. C-1)
- a *slow wireless network* connection for mobile components (req. C-1)
- the roomware components are operated inside *dedicated meeting rooms* (req. U-1)
- the application scenarios cover *dynamic collaboration, informal meetings, and creative or design tasks* (req. H-2, U-4)

↓ Next chapter

The requirements are referred to in the next chapter with respect to the related work. They also form the basis by which the proposed conceptual model (chapter 4), software architecture (chapter 5), and framework (chapters 6 and 7) are evaluated.

### 3. Related Work

---

This chapter studies the state of the art that meets some of the requirements identified in the previous chapter. The relevant work done in the areas of HCI, UbiComp, CSCW, and software technology is analyzed. The conducted analysis reveals that no model or system covers all addressed requirements. They all have their strengths in their original domain only, failing to meet requirements that originate from other research areas. Especially, it becomes apparent that CSCW models and frameworks fail to cope with the requirements of ubiquitous computing, while UbiComp systems offer no adequate support for synchronous collaboration. However, software technology provides several approaches to increase the modifiability, extensibility and reusability of software systems. The chapter closes with a summary of important concepts constituting the fundamentals for the remainder of the thesis. The focus of the thesis is refined with respect to the presented state of the art. It is concluded that there is currently no appropriate support for synchronous collaboration in ubiquitous computing environments.

---

The requirements defined in the previous chapter are now used to evaluate other models and systems. It is discussed to what degree the requirements are fulfilled by existing systems and which ideas influenced the design of the solution proposed in this thesis.

As ubiquitous computing environments combine research outcomes from different areas (see fig. 1-2), related work from all areas has to be considered in order to be combined and extended to form a consistent solution. First, results from the domain of software architecture and object-oriented frameworks are presented. These are considered to help understand and design the software architecture and framework. The next sections present related work from human-computer interaction (section 3.2), ubiquitous computing (section 3.3), and computer-supported cooperative work (section 3.4). As this thesis contributes at four levels—the conceptual, the architectural, the framework, and the application level—related work is analyzed according to these levels. The last section summarizes important concepts and architectural designs that influenced this work.

---

↓ Section outline

#### 3.1. Object-Oriented Frameworks and Architectures

Software reuse has been proposed as an approach to cope with the increasing complexity of software. However, software reuse is still hard (Nowack, 1999; Krueger, 1992; Kiczales, 1994; Garlan *et al.*, 1995). This section presents software techniques that have proven to be successful in improving reusability and extensibility of software systems (req. S-1, S-2).

- Software *architectures* define the high-level structure for software systems. By analyzing successful software architectures in a given problem domain, structures can be identified that are common for the whole problem domain.
- Software *frameworks* allow the reuse of implemented software architectures, offering specific support for extensibility. In contrast to *class libraries* they focus on design reuse, not code reuse only.

All reuse techniques have in common that they rely on some forms of abstraction, generalization, and specialization (Krueger, 1992).

##### 3.1.1 Software Architecture

At the end of the 1960's, when software systems began to become more complex, a debate started on how to structure a software system (Dijkstra, 1968; Parnas, 1972). Today, software architecture is established as a discipline of software engineering of its own (Perry and Wolf, 1992; Garlan and Shaw, 1993). The term "software architecture", however, is not used consistently in literature and practice. Commonly, software architecture is defined as "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them" (Bass *et al.*, 1999). While this definition focuses on the architecture of a *particular system*, the term "architecture" is also used to refer to an architectural style, such as "client-server architecture" (Garlan, 2001).

A software architecture is often described by a set of components, connectors, and additional constraints or properties (Gregory D. Abowd *et al.*, 1993; Garlan, 2001). An *architectural style* suggests a vocabulary of component and connector types, together with a topology of how they are combined (Bass *et al.*, 1999; Perry and Wolf, 1992; Phillips, 1999; Gregory D. Abowd *et al.*, 1993).

In contrast to an architectural style, a conceptual or *reference model* specifies the complete structure of some class of systems at a relatively large granularity (Phillips, 1999; ter Hofte, 1998).<sup>13</sup> It shows the conceptual structure with its fundamental functional elements. It should be possible to map all systems of this class to the structure defined by the reference model. Without such an architectural framework it is difficult to structure applications (Calvary *et al.*, 1997).

Within a software development process, software architecture plays a key role in laying a foundation to meet the requirements (Castro and Kramer, 2001; Garlan, 2001; Foegen and Battenfeld, 2001). In particular, software architecture has its strength in meeting non-functional requirements (Kazman and Bass, 1995).

To describe an architecture, it has proven helpful to use different *views* (Perry and Wolf, 1992; Kruchten, 1995). Typically, views explaining the static structure, the dynamic behavior, and distribution are distinguished (Garlan, 2001). It is a common approach in software engineering to develop different models of the software system for each view of the architecture (Jacobson *et al.*, 1992; Bass *et al.*, 1999).

---

<sup>13</sup> Bass *et al.* (1999) further distinguish between a *reference model*, seen as the division of functionality, and a *reference architecture*, which maps a reference model onto software components.

In general, a *model* is an abstraction of a *system* (ter Hofte, 1998; Jacobsen, 2000). As abstraction implies simplification, creating different models of a system helps cover important aspects—while reducing the overall complexity for each aspect (Nowack, 1999).

The following describes two techniques for structuring software architectures that are relevant to the solution contributed by this thesis: *level of abstraction* and *separation of concerns*.

↓ Section preview

#### Levels of Abstraction

One of the first techniques that was used to structure software systems is using a layered structure with distinct levels of abstraction (Dijkstra, 1968; Buschmann *et al.*, 1996; ter Hofte, 1998). Each layer defines abstractions that can be used by higher layers to implement the functionality. This way, the *semantic gap* between two adjacent levels is much lower compared to building the overall functionality from scratch (Demeyer, 1996). Introducing levels of abstraction into a software system is seen as its *vertical* structure. There is empirical evidence that using a layered approach for design and implementation can reduce the development costs (Zweben *et al.*, 1995).

In the context of framework development, it has been recommended to define three layers as part of the functional view on the architecture (Succi *et al.*, 1999): The *environment layer* encapsulates low-level middleware- and platform-specific features. The *domain-specific layer* provides services common to a specific domain, while the *application-specific layer* contains services that are only used by particular applications.

#### Separation of Concerns

Separation of concerns is another principle that structures a software system. Abstractions are defined at the *same level of abstraction*, in order to simplify a problem (Jacobsen, 2000; Alencar *et al.*, 1999; Kazman and Bass, 1995). This structure is also called *horizontal decomposition*.

A software component can be regarded as *closed* if it has a well-defined and stable interface that hides the implementation details (Meyer, 1988). By hiding internal information and defining an externally visible interface, the software components implementing the abstractions can be loosely coupled, easing modifiability, portability, and reusability (Parnas, 1972; Kazman and Bass, 1995; ter Hofte, 1998).

A module is said to be *open* if it is still available for extension. This means that functionality can be added or functionality can be refined. Bertrand Meyer stresses that it is important for software components to be both open and closed in the sense defined here (Meyer, 1988).

Recently, a discussion has started that concerns should be separated along multiple dimensions (Tarr *et al.*, 1999; Herrmann and Mezini, 2000). Using techniques such as aspect-oriented programming (Kiczales *et al.*, 1997a), it is possible to combine orthogonal concerns in a modular way.

#### 3.1.2 Software Frameworks

There are several techniques for how software can be reused. A very simple form, aiming at code reuse only, is the class library concept as known from object-oriented programming (Hong and Landay, 2001). In contrast, a framework can be seen as a reusable, “semi-complete” application that can be specialized to produce custom applications (Johnson and Foote, 1988; Fayad and Schmidt, 1997; Fayad *et al.*, 1999). Apart from providing implementation classes that can be reused, a framework always predefines the complete architecture of the applications and enables applications to add functionality. Typically, frameworks and toolkits follow a specific architectural style (see above). Big parts of the flow of control are handled by the framework, causing an *inversion of control*, as the framework calls the application code. This is contrary to class or function libraries, where the library code is typically called by the application.

### 3. Related Work

As a result, developers are released from the burden of creating their own architecture. Instead, a proven design can be reused along with parts of the code (Fayad, 1999). The developer can concentrate on the parts specific to the particular application.

A software component is a “physical packaging of executable software with a well-defined and published interface” (Hopkins, 2000). Components can be combined to form larger systems. Therefore, a software component also aims to provide reusable software. While frameworks are designed to be extended by custom code, components focus on evolution of software systems. By structuring a system as a set of components with well-defined interfaces, each component can be modified with little effect on other components. However, in order to gain the desired independence, components have to rely on a common software framework providing the necessary infrastructure (Johnson, 1997). Mismatching assumptions about the available infrastructure are one of the major causes for incompatibility between components (Garlan *et al.*, 1995). Therefore, frameworks and components can be thought of as complementary technologies.

A software *infrastructure* is a software system<sup>14</sup> that acts as a foundation for other systems (Hong and Landay, 2001). It provides an architectural framework for other systems and offers dedicated services. One way to implement a software infrastructure is to provide a generic framework that implements the software architecture and hides the underlying technology from applications (see figure 1-3 on page 7). Challenges for an infrastructure are the definition of standard data formats and protocols and the design of basic services (Hong and Landay, 2001).

Apart from implementing an architecture, frameworks that offer a large number of reusable components, are sometimes referred to as *toolkits* (Hong and Landay, 2001). Typically, frameworks for graphical user interfaces have many predefined widgets that cover the needs of most applications. However, in context of user interfaces, the term “toolkit” is often used to describe a library of widgets (Myers, 2003).

Due to different scopes of frameworks, it is useful to introduce a classification schema (Fayad and Schmidt, 1997; Fayad, 1999). *System infrastructure frameworks* simplify the development of portable system infrastructures, e.g. operating systems. *Middleware integration frameworks* are used to integrate distributed applications. For example, they handle exchange of data in distributed environments. *Enterprise application frameworks*, finally, address a specific application domain. An example would be a manufacturing or banking framework.

#### Designing for Extensibility

Depending on the techniques used to create extensions and to add application-specific behavior, frameworks can be classified into white-box and black-box frameworks (Fayad, 1999; Demeyer, 1996; Gamma *et al.*, 1995; Johnson and Foote, 1988). *White-box frameworks* use the techniques of object-oriented languages to add extensions. Typically, sub-classes can be derived from dedicated base-classes, which provide a set of methods to be overridden and refined. This requires a detailed understanding of the framework’s architecture. *Black-box frameworks* define interfaces for components that can be integrated. Instead of using inheritance as main extension technique, they rely on composition and delegation, which is often easier to use for application developers.

Those parts of a framework that are designed to be extended are often called *hot spots* (Schmid, 1997; Schmid, 1999; Pree, 1999) or *hooks* (Froehlich *et al.*, 1997; Froehlich *et al.*, 1999). To be useful for many applications within the same domain, a framework must support the parts common to the applications within a domain (also called *frozen spots*), while providing hooks for the variable aspects. The variable aspects can be detected, e.g., by systematic

---

<sup>14</sup> A software system does not have to be a single software application only.



generalization (Schmid, 1997) or by identifying the axes of variability (Demeyer *et al.*, 1997). In this respect, the framework will offer a software architecture that can be easily adapted to target requirements.

#### Object-Oriented Techniques for Extensions

Apart from the “classical” object-oriented techniques for extensions (Johnson and Foote, 1988; Krueger, 1992), such as inheritance with polymorphism and dynamic binding (in the case of white-box frameworks) or composition and delegation (in the case of black-box frameworks), new techniques have been proposed.

*Open implementation* is a technique to provide links for tailorability of software components (Kiczales, 1994; Kiczales *et al.*, 1997b; Demeyer, 1996; Buschmann *et al.*, 1996). This is done by adding a meta-level interface (or *meta-object protocol*) in addition to the software component’s base interface (Kiczales *et al.*, 1991; Bouraqadi-Saâdani *et al.*, 1998). The meta-object protocol provides the ability to do self-analysis and self-adaptability, called *reflection*.

A reflective program is one that reasons about itself (Foote and Johnson, 1989). Typically, reflection allows inquiring for the type or class of objects, to check the existence of attributes and methods, or to send dynamically computed messages. Using meta-classes, the behavior of classes and their instances can be controlled (Klas *et al.*, 1989). Self-adaptability is an important property for ubiquitous computing applications, in order to react appropriately on environment changes (req. U-3). Recently, reflection has been used to increase adaptability of middleware (Kon *et al.*, 2002).

*An example where a meta-level is used to be adaptable for heterogeneous and mobile devices is the object-oriented Apertos operating system (Yokote, 1992; Yokote et al., 1994). Abstractions are separated from possible implementations to be able to map the abstractions at runtime to the currently appropriate implementation. This way, Apertos supports different strategies for persistence, object migration, or security.*

---

Example 3-1:  
Apertos

One form of open implementation is the possibility to extend classes defined in other modules. The extension mechanism is quite similar to inheritance. The only difference is that no new class is defined for the additional behavior. Instead, new features are added to the base class. Of course, the new features can only be used by modules that use the module defining the extension. This mechanism is one possibility of hyperslicing for n-dim separation of concerns (Tarr *et al.*, 1999), with the restriction that no methods can be combined.

Several programming languages support an extension mechanism. In functional programming languages that provide generic methods like CLOS (Keene, 1989; Paepcke, 1993) and Cecil (Chambers, 1992; Chambers, 1995) new implementations of generic methods can be added by different modules. For untyped interpreted languages, an extension mechanism can be added easily. So, Perl’s implementation of classes as packages (Wall *et al.*, 2002) allows adding methods in any module (file), and attributes can be added dynamically as well. Smalltalk (Goldberg and Robson, 1989) has the capability to add methods to classes by other packages. In typed programming languages, it is more difficult, as type safety has to be ensured. MultiJava, an extension of Java, enables class extensions (called “open classes”) by providing multiple dispatch (Clifton *et al.*, 2000).

## 3.2. Software Models for Human-Computer Interaction

This section discusses related work coming from the research area of human-computer interaction.

### 3.2.1 Conceptual Models and Architectural Styles from HCI

Conceptual models and reference models specify the complete structure of a class of systems at a relatively coarse level of granularity. They show the conceptual structure with its fundamental functional elements. For all systems within this class it should be straightforward to map

### 3. Related Work

them onto this structure. For interactive systems, several popular models have been developed.

#### Traditional Application Models

In the 80's, researchers started thinking about the structure of interactive applications. The first reference model for interactive applications was the *Seeheim model* (Pfaff, 1985; Phillips, 1999; Kazman and Bass, 1995), which introduced the separation of layers for the application (i.e. the functional core), dialog, and presentation. It was refined into the *Arch* model in the early 90's (Bass *et al.*, 1992; Phillips, 1999; Kazman and Bass, 1995). Arch introduced an abstract interface for the functional core, which is called "functional core adapter" (fig. 3-1). The logical interaction layer was added as a place for widget libraries and user interface tool-kits such as Motif or MFC.

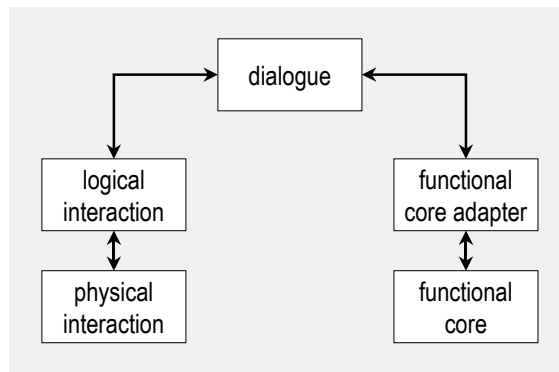


Figure 3-1. Arch reference model (Kazman and Bass, 1995)

A different approach, which was originally developed for the Smalltalk user interface framework, is the *model-view-controller* (MVC) paradigm (Krasner and Pope, 1988a; Buschmann *et al.*, 1996). It separates the model—which can be seen as a combination of Seeheim's application and dialog component—from view and controller, which divide the presentation in input and output functionality (see fig. 3-2). This way, an application can be divided into small components that are easier to develop and maintain. In addition, the goal was to create reusable view and controller classes.

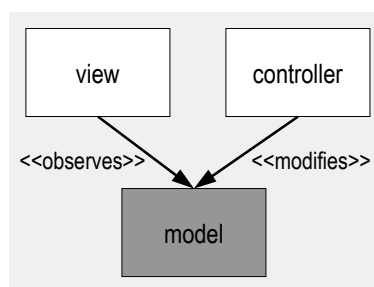


Figure 3-2. Model-View-Controller

Some later systems used a simplified version of model-view-controller, as they experienced that the controller can be rarely reused independently from the view. Therefore, they combined controller and view, and separated model and view only (Linton *et al.*, 1989; Morris *et al.*, 1986). The *model-view-presenter* (MVP) framework integrates the functionality of the controller in the view, but introduces the presenter that governs how the model can be manipulated by the user interface (Potel, 2000; Bower and McGlashan, 2000).

The separation of the functional core from the presentation provides the independence that is important whenever, e.g., the presentation has to be changed while the application needs no modification. This way, the interaction style and user interface concepts can be used without modification to the application itself (req. H-1, H-2).

The *PAC-AMODEUS* model, developed by Nigay and Coutaz (1991) tries to combine an MVC-like approach with the Arch architectural model. It was developed from the experience that the different layers of the Arch model often have a heterogeneous structure. If, e.g., a user interface toolkit is used to implement the presentation layer, the developer has no influence on the architectural model used. Instead, it is proposed to use a hierarchy of so-called “PAC agents” to structure the dialog control layer (fig. 3-3). A PAC agent is a software component with separated “facets” for presentation, abstraction, and control (Buschmann *et al.*, 1996). The control facet handles the communication between the presentation and abstraction facets, as well as the communication with other agents. PAC agents are organized in a hierarchy. The root represents the overall application, while the leaves correspond to single interaction objects like menus, icons, or windows.

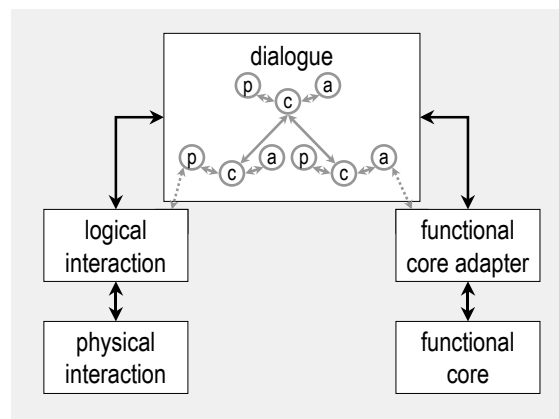


Figure 3-3. Reference architecture of PAC-AMODEUS (Nigay and Coutaz, 1991)

In this model, it is possible to support different interaction modalities by integrating several presentation and interaction components in the dialog component (Coutaz, 1997). However, this model is not designed to dynamically exchange modalities.

In order to develop a user interface design tool that allows expressing abstract conceptualizations of an interface, the *HUMANOID* model has been developed (Luo *et al.*, 1993; Szekely *et al.*, 1993). Most interestingly is the separation of application data, application model, presentation, and manipulation (Szekely *et al.*, 1992; Szekely, 1990). The *application data* describe the objects that are manipulated. The *application model* specifies the functionality the application provides for manipulating the objects. The *presentation* describes the (visual) appearance. The *manipulation* defines what user actions invoke what operations.

While the separation of data, presentation, and manipulation originates from MVC, the contribution of the *HUMANOID* model is a further *separation of data from the operations* used to manipulate it.

#### Models for Post-WIMP User Interfaces

The models presented so far were designed with traditional user interfaces in mind, supporting a mouse and keyboard interaction style. They offer appropriate separation of concerns for interaction and application functionality. However, their application is limited if newer interaction and user interface concepts—like pen or gesture interaction (Baudel and Beaudouin-Lafon, 1993), or zoomable user interfaces (Bederson and Hollan, 1994)—are used for an application.

### 3. Related Work

*Instrumental Interaction* is an approach to model both “traditional” WIMP (see page 20) and Post-WIMP user interfaces (Beaudouin-Lafon, 2000). It defines “domain objects” and “interaction instruments”. Domain objects are the “potential objects of interest for the user of a given application”, e.g., the objects defining a text document if the user edits text. Interaction instruments serve as a “mediator” between the user and domain objects. They transform the user’s action into commands that operate on the domain objects; they further provide feedback about the execution and results of the command. Therefore, an interaction instrument defines a (possibly dynamic) association of a physical part (i.e. the input device) and a logical part (e.g. a view on a screen).

The focus of this interaction model lies on interface *design* and not on interface *development*; the latter is the duty of architectural models.

Another concept to acknowledge the new requirement of more flexible user interface design is the framework for *plasticity* (Thevenin and Coutaz, 1999; Coutaz *et al.*, 2003). Plasticity (or adaptability) describes the property of a user interface how easy it can be adapted to changing requirements. It is argued that plasticity becomes an important issue if an application is to support a wide range of heterogeneous devices (req. U-1)—without the need to re-implement large portions of the user interface for every platform. Therefore, the model was inspired by model-based user interface generators. The conceptual model distinguishes two informal or semi-formal, and four formal models that provide the input to generate an appropriate “physical” user interface for a given platform. Within the context of this thesis, the four formal models only are relevant.

The “abstract user interface model” is the part specific for a given application. It describes the user interface of an application independent of concrete interaction and user interface elements. The “interactors model” describes all available interactors, i.e., components capable of processing and producing events. Typical examples of interactors are widgets or speech sentences. The “platform model” gives a description of the characteristics of the target platform. It includes available interaction devices, computational facilities (e.g. memory and processing power), or communicational facilities (e.g. availability and bandwidth of communication channels). Finally, the “environment model” specifies the context of use. It covers all objects, persons, and events that may impact the application’s functionality. This is closely related to the work on context aware applications as discussed below (see section 3.3).

The main contribution of this conceptual model relevant to this thesis is that the focus of models for interactive applications is widened to include also properties of the context the application is running in. Therefore, it is possible to abstract from the concrete platform and context—gaining independence from the current environment, which results in more flexibility in adapting to new contexts. This is important for designing user interfaces for ubiquitous computing environments.

Tangible User Interfaces: Model–Control–Representation (physical, digital)

Tangible user interfaces constitute a special case of Post-WIMP user interfaces. “Graspable” or “tangible” user interfaces facilitate the manipulation of digital information by interacting with physical objects (Fitzmaurice *et al.*, 1995; Ishii and Ullmer, 1997).

In the systems described so far, physical objects are completely ignored (as is in most models) or seen as adorning the context of the user interface only. Tangible user interfaces (TUIs) differ in that they treat physical objects as first-class entities of the user interface.

The MCR*pd* interaction model developed by Ullmer and Ishii (2000) extends the MVC model (described above) to fit the description of tangible user interfaces. While the “model” and “controller” parts are also present in a TUI, the “view” part is replaced by the *digital representation (rep-d)* of a physical object. In addition, the physical object itself is included in the interaction model as the *physical representation (rep-p)* of the (digital) model object. In contrast to the view (rep-d), which renders the current state of the model object, the physical representation

is only used as input object in many systems. Therefore, the controller is associated with the physical representation, trying to update the model whenever the physical object is manipulated. However, if the physical object is equipped with actuators, it is also possible to reflect the state of the model in its physical representation. In addition to the models defined by the plasticity framework, MCRpd allows to use physical objects not only as part of the environment, but also as first-class interaction objects (called “interactors” in plasticity).

### 3.2.2 User Interface Application Frameworks

This section gives prominent examples of user-interface application frameworks that introduce significant concepts or mechanisms. It does not have the intent to give an extensive overview of all application frameworks coping with the construction of user interfaces.

#### User Interface Toolkits

The following presents systems that help in constructing user interfaces.

The *Garnet* system (Myers, 1990; Myers *et al.*, 1992) is a user interface development environment written in Lisp. It is based on a *prototype-instance object system* (Lieberman, 1986). It uses a *retained object model* for all graphics. This means that for anything that is displayed, a corresponding object has to be created. This is related to the views of the MVC paradigm, but at a finer level of granularity.

While the original MVC uses notifications sent by the model to the view whenever the model is changed to implement the observer pattern (Gamma *et al.*, 1995), Garnet uses a universal one-way *constraint system* to couple values among objects. Any slot of an object can contain code to compute its value instead of containing the value itself. The value is automatically re-computed whenever there are changes in the slots that were read to compute the current value.

The input model is also an extension of MVC. As most controllers in MVC are closely related to a single view, Garnet introduced the concept of “interactors” to separate the input behavior from presentation. Interactors can be considered as a black box (Fayad and Schmidt, 1997) implementation of the most common input behaviors. This way, interactors can be easily reused in many situations that do not require an uncommon interaction style.

The *Amulet* framework is the successor of Garnet (Myers *et al.*, 1997; Myers *et al.*, 1998a). It is written in C++ and improves some features of Garnet that turned out to be inconvenient in practice. First, Amulet adds a strictly layered design (see figure 3-4). “Gem”, the graphics and input layer defines a portable framework for handling of low-level graphics and interaction. The “ORE” object system implements a prototype-instance object system in C++. As with Garnet’s object system, it can be used for all kinds of objects, not only for those related to the user interface. The constraint system was extended to allow multiple constraint solvers to co-exist. A set of six “interactors” is provided that is sufficient for all behaviors found in today’s user interfaces. It also includes a gesture interactor that supports pen gestures. To support new forms of interaction, such as speech input, new interactors can be added. On the top of the architecture, two layers defining command objects and widgets were added. An interface builder called “Gilt” uses all of the lower layers. Later, the Amulet framework was also extended to support simultaneous input from several concurrent users (see section 3.4.3).

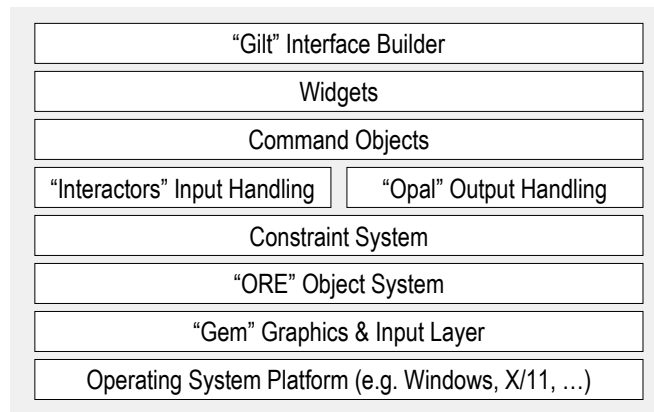


Figure 3-4. Layers defined in *Amulet* (Myers *et al.*, 1997)

Apart from Garnet and Amulet, several other systems have investigated the use of constraints for the creation of user interfaces (Bharat and Hudson, 1995; Hudson and Smith, 1996; Freeman-Benson, 1990; Szekely and Myers, 1988; Epstein and LaLonde, 1988).

The *Jazz* toolkit aims to help with the construction of zoomable user interfaces (Bederson *et al.*, 2000). It was built based on the experiences with Pad++ (Bederson *et al.*, 1996; Bederson and Meyer, 1998). Similar to Garnet and Amulet, a retained graphics model (here, called "scene graph") is used.

However, most interesting for the design of frameworks is the "minilithic" design philosophy on which Jazz is based. In contrast to a monolithic approach where classes contain many methods implementing functionality of different aspects, in Jazz functionality is added rather by composing, and not through inheritance.

Therefore, for example, many decorator nodes (also called wrapper objects) are used to add behavior by composition (Beck and Johnson, 1994; Gamma *et al.*, 1995; Brant *et al.*, 1998)—in contrast to adding this functionality to a subclass. Support for handling the management of several decorators for one object is not provided by the object itself. Instead, editor objects are introduced that manipulate decorators, e.g., return a reference to a specific kind of decorator. If a requested decorator is not yet available, a new instance is transparently created and inserted in the scene graph.

#### Pen and Gesture Support

Pen and gesture input is an interesting example of how a different interaction style is supported. Besides, pen and gesture input itself is also important for the work presented here (see section 7.5.2).

SATIN is a toolkit to develop informal pen-based applications (Hong and Landay, 2000). It was developed to create a generalized software architecture for informal pen-based applications, focusing on how to handle sketching and gesturing in a reusable manner.

SATIN defines a number of general concepts in order to process pen input. A "stroke" is the elementary object created by the user's pen. The "stroke assembler", aggregates user input into strokes and dispatches them as events. A "recognizer" classifies ambiguous input and returns an n-best list ordered by probability. Finally, the "interpreter" takes actions based on user-generated strokes. The interpreter also decides whether or not and what recognizer to use.

The stroke assembler can be seen as a special interactor developed for handling low-level pen-input. This is the same approach as taken in Garnet's extension for supporting gesture recognition (Landay and Myers, 1993). The approach of separating the recognition algorithm from the interpreter results in more flexibility in choosing and replacing the algorithms used (Henry *et al.*, 1990). The interpreter can be seen as a MVC controller that handles pen events instead of mouse or keyboard events.

3.2.3 Discussion of HCI Models

Table 3-1 gives an overview of what requirements are met by the presented models and frameworks. The main contribution of the human-computer interaction model lies in the separation of concerns related to the user interface. This makes it easier to use a different user interface or interaction style without having to change the rest of the application (req. H-1, H-2, UH-1, S-1, S-2).

The newer conceptual models (“instrumental interaction” and “plasticity”, page 40) have been developed because not all devices can be treated homogeneously. This is important for the support of ubiquitous computing environments (req. U-1). Plasticity is the most advanced model, as it acknowledges the need to include the environment within the model (req. U-3) and that ubiquitous computing environments are changing dynamically (req. U-4). MCRpd (page 40) is the only conceptual model offering specific support for physical user interfaces (req. UH-3).

However, none of the models and frameworks presented here offers support for collaboration (req. C-1, C-2, C-3, UC-1) or for anything that goes beyond the boundary of a single device (req. U-2, UH-2).

Requirement	H-1	H-2	U-1	U-2	U-3	U-4	UH-1	UH-2	UH-3	C-1	C-2	C-3	UC-1	S-1	S-2
<i>Conceptual Models</i>															
Seeheim / Arch	✓	✓												✓	
MVC	✓	✓												✓	✓
PAC-AMODEUS	✓	✓					✓							✓	✓
HUMANOID	✓	✓				✓								✓	✓
Instruments	✓	✓							(✓)					✓	
Placticity	✓	✓	✓		(✓)	✓	✓		(✓)					✓	
MCRpd	✓	✓			✓				✓					✓	✓
<i>Frameworks</i>															
Garnet / Amulet	✓	✓				✓								✓	✓
Jazz	✓	✓				✓	✓							✓	✓
SATIN	✓					✓	✓							✓	✓

Domain	Req.	Requirement Name	Domain	Req.	Requirement Name
HCI	H-1	Different Forms of Interaction	CSCW	C-1	Multi-Device Collaboration
	H-2	Different User Interface Concepts		C-2	Flexible Coupling and Modeled Collaboration Mode
UbiComp	U-1	Multiple and Heterogeneous Devices	UbiComp & CSCW	C-3	Multiple-User Devices
	U-2	Multiple-Computer Devices		UC-1	Collaboration with Heterogeneous Devices
	U-3	Context and Environmental Awareness			
	U-4	Dynamic Configuration			
UbiComp & HCI	UH-1	Adapted Presentation	SE	S-1	Generic Functionality—Reusability
	UH-2	Multiple-Device User Interface and Interaction		S-2	Tailorable Functionality—Extensibility
	UH-3	Physical Interaction			

✓ = requirement is fully supported  
 (✓) = requirement is partially supported

Table 3-1. Comparison of the HCI models against the requirements for a conceptual model for the software infrastructure for roomware environments. Requirements H-1 and H-2 are supported by most HCI models and frameworks. Requirements U-2, UH-2, C-1, C-2, C-3, and UC-1 are not explicitly addressed. This is indicated in the table by the background color of the columns.

#### 3.3. Software Models for Ubiquitous Computing

As the research area of ubiquitous computing is quite new compared to the other areas, no established conceptual models exist yet. Garlan (2000) identifies the development of architectures for ubiquitous computing as one of the major challenges for future software architectures. However, first models are currently being developed, and looking at the existing prototypes for software infrastructures for UbiComp environments, the underlying concepts can be investigated.

This section therefore presents conceptual models and software architectures of the state of the art of ubiquitous computing. It analyses to what degree they fulfill the requirements for the software infrastructure for roomware environments.

##### 3.3.1 Environmental Awareness

The systems presented in this section include context information as part of the application design. This leads to context-aware applications that can draw on a richer source of information for the reactions. Here, only three systems are presented. An extensive survey has been compiled by Chen and Kotz (2000). Recent developments are discussed in (Moran and Dourish, 2001).

##### EasyLiving: Building Intelligent Environments

The EasyLiving project at Microsoft Research (Microsoft Research, 2001) addresses the development of architecture and technologies for intelligent environments with heterogeneous devices (Shafer *et al.*, 2000; Shafer *et al.*, 1998; Brummit *et al.*, 2000). It aims at developing an architecture that aggregates diverse devices into a coherent user experience.

To achieve this goal, EasyLiving provides a middleware system, a geometric world model, support of sensing devices, and abstract service descriptions. The *middleware system* (“InConcert”) focuses on passing messages between mobile devices. It has no support for synchronous collaboration and concurrent access to shared objects, however. While the focus of the EasyLiving project is on the handling of many devices and their adaptability to context changes, the system also provides parts that define a conceptual model. The EasyLiving *Geometric Model* is a very advanced description of spatial relationships between devices and users. *Sensing devices* collect information about the state of the physical environment, to be able to keep the Geometric Model up to date. *Abstract descriptions of services* support the decomposition of device control, internal logic, and user interface. This functionality is quite similar to ICrafter (which is part of iROS, described below).

##### CoolTown: Integrating the Real with the Virtual World

The HP Labs are working on the CoolTown project (HP Labs, 2003), which aims at integrating the real with the virtual world by adding Web presence for “people, places, and things” (Kindberg *et al.*, 2000; Caswell and Debaty, 2000). A Web presence can be considered as an extended homepage that dynamically reflects the current status.

Conversely, physical spaces can be extended to integrate the virtual world by placing infrared beacons that provide URLs related to the physical space. URLs are exchanged to access services and to transfer content, e.g. a Web-present printer renders and prints the documents attached to the URLs it receives. A service called “PlaceManager” is responsible for providing views of the resources present in the place and related services.

##### ContextToolkit: Abstractions for Context Information

The ContextToolkit (Dey *et al.*, 2001b; Dey *et al.*, 2001a) was developed by Anind Dey at the Georgia Institute of Technology as part of his Ph.D. (Dey, 2000). He provides a definition of “context”, a conceptual framework for context-aware applications, and a toolkit that instantiates the conceptual framework to facilitate rapid development of applications.



The conceptual framework aims to ensure the separation of concerns principle by introducing abstractions for context information. “Context widgets” adapt the idea of GUI widgets to the context area by defining abstractions of specific context information, thus hiding sensor details. “Interpreters” and “aggregators” produce higher-level information by interpreting and combining the information provided by the context widgets. Dey defines “services” as the complement of “widgets”. While widgets provide input to an application, context services modify the state of the environment, e.g. by using actuators. Finally, “discoverers” maintain a registry of the services that are currently available within an environment.

### 3.3.2 Infrastructure for Ubiquitous Computing Environments

The projects presented in this section concentrate on technology-enriched rooms. These rooms are equipped with a range of devices that are integrated with the overall environment. In order to enable a fluid interaction in this kind of rooms, an appropriate infrastructure needs to be developed.

#### Aura: Architectural Framework for User Mobility

At Carnegie Mellon University, the Aura project is creating an architectural framework for user mobility in ubiquitous computing environments (Carnegie Mellon University, 2002; Sousa and Garlan, 2002; Garlan and Schmerl, 2001). The main components are a first-class representation of user tasks, called “personal aura”, and a representation of *tasks* as a collection of *abstract service descriptions*. The abstract service descriptions can be mapped to *concrete service suppliers* that are available in the environment. A “Task Manager” is responsible for allocating tasks to available resources, thus minimizing user distraction. The “Environment Manager” is aware of which service suppliers are currently available in the environment. Finally, “Context Observers” provide information on the physical context to the Environment Manager.

These components are placed at the three layers defined by Aura’s architecture (Cheng *et al.*, 2002). The “Runtime Layer” is the lowest layer, containing the Environment Manager and the runtime system. The next layer, called “Model Layer”, defines the externalized architectural model. The top layer, “Task Layer”, contains the Task Manager, which defines the task model.

#### Gaia: Operating System and Application Model for Active Spaces

The Gaia project (University of Illinois at Urbana-Champaign, 2002; Román *et al.*, 2002; Román, 2003) at the University of Illinois at Urbana-Champaign (UIUC) is developing a software infrastructure for UbiComp environments (which they call “active spaces”). They are developing an operating system for active spaces called “GaiaOS” and a conceptual model called “MPACC” for applications built using GaiaOS.

GaiaOS (Román *et al.*, 2001b) is a component-based *meta-operating system* or middleware operating system. The term “meta-operating system” is used to stress the fact that it is an operating system for whole ubiquitous computing environments, built on top of currently existing operating systems. GaiaOS consists of two main parts, the Unified Object Bus and the GaiaOS kernel. The Unified Object Bus defines a common interface to manipulate components in the active space to hide the heterogeneity of the hardware devices and software protocols.

The GaiaOS kernel contains a minimum of required services for an active space. This includes a *naming service* to access distributed objects; an *event manager* to distribute information; a *discovery service*, which is responsible for tracking software components, people, and physical objects; a *space repository* that stores information about arbitrary entities (e.g. devices, services, or users); a *security service*; and the *data object service*. The data object service (Hess *et al.*, 2001a) is a low-level infrastructure to deliver data objects to computers. To handle the heterogeneity of UbiComp environments, it is able to adapt content depending on the current context, e.g., the current location, or the used device.

### 3. Related Work

On top of the kernel services, other services can be implemented. GaiaOS has been realized using a customized CORBA implementation (Román *et al.*, 2001a).

As part of the Gaia project, an “application model” for ubiquitous applications has been developed (Román and Campbell, 2001). It was realized that models for traditional applications offer not enough support for the features needed in a ubiquitous computing environment. Therefore, the model–view–controller model (see section 3.2, p. 38) has been extended to the *model–presentation–adapter–controller–coordinator* (MPACC). The term “presentation” is used instead of “view” as a ubiquitous computing environment offers more modalities besides visual-based interaction. Similarly, controllers are not only used for input devices but for any physical and digital context that can affect the application. “Adapters” are introduced to transform a model to a form that is understood by a given presentation. This can help increase the reuse of both models and presentations.

Finally, the “coordinator” is a “meta-level component” that manages the application composition and applies adaptation aspects. It knows the adaptation policies and configuration rules that are used for combining the components.

#### iROS: Infrastructure for Interactive Rooms

The Interactive Workspaces project at Stanford University (Stanford University, 2000) has developed a couple of tools that facilitate working in a “technology-enriched” space. The Interactive Room (iRoom) was set up with a hardware technology very close to the one used in i-LAND at Fraunhofer IPSI. Although the Interactive Workspaces project focuses on a meeting and task-oriented working situation as application context, they offer no specific support for synchronous collaboration. Instead, they aim to enable the use of both new and legacy applications in ubiquitous computing environments, and the interaction between the two. The focus is on the dynamic and continually evolving nature of interactive spaces.

The software infrastructure for the iRoom is *iROS*, the “interactive room operating system” (Johanson *et al.*, 2002a; Ponnekanti *et al.*, 2003). It defines a set of low-level services to facilitate the development of applications for ubiquitous computing environments. The major sub-systems of iROS are the Event Heap, the Data Heap, and ICrafter.

The *Event Heap* provides the core infrastructure of iROS (Fox *et al.*, 2000; Johanson and Fox, 2002; Johanson and Fox, 2004), being responsible for coordination of applications and services. It uses a blackboard metaphor (Winograd, 2001b), by exchanging tuples that are placed on and obtained from the Event Heap. The tuple-space model was introduced in Linda (Ahuja *et al.*, 1986; Carriero *et al.*, 1994). Any service can post tuples to the central instance of the Event Heap, and can subscribe to a tuple-pattern. Linda’s model of the tuple-space (which is also used by TSpaces (Wyckoff *et al.*, 1998) or JavaSpaces (Sun Microsystems, 2002; Sun Microsystems, 2000)) has been extended in a number of ways. Most important, Event Heap’s tuples are self-describing to make it easier to adapt applications to work together. A publish-subscribe mode is introduced as an alternative to the normal polling mechanism. Also, every tuple can be received by multiple services. This allows a broadcast-like communication. The benefit of a tuple-based communication is that communication can be anonymous, i.e., sender and receiver do not need to know each other. This eases the extensibility of the software infrastructure.

The *Data Heap* is a per-room attribute based file store that can be used to store persistent information. The Context Memory is a part of the Data Heap, holding meta-information about the files stored. To increase flexibility, the Data Heap provides facilities for data transformation.

*ICrafter* is a service framework that enables the software infrastructure to select, generate, or adapt user interfaces for the services available in a ubiquitous computing environment (Ponnekanti *et al.*, 2001). The Service Discovery keeps track of running services. User interfaces are generated from abstract service descriptions that are provided by every service. In-

formation about the used device is used to decide what kind of interface to select (if a tailored interface exists for this class of devices) or to generate (from the service description). This way, a wide range of different user interfaces can be supported.

#### one.world: System Architecture for Pervasive Computing Environments

The one.world project at the University of Washington (University of Washington, 2003) is creating a system architecture for pervasive computing (Grimm *et al.*, 2002; Grimm *et al.*, 2001). They follow three main principles: first, expose change, e.g. of the environment or of network connection, rather than hiding it, to enable applications to react upon changes; second, to compose dynamically, to be able to have an appropriate reaction (this relates to our requirement U-4); and third, to separate data and functionality. The third point enables data and functionality to evolve independently. This is discussed in section 4.3 below.

The one.world project defines three basic abstractions and a set of services. *Tuples* provide a common data format. *Components* define functionality. *Environments* group tuples and components. Components can use five core services that are provided by one.world. *Checkpointing* allows the capture and restoration of application state. *Migration* moves components and their data between environments, which can reside on different nodes. Components can communicate by *passing events*, including facilities for remote event passing. *Discovery* allows sending events to services with an unknown location. To manage asynchronous interactions, *logic*, i.e. computations that do not fail, is separated from *operations* that might fail.

#### Oxygen: Pervasive Human-centered Computing

The Oxygen project at the Massachusetts Institute of Technology (MIT) (Massachusetts Institute of Technology, 2003) embraces a couple of activities to create an infrastructure for human-centered pervasive computing. In contrast to the other projects mentioned, Oxygen is not only concerned with software technology, but following a universal approach, it covers device and network technologies as well. The focus is on multimodal interaction in “intelligent environments” (Coen, 1998).

The software infrastructure of Oxygen is investigated by two projects. *Metaglu*e is a robust architecture that provides the “computational glue” for software agents (Coen *et al.*, 1999). It offers support for resource management, configuration changes, agent migration, and agent persistency. *Hyperglue* (Shrobe *et al.*, 2002) increases the scalability of Metaglu. The resource management is extended to support high-level service discovery and routing of messages among instances of Metaglu.

GOALS is an architecture that focuses on software adaptability to changes and evolving system requirements (Ward *et al.*, 2002). In order to be able to adapt to changes, goals are formalized as a language construct. Software components that provide *techniques* for fulfilling goals are dynamically combined to construct the desired services. This way, the GOALS system integrates software services to accomplish user-defined goals.

#### 3.3.3 Discussion of UbiComp Models

Table 3-2 gives an overview of what requirements are met by the models, frameworks, and infrastructures presented. The systems presented in this section have strengths in handling of multiple heterogeneous devices (req. U-1), integration of the environment (req. U-3), and handling of dynamic changes (req. U-4). Abstractions are introduced in order to facilitate reuse and extensibility of components and services (req. S-1, S-2).

Dedicated support for adaptation of the presentation of information and user interfaces to devices (req. UH-1) is addressed by Aura, Gaia, and iROS (ICrafter). Aura separates tasks that contain abstract service descriptions from service suppliers. Gaia introduces an adapter that can be placed between model and presentation. ICrafter generates user interfaces from abstract service descriptions. Gaia and iROS only, offer explicit support for different forms of interaction, and different user interfaces (req. H-1, H-2). Both iROS and EasyLiving enable

### 3. Related Work

interaction with a user interface spread over multiple devices (req. UH-2). ContextToolkit, CoolTown, and EasyLiving are designed to integrate elements of the environment as part of the user interface (req. UH-3).

Interesting in one.world is the emphasis on the importance of separating data (tuples) and functionality (components) as known from HCI models (see section 3.2.1) in the context of ubiquitous computing. This separation enables the manipulation of the same data in different contexts, with different devices, and different functionality.

However, none of the models and frameworks presented here offers support for collaboration (req. C-1, C-2, C-3, and UC-1), or for multiple-computer devices (req. U-2). The iRoom, only, provides limited support for multiple-computer devices via PointRight (Johanson *et al.*, 2002b). PointRight uses a static model of the topology of the devices available within the iRoom in order to re-direct mouse and keyboard input to an arbitrary computer.

Requirement	H-1	H-2	U-1	U-2	U-3	U-4	UH-1	UH-2	UH-3	C-1	C-2	C-3	UC-1	S-1	S-2
<i>Environmental Awareness</i>															
EasyLiving			✓		✓	✓		✓	✓					✓	✓
CoolTown			✓		✓	✓		✓						✓	✓
ContextToolkit			✓		✓	✓		✓						✓	✓
<i>Infrastructure for Ubiquitous Computing</i>															
Aura			✓		✓	✓	✓		✓						✓
Gaia & MPACC	✓	✓	✓		✓	✓	✓							✓	✓
iROS	✓	✓	✓	(✓)	✓	✓	✓	✓						✓	✓
one.world			✓		✓	✓								✓	✓
Oxygen			✓		✓	✓								✓	✓

Domain	Req.	Requirement Name	Domain	Req.	Requirement Name
HCI	H-1	Different Forms of Interaction	CSCW	C-1	Multi-Device Collaboration
	H-2	Different User Interface Concepts		C-2	Flexible Coupling and Modeled Collaboration Mode
UbiComp	U-1	Multiple and Heterogeneous Devices		C-3	Multiple-User Devices
	U-2	Multiple-Computer Devices	UbiComp & CSCW	UC-1	Collaboration with Heterogeneous Devices
	U-3	Context and Environmental Awareness		SE	S-1
	U-4	Dynamic Configuration	S-2		Tailorable Functionality—Extensibility
UbiComp & HCI	UH-1	Adapted Presentation	✓ = requirement is fully supported		
	UH-2	Multiple-Device User Interface and Interaction	(✓) = requirement is partially supported		
	UH-3	Physical Interaction			

Table 3-2. Comparison of the UbiComp models and infrastructures against the requirements for a conceptual model for the software infrastructure for roomware environments. Requirements U-1, U-2, and U-4 are supported by most UbiComp systems. Requirements C-1, C-2, C-3, and UC-1 are not explicitly addressed. This is indicated in the table by the background color of the columns.

### 3.4. Software Models for Computer-Supported Cooperative Work

Researchers have developed a number of models, architectures, and toolkits for groupware systems. This section presents conceptual models, architectural styles, support for single display groupware, and groupware frameworks or toolkits for heterogeneous environments. A very extensive and excellent survey of groupware architectures was done by Phillips (1999); another article discusses programming abstractions and techniques (Greenberg and Roseman, 1999).

In the domain of CSCW, the concept of local and shared objects is important. An object is called *local* if it is local to a single computer, i.e. it can be accessed only by processes running

on this computer. Objects that can be accessed by multiple computers are called *shared* objects.

#### 3.4.1 Conceptual Models from CSCW

For groupware systems, several conceptual models have been developed. As mentioned above, a conceptual model specifies the complete structure of some class of systems at a relatively coarse granularity. This section presents two popular models for groupware systems, Patterson's taxonomy, and Dewan's "Generic architecture".

##### Patterson's Taxonomy

Patterson (1995) proposed the first reference model specific to groupware systems. It is based on the assumption that groupware applications need to maintain a shared state but use multiple displays to render the shared information. Therefore, Patterson divides the application into four levels. The "display" is implemented in the video hardware. "Views" represent a logical representation of the information in a format that can be displayed. The "model" is the shared information itself. It is kept persistent in a "file". Based on this separation, Patterson identified three classes of architectures for groupware: architectures with *shared state* that keep only a single instance of the model, those with *replicated state* that allow several instances of the model, which are synchronized, and *hybrid* architectures that are a mixture of the other two approaches. It is assumed that all levels above the first replicated one are replicated as well. Therefore, this model is sometimes referred to as the "zipper" model.

##### Dewan's Generic architecture

Patterson's architecture was generalized by Dewan (1999), keeping the structure of an (optional) centralized stem with replicated branches. In addition to Patterson's model, Dewan allowed an arbitrary number of layers to be defined, being aware that up to five layers are sufficient for most systems (fig. 3-5). Lower layers are related to the hardware and interaction devices; higher layers handle domain specific functionality. Components communicate by sending events. Events between layers are called "interaction events"; events between replicated instances at the same layer are called "collaboration events".

This model can serve as a good basis for coupling control, mapping of functionality to processes (including issues such as concurrency, replication, distribution), and mapping functionality to processors.

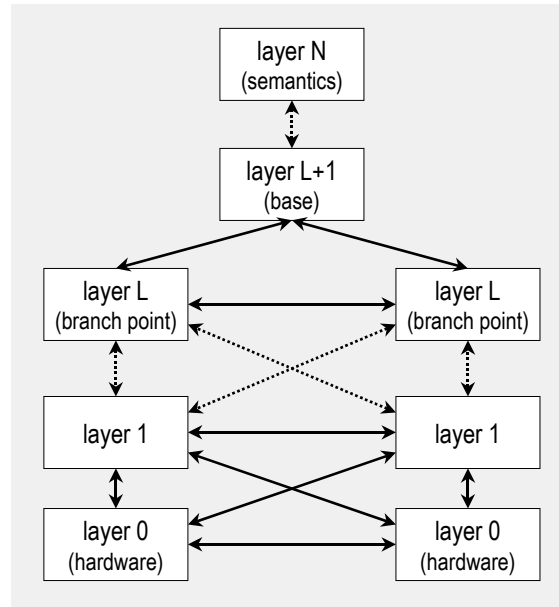


Figure 3-5. Dewan's generic architecture (Dewan, 1999)

#### 3.4.2 Architectural Styles for CSCW

An architectural style suggests a vocabulary of component and connector types, together with a topology of how they are combined (Phillips, 1999). Typically, frameworks and toolkits follow a specific architectural style.

This section presents architectural styles for groupware systems. Some of them—such as PAC\* or C-2—have been created by augmenting a style for single-user applications for multi-user settings.

##### PAC\*

The PAC\* style is an extension of the PAC-AMODEUS style (see section 3.2) for multi-user applications (Calvary *et al.*, 1997), using Dewan's generic architecture. In addition, no restrictions are made about which levels are replicated and which are centralized. Similar to PAC-AMODEUS, every Dialog Control is equipped with PAC agents. As in Dewan's model, events are passed vertically between adjacent local layers and horizontally between replicates. However, within the Dialog Control, all communication is handled by the control facets of every PAC agent.

As the PAC\* architectural style is at a relatively abstract level and integrates aspects of reference models, it can be positioned between an architectural style and a reference model.

##### Shared Model–View–Controller

The idea to couple presentation with the model using constraints, as used in Garnet or Amulet (see section 3.2 above), inspired researchers to develop multi-user architectures based on this approach.

The *abstraction-link-view* (ALV) style was developed at Bellcore as part of the architecture of the Rendezvous systems (Patterson *et al.*, 1990; Patterson, 1991; Hill *et al.*, 1994). It proposes to separate abstractions (the equivalent to MVC models) from their views, and to use constraints (called links) to couple the views to their abstractions. As part of the Rendezvous system, a constraint maintenance system was implemented (Hill, 1993), supporting declarative one-way constraints, similar to those of Garnet (Myers *et al.*, 1992). Abstractions and views can be structured hierarchically. Abstractions are always centralized, while the view hierarchy is replicated for every client. Therefore, ALV can be mapped to Dewan's generic architecture

with three layers and a single centralized layer. Unlike MVC, controllers (here implicitly part of the view) modify cached values in the local views; changes are then propagated by the links to the abstraction.

The main advantage of the ALV style is flexibility in how to separate domain models from their presentation in the multi-user case. This gives “dialog independence”, i.e., different presentations can be used for different devices and different users (Hill *et al.*, 1994). As links can adapt data while transferring between model and presentation, ALV offers a high potential for reuse of both models and presentations.

The *Clock* architectural style is directly based on MVC (Graham *et al.*, 1996; Urnes and Graham, 1999). It adopts ALV’s idea to link the view to the model using (one-way) constraints, but supports a separate controller sending requests and update events to its model. It also separates the data (here described as abstract data type) from the application model (called functional core as in Seeheim).

In *Clock*, groupware applications are defined in a component hierarchy, where every component can consist of a model, view, and controller. In correspondence with Dewan’s generic architecture, the components of the stem are shared up to a specific level. Components located further down the hierarchy are local to every client. Interestingly, requests that are not performed by a model are propagated up the component hierarchy. This introduces an elegant way of sending requests to the shared models at the root of the hierarchy.

By proposing a hierarchy of MVC components, the *Clock* architectural style differs from ALV in that it uses separate hierarchies for models (abstractions) and views. On the other hand, this relates a *Clock* component to a PAC agent, but using different strategies to pass messages.

The *COAST* groupware framework has introduced another variation of a shared MVC architectural style (Schuckmann *et al.*, 1996; Schümmer *et al.*, 2000). *COAST* is based on the concept of a shared-object space that allows distributed clients to access the shared objects, while transparently hiding details of communication. Therefore, this can be implemented using different distribution architectures. The *COAST* framework, however, handles the synchronization of replicated shared objects.

Model objects are always shared in the *COAST* architectural style. View objects, in contrast, are always local to a single client. It is argued that view objects must be local, as they interact with the local hardware and local interaction devices.

Similar to ALV or *Clock*, views are coupled to their model using one-way constraints. Controllers modify shared model objects. Changes are propagated to other replicates of the changed object. As distinct from other implementations, in *COAST*, views are never notified about changes by their controllers; updating of views is always triggered by their dependencies to the model. This way, the originator of a change is transparent. It makes no difference whether the change happened due to local input or whether it is propagated from another client. The system on today’s desktop PCs is fast enough that this approach causes no performance problems at run-time, which was the reason to have the controller directly modify its view in former systems. Details about performance optimizations in *COAST* are described in section 6.1.3.

Similar to ALV and the original MVC, the *COAST* style constructs separate hierarchies for views and models. This is because all views are always local while models are shared. Furthermore, the *COAST* style suggests further separating the model into the domain and application models (Schuckmann *et al.*, 1999). It is argued that by sharing both domain and application models, different collaboration modes (such as tightly- or loosely-coupled collaboration) are easily realized.

Compared to *Clock*, the *COAST* style offers more flexibility in decoupling the presentation from the model, which is an important property of ubiquitous computing environments.

### 3. Related Work

#### Chiron-2 (C2)

Similarly to the PAC\* style, the Chiron-2 (C2) architectural style (Taylor *et al.*, 1996) is an extension of a style developed for single-user applications (Taylor *et al.*, 1995). It does not explicitly distinguish between model and view components. Instead, it organizes components at an arbitrary number of layers. Components at “top” layers are considered to be models, while components at lower layers are related to views. All layers are separated by “connectors” that handle all communication between components. This implies that only components on adjacent layers can communicate directly.

The advantage of connectors is that the architecture can handle the distribution between multiple clients. As components never talk directly with each other, connectors can be introduced to route events to different clients, or to broadcast messages.

#### 3.4.3 Single Display Groupware

While traditional groupware architectures and frameworks aim at collaboration involving multiple computers, single display groupware concentrates on multiple users working with the same computer. Here—apart from issues of user interface design—the handling of multiple input devices and concurrent input are important issues.

While the groupware architectures and models presented so far only mention that single display groupware is a very simple special case (Dewan, 1999; Calvary *et al.*, 1997), some researchers have started investigating architectural issues for single display groupware. These are described next.

#### Multi-Device Multi-User Multi-Editor (MMM)

The Multi-Device, Multi-User, Multi-Editor (MMM) project at Xerox PARC developed one of the first single-display groupware applications. The architecture of MMM defines a hierarchy of *editors* that is used to structure the application (Bier and Freeman, 1991). Each editor has a space assigned on the display for which it is responsible. Child editors belong to a part of the area—like views in a MVC view hierarchy. An editor runs in its own process and is responsible for handling input events, manipulating its data, and updating the screen.

Input events are collected by a device process and put into the system queue. The notification process is responsible for the dispatching of events from the system queue to the appropriate editor’s queue. It also maps the device to its user by maintaining a “device ownership table”. When an event is dispatched from an editor to its child editors, the coordinates of the event are transformed to the editor’s local coordinate system. To avoid interference between users, events are dispatched to the children only if the editor is not busy, e.g., not being moved.

To execute commands, every editor stores the current modes and preferences on a per-user basis.

#### Pebbles

As part of the Pebbles project at Carnegie Mellon University (Carnegie Mellon University, 2000), support for single display groupware has been developed by extending the Amulet toolkit (see section 3.2.2 on page 41) to handle multiple input streams (Myers, 1999).<sup>15</sup> Pebbles is a framework that eases communication between a PC and one or multiple PDAs (Myers, 2001). It supports multiplexing and de-multiplexing of messages exchanged over a simple (infrared, serial cable, or network) connection between several applications on a PDA and their counterparts on the PC. In order to support single display groupware, a PDA can be used as an input device for a PC, emulating mouse and keyboard. The messages are received by an Amu-

---

<sup>15</sup> Amulet v5 that includes Pebbles support has not been released (email from B. Myers, Feb 19, 2003).



let Pebbles Handler running on the PC, which passes them as events, enriched with information about the originating device and the user's ID, to Amulet's Event Handler.

The widgets and interactors defined in Amulet are augmented to handle concurrent users. In contrast to traditional MVC, multiple controllers (interactors) are needed for the same view. Apart from that, different modes for interaction with widgets were identified. A widget marked "anyone-mixed-together" can simultaneously be used by different users. The default for most widgets is "one-at-a-time", which means that another user can interact with these widgets only, if no one else is currently using it. This corresponds to floor control strategies found in many groupware systems (Greenberg, 1991).

#### Multiple Input Devices (MID)

MID (Hourcade and Bederson, 1999) is a framework to enable single display groupware developed in Java at the University of Maryland to build the underlying infrastructure for Kid-Pad (Druin *et al.*, 1997), (Stewart *et al.*, 1998). MID stands for "Multiple Input Devices". It uses multiple mice connected via USB to a single PC. It provides capabilities to dispatch events generated by multiple mice, using a mechanism similar to Java's event dispatching mechanism. Components can add `MIDMouseListener`s in order to receive input from MID.

#### 3.4.4 CSCW Toolkits and Frameworks for Heterogeneous Environments

Some frameworks have been developed to investigate how to handle synchronous collaboration when using heterogeneous devices.

##### QuickStep and Pocket DreamTeam

QuickStep and Pocket DreamTeam are two frameworks to support the creation of synchronous groupware for mobile devices developed at the University of Hagen (Germany).

QuickStep (Fernuni Hagen, 2002a; Roth and Unger, 2000) has been designed for small, low-power handheld devices with low bandwidth. In addition, the operating systems of most handheld devices do not support threads. This imposes another challenge on a groupware framework. Therefore, QuickStep offers no true synchronous collaboration. Shared data can only be modified by the creator of a record, which avoids the need for conflict detection and resolution. Yet, this approach enables fast synchronization of replicates. To support privacy, private information that is stored in records can be masked before transmitting.

Pocket DreamTeam (Roth, 2002), is an extension of the DreamTeam groupware framework (Fernuni Hagen, 2002b; Roth, 2000) for mobile handheld devices. It was developed driven by the constraint to be interoperable with the original DreamTeam framework. To cope with the restrictions of wireless connections, an optimistic concurrency control strategy was added. Since the functional core of a groupware application is portable between the desktop and handheld platform, a new user interface has to be created for every platform.

##### Manifold

Manifold (Rutgers University, 2002; Krebs *et al.*, 2000; Marsic, 2001) is an extension of the DISCIPLE groupware framework (Wang *et al.*, 1999) to handle heterogeneous devices. It is based on a data-centric approach to sharing. This means that data is shared among all collaborators, but the visualization is rendered independently on every device. Therefore, different presentations can be used.

The Manifold architecture defines three tiers; one for the presentation (using MVC), domain logic, and collaboration functionality. This results in a separation of concerns. Depending on the platform, a different implementation of domain logic or presentation can be used. In order to exchange information between different platforms, a defined representation (based on XML) is used, which is transformed into a representation feasible for the particular platform.

### 3. Related Work

This approach is interesting because the different concerns for model and presentation are separated in a way that facilitates even different representation of models for different platforms.

#### XWeb

Another architecture that aims at enabling synchronous collaboration with heterogeneous devices is XWeb, which is being developed at the Brigham Young University (Olsen *et al.*, 2000a; Olsen *et al.*, 2000b).

It extends the HTTP protocol to be able to query and subscribe to XML documents. Clients can subscribe to XML documents in order to be synchronously notified upon changes to that document. To enable platform independent user interfaces, documents are coupled with a device-independent description of a user interface (called “view”) appropriate for this document. Views are encoded in XML and can be rendered for the particular device and the particular interaction modality. A tuple built from a document reference and a reference to a view is called a “task”. A “session”, finally, couples a task with a reference to the current focus within the task’s view. This model was used to build the “join and capture” interaction style for a collaboration environment with multi-modal interaction capabilities (Olsen *et al.*, 2001).

XWeb aims at defining an architecture supporting multi-modal interaction and collaboration, which is as simple as the Web. However, apart from defining data formats and protocols, no guidance is given on how to design the architecture of services and applications for a ubiquitous computing environment.

#### 3.4.5 Discussion of CSCW Models

Table 3-3 gives an overview what requirements are met by the presented models, architectural styles, and frameworks. The systems can be divided into two categories: multi-computer and single display groupware. Multi-computer groupware is designed to support collaboration with multiple devices and flexibly control the degree of coupling (req. C-1, C-2). Single display groupware, on the other hand, aim at multiple-user devices (req. C-3). Only Pebbles, Pocket DreamTeam, Manifold, and XWeb acknowledge the usage of different devices and platforms for collaboration (req. U-1, UC-1).

Most models also propose to separate the interaction and user interface from the domain-dependent parts of the applications (req. H-1, H-2). Abstractions are introduced in order to facilitate re-use and extensibility of components and services (req. S-1, S-2).

Altogether, the focus of CSCW models and architectures is not on the adaptation of presentations, user interface crossing the boundaries of a single device, integration of context information, or physical interaction (req. U-2, U-3, UH-1, UH-2, and UH-3).

### 3.5. Conclusions: What to Use, What is out of Focus & What is missing

Requirement	H-1	H-2	U-1	U-2	U-3	U-4	UH-1	UH-2	UH-3	C-1	C-2	C-3	UC-1	S-1	S-2
<i>Conceptual Models</i>															
Patterson's Taxonomy	✓									✓	✓			✓	✓
Generic architecture	✓									✓	✓			✓	✓
<i>Architectural Styles</i>															
PAC*	✓	✓					✓			✓	✓			✓	✓
Shared MVC	✓						✓			✓	✓		✓	✓	✓
Chiron-2	✓	✓	✓			✓				✓	✓		✓	✓	✓
<i>Single Display Groupware</i>															
MMM												✓			
Pebbles			✓	✓		(✓)		✓		✓		✓	✓	✓	✓
MID												✓		✓	
<i>Heterogeneous Environments</i>															
Pocket Dream Team	✓	✓	✓			✓				✓	✓			✓	✓
Manifold	✓	✓	✓			✓	✓			✓	✓			✓	✓
XWeb	✓	✓	✓			✓	✓	(✓)		✓	✓			✓	✓

Domain	Req.	Requirement Name	Domain	Req.	Requirement Name
HCI	H-1	Different Forms of Interaction	CSCW	C-1	Multi-Device Collaboration
	H-2	Different User Interface Concepts		C-2	Flexible Coupling and Modeled Collaboration Mode
UbiComp	U-1	Multiple and Heterogeneous Devices		C-3	Multiple-User Devices
	U-2	Multiple-Computer Devices	UbiComp & CSCW	UC-1	Collaboration with Heterogeneous Devices
	U-3	Context and Environmental Awareness			
	U-4	Dynamic Configuration			
UbiComp & HCI	UH-1	Adapted Presentation	SE	S-1	Generic Functionality—Reusability
	UH-2	Multiple-Device User Interface and Interaction		S-2	Tailorable Functionality—Extensibility
	UH-3	Physical Interaction			

✓ = requirement is fully supported  
(✓) = requirement is partially supported

Table 3-3. Comparison of the UbiComp models and infrastructures against the requirements for a conceptual model for the software infrastructure for roomware environments. Requirements C-1 and C-2 are supported by most CSCW systems; requirement C-3 is supported by SDG systems. Requirements U-2, U-3, and UH-3 are not explicitly addressed. This is indicated in the table by the background color of the columns.

### 3.5. Conclusions: What to Use, What is out of Focus & What is missing

The conclusion of the discussion of related work is that the state of the art in the areas of human-computer interaction, ubiquitous computing, and computer-supported cooperative work fulfills only a part of the requirements identified for synchronous collaboration in ubiquitous computing environments. It became clear that currently there is no conceptual model for applications addressing synchronous collaboration in ubiquitous computing environments that provides guidance for designing an appropriate software architecture (Winograd, 2001b; Kazman and Bass, 1995). Still, each relevant research area can contribute important ideas for some particular aspects.

This section, therefore, summarizes the concepts that are used in this thesis. The focus of the dissertation is clarified considering the related topics. Finally, it is analyzed what aspects are missing to meet the goal of this thesis. These aspects are addressed in this thesis.

↓ Section outline

### 3. Related Work

#### 3.5.1 What to Use: Important Concepts

The important concepts that have been identified are listed grouped according to their level: conceptual, architectural, and design level. Each concept is marked in bold.

##### Conceptual Level

Many models stress that a conceptual model has to **separate different concerns**. HCI models, such as MVC, started with the **separation of the domain model from the interaction** aspects (e.g. view and controller). HUMANOID contributes the additional **separation of data and application models**. The Arch model and its refinement PAC-AMODEUS define a separate layer for the **user interface**.

Post-WIMP models like Plasticity or MCRpd identified the need for the explicit modeling of information of the **environment** in order to allow adaptations to different platforms and reactions to physical interactions. A model of the environment is also used in several UbiComp systems such as CoolTown and EasyLiving that keep a representation of “people, places, and things” (Kindberg *et al.*, 2000).

UbiComp models and infrastructures often use a dedicated component responsible for coordinating other components. Examples are MPACC’s coordinator or the environment manager used in Aura. These are **core services** that manage the architecture of applications. Applications being able to manipulate their architecture are called *reflective*.

Apart from separating basic concerns, it is common to define different layers representing **different levels of abstraction**. For example, PAC, Patterson’s generic architecture, C2, and Aura propose to use multiple layers. However, the layers of the Arch reference model mix separate concerns and different levels of abstraction. While functional core, dialog control, and presentation are separate concerns, the virtual presentation and virtual application are an abstracted view on presentation and functional core. UbiComp architectures such as Aura define three layers for runtime, model, and task. The runtime layer is similar to the core services provided by GaiaOS.

To enable synchronous collaboration, groupware frameworks have shown the importance of sharing information and coupling behavior. Depending on the involved devices and available infrastructure, **different distribution architectures** are appropriate to implement the shared-object space. For heterogeneous environments, it must be possible to use **different implementations** for the same conceptual model and convert information when transmitted between devices, as shown in Pocket DreamTeam or Manifold.

##### Architecture Level

In order to implement a conceptual model, many **different architectural styles** can be used. In practice, it might be advantageous to use different styles together to implement different concerns. To implement interaction and user interface, styles such as MVC, PAC, or MPACC can be used. For **pen based interaction**, gesture handling similar to the one provided by SATIN is needed. To access the context information represented by the environment model, **context widgets** (as defined in the ContextToolkit) are a good abstraction. In a dynamic environment, an application model defining an **abstract user interface** together the abstract service descriptions helps adapting to new situations, as tested in Plasticity, XWeb, Aura, and EasyLiving.

To gain more flexibility in combining software components, PAC and the C2 architectural styles propose to use explicit controller facets or **connection objects** for the routing of messages between components or agents. Alternatively, a **blackboard architecture** such as the Event Heap of the iRoom project allows a flexible communication of services.

As overall design rules, the “**minilithic**” **design philosophy** followed in Jazz seems to be appropriate. If functionality is added by composition (using wrapper or decorator objects) instead of inheritance, small components with a well-defined functionality are created that are likely

to be reusable in different contexts. Additionally, promising experiences were made with a **prototype-instance paradigm**, **constraints**, and **interactors** in Garnet and Amulet.

Research in CSCW revealed that for distributed applications a **shared-object space** is a powerful abstraction that often is preferable with respect to synchronization using events. Shared-MVC couples local presentation objects with a shared model. Additionally, XWeb uses a shared abstract description of the user interface. For mobile devices in a dynamic environment, it improves the overall performance to use **optimistic transactions** as in Pocket DreamTeam, COAST, and XWeb.

To allow multi-user devices, it is necessary to have **multi-user event handling** and concurrency-safe widgets as provided by MMM, Pebbles, and MID.

#### Design and Implementation Level

To provide the software infrastructure needed to implement an architecture, **software frameworks** have certain benefits. Frameworks offer a proven design together with many reusable components while being open for extensions.

Such a framework would be classified between a “system infrastructure framework” and a “middleware integration framework” according to the classification schema by Fayad (1999). The resulting infrastructure can be seen as a **middleware operating system**, similar to GaiaOS.

#### 3.5.2 What is out of Focus for the Dissertation

As shown above, the research areas that are relevant to this thesis are very wide. This section, therefore, describes the focus of this thesis to clarify which related topics are examined in what depth.

The goal of the proposed conceptual model for applications supporting synchronous collaboration in roomware environments is to **cover a wide range of applications** in this area. Necessarily, a conceptual model has to be at an abstract level, leaving room for individual designs and implementation approaches. Especially, the conceptual model makes no restriction for architectural style and distribution architectures.

The **software infrastructure for the roomware components** created as part of the i-LAND project is used as sample application of the conceptual model. Having a concrete application allows making decisions for all aspects where the conceptual model allows several choices. The choices that were taken are mentioned along with the presentation of the architecture and design.

However, there are many related topics that this thesis does not address explicitly; it rather relies in some parts on work done by others. Some of these topics are mentioned in this chapter. For example, no sophisticated model for the **physical environment** is created in this thesis. Instead, several models of the environment are being developed by other researchers, e.g., in EasyLiving or CoolTown. In this thesis, a simple model of roomware environments is used. The same applies to models and framework for **context awareness**. The used **document model** is also not the focus of this thesis.

With respect to the need for flexibility and mobility, no mechanisms for **code migration** are included, although it should be straightforward to include it in the presented framework. No mechanisms are currently provided for dynamic service handling, such as **service discovery**; solutions have been developed by other projects, e.g. Gaia or iRoom. As all roomware components use visual displays and pen input, there is no need for **automatic adaptation of data formats**, **automatic generation of user interfaces** from abstract descriptions, or support for **multi-modal input**. However, this could be integrated as part of the data, user interface, and interaction model. As the roomware components currently reside in the same local network, there is no strong need for secure transmission of information. **Secure access on data and privacy** is also not a focus of this thesis, although this might be crucial for real-life applica-

### 3. Related Work

tions. The thesis does not contribute new **architectural styles** and technologies for implementing **distribution architectures**. Also, issues such as **mobile databases** are not investigated.

#### 3.5.3 What is missing: Contributions of this Dissertation

As seen, several relevant concepts have been developed. In order to support the development of groupware applications in ubiquitous computing environments, the related state of the art is not sufficient. Models from **human-computer interaction** have no support for collaboration and multi-device interaction. Approaches from **ubiquitous computing** support interaction in heterogeneous environments, but have the focus on individuals rather than synchronous group collaboration. **Computer-supported cooperative work** concentrates on the collaboration aspect, but assumes homogeneous devices used for cooperation.

However, as ubiquitous computing environments will be used for collaboration, better support is needed for building appropriate applications. A **conceptual model** must include all aspects of interaction and collaboration in ubiquitous computing environments.

---

↓ Next chapter

The next chapter presents the BEACH conceptual model that has been developed to fulfill all requirements discussed in chapter 2. It extends the most promising approaches discussed in this chapter to form a unified model that is feasible for supporting synchronous collaboration in ubiquitous computing environments.

## 4. A Conceptual Model for UbiComp Applications

---

In order to have a high-level structure of applications for ubiquitous computing environments, the BEACH conceptual model is presented in this chapter. After identifying the relevant design dimensions, the *three design dimensions*—separation of concerns, coupling and sharing, and the level of abstraction—are discussed. At the first dimension, *five basic concerns* represent aspects of ubiquitous computing applications that need to be separated: interaction, environment, user interface, application, and data. The second dimension explores *coupling and sharing* issues based on the basic concerns. It is argued that depending on an application's needs the parts related to different concerns (as defined by the first dimension) have to be shared in order to achieve the desired functionality. This extends the current view on handling sharing in ubiquitous computing applications. Four conceptual *levels of abstraction* constitute the third dimension of the BEACH model: the core, model, generic, and task level. Each level helps to reduce the complexity of constructing software components at the higher levels and ensures interoperability, reusability, and extensibility. The chapter closes with a discussion of the BEACH conceptual model, also comparing it with related work presented in the previous chapter. This comparison demonstrates that the BEACH model comprises—and exceeds—what can be expressed by the state of the art.

---

In this chapter, a conceptual model for synchronous ubiquitous computing applications is presented, which has been developed as part of this thesis. It was developed based on the described related work to meet the requirements defined above.

According to the definition by Nowack (1999), p. 29 a “conceptual model describes a conceptual understanding of something, and it is based on concept formation in terms of classification, generalization and aggregation. Hence, conceptual modeling implies abstraction”. Abstraction is an essential technique to overcome software complexity by allowing the developer to focus on one specific aspect at a time. Jacobson *et al.* (1992) see system development as model building, and stress that different conceptual models are created depending on which

aspects of a system one wishes to model. Jacobsen (2000) stresses that conceptual models support understandability and reusability.

Definition:  
conceptual model

In this thesis, we use “conceptual model” as defining the very high-level structure of an application (Phillips, 1999, p. 3; Coutaz, 1997, p. 5), sometimes also called “conceptual architecture” (Anderson *et al.*, 2000). By using this structure for applications, basic components are identified that have a clear separation of concerns, thus supporting their independence and increasing their flexibility and adaptability (see section 3.1 and Alencar *et al.*, 1999).

The model presented here is an updated version of the model published in (Tandler, 2001b), adding a third dimension for *coupling and sharing* (Tandler, 2004). In addition, a graphical notation to visualize the model in design diagrams is proposed. This first version of the model was published in (Tandler, 2001a).

↓ Chapter outline

This chapter first identifies three design dimensions of the conceptual model. Subsequently, its properties are discussed, according to the design dimensions. The succeeding section summarizes the conceptual model. To examine the relations of the BEACH model to related models, the model is compared to related work. Finally, the model is discussed. Examples of how the conceptual model can be applied in the design of a concrete architecture for roomware applications are shown in the next chapter.

#### 4.1. Design Dimensions

In order to identify the design dimensions for a conceptual model, results of all contributing research areas (identified in section 1.1) have to be considered. Looking at these four areas, contributions for a conceptual model can be identified (fig. 4-1):

- Human-Computer Interaction (HCI) is concerned with *user interface & interaction*.
- CSCW has identified different degrees of *coupling* and different mechanisms for *sharing*.
- Ubiquitous computing (UbiComp) has to deal with *device heterogeneity* and the relation to the *environment* in which the devices are used.
- And, finally, *separating specific concerns* and defining *levels of abstraction* are very important software modeling techniques.

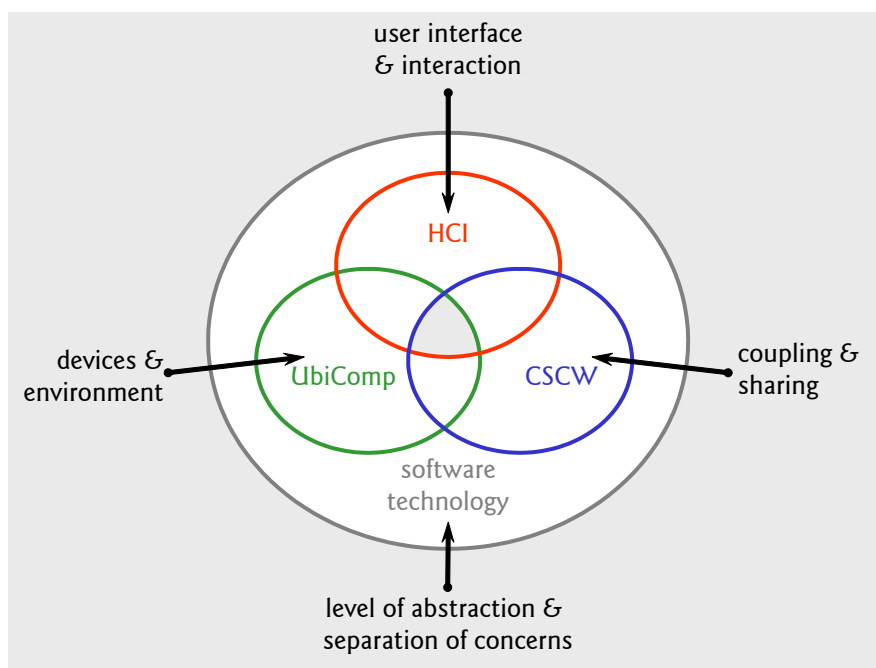


Figure 4-1. Contributions to collaborative ubiquitous computing applications by the different relevant research areas, extending figure 1-2.



These contributions can be arranged as three design dimensions: *separation of concerns*, *coupling and sharing*, and *level of abstraction*. The contributions “degree of coupling” and “level of abstraction” define a dimension on their own. In contrast, “user interface & interaction” and “devices & environment” represent different concerns of UbiComp software systems that should be separated to simplify building abstractions and models (Jacobsen, 2000; Alencar *et al.*, 1999). Hence, they can be combined to a single dimension. Separation of concerns and levels of abstraction are two independent properties of a system structure (Parnas, 1972). This allows seeing them as independent dimensions.

## 4.2. The BEACH Conceptual Model

The BEACH conceptual model is a generic model providing a structure for all kinds of applications supporting synchronous collaboration in ubiquitous computing environments. Before the three dimensions are discussed in detail, an overview of the conceptual model is shown in figure 4-2. Looking at the dimension of the level of abstraction and the dimension of the separation of concerns, these two dimensions form a grid, which can be used to place software components or assign software functionality (see fig. 4-12 below). In contrast, the degree of coupling specifies the level of collaboration for this functionality rather than defining or categorizing the functionality itself.

In order to be applicable to a wide range of applications and architectures, the model specifies a coarse-grained structure at a high level of abstraction. Thereby, the conceptual model leaves much freedom for application developers and architects to choose approaches appropriate for the problem at hand.

Foremost, the conceptual model does not impose a restricted set of *architectural styles*. Rather, many architectural styles can be used to implement the model. The same is true for the *distribution architecture*. Depending on the constraints of the platform and requirements in terms of collaboration, a different distribution architecture can be selected.

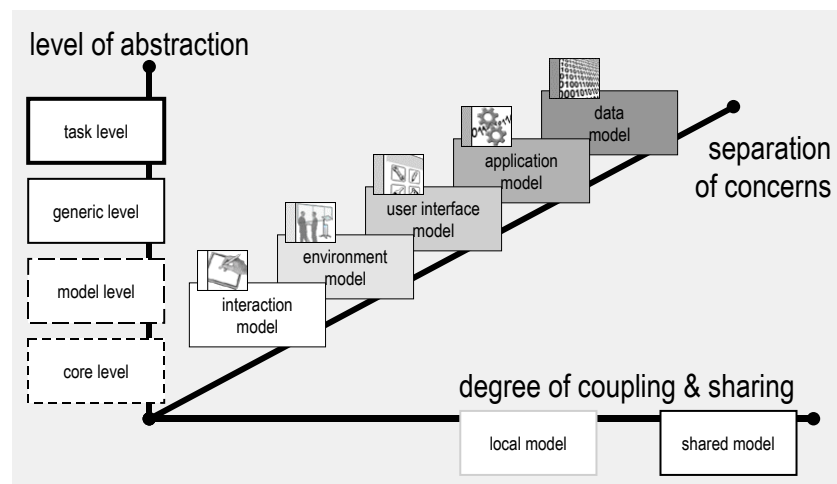


Figure 4-2. Notation for the three design dimensions of the BEACH conceptual model

Figure 4-2 suggests a graphical notation that can be used in design diagrams to denote the position of classes within the design dimensions of the conceptual model. The *level of abstraction* is indicated by the border line style. The higher the level of abstraction, the thicker is the line. The *degree of coupling* is shown by the color of the border; a black border denotes fully shared models, light gray stands for local models. To help the *separation of concerns* the five basic models are indicated by the background color and an icon. This aids developers in understanding the design of a ubiquitous computing application.

These three design dimensions—*separation of concerns*, *coupling and sharing*, and *level of abstraction*—constitute the basic dimensions of the conceptual model proposed in this thesis. Each of these dimensions will be discussed next.

### 4.3. First Dimension: Separating Basic Concerns

*Separation of concerns* is a principle to structure a software system. Abstractions are defined at the same level of abstraction, in order to simplify a problem. This structure is also called *horizontal decomposition*. The BEACH model proposes to separate five basic concerns: data, application, user interface, environment, and interaction models.

As described above, it is necessary for different devices to have different user interface elements (req. H-1). Oppositely, different functionality is useful depending on the device(s) at hand (req. S-2). Clearly separating different responsibilities within the software helps provide the flexibility that different devices need (req. S-1). Therefore, we distinguish models for the data, application, user interface, environment, and interaction, as figure 4-3 shows. The term “model” here refers to a part of an application handling a specific concern (Jacobsen, 2000). The importance of separating distinct concerns is reviewed in section 3.1.

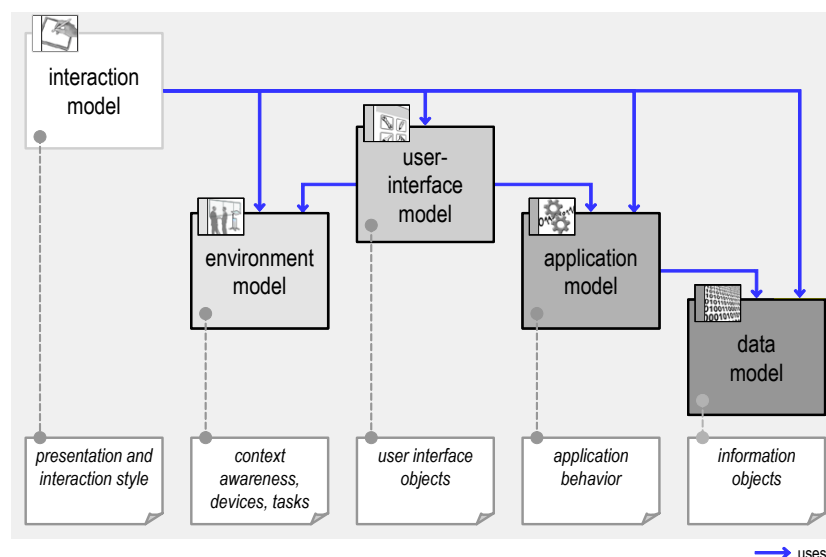


Figure 4-3. Dependencies between data, application, environment, user interface, and interaction models. The user interface model can draw on information available in the environment model to define an application’s user interface that is adapted to the current environment.

The *data model* specifies the kind of data the users can create and interact with. To work with data, the *application model* provides the necessary functionality. These two models are independent of the currently used or supported hardware device. Instead, the *environment model* describes available devices and other relevant parts of the environment. The *user-interface and interaction models* define the framework for how the software can present its functionality to the user, taking into account the properties of the environment model.

These models are applicable to other applications besides those for ubiquitous computing and roomware, as well. Yet, because of the heterogeneous environment in which ubiquitous computing applications operate, they have a strong need for a structure that is clear yet flexible enough to adapt components independently for different situations.

According to the definition given in (Demeyer *et al.*, 1997), the five basic models represent axes of variability (see also section 3.1) of roomware applications. For each model, several

characteristics are possible. The model itself, thus, has to incorporate the parts common for all characteristics.

When applying the conceptual model to design a software system, the developer has to decide how to create an appropriate class structure. As the five basic models refer to different aspects of an application, it might not be sensible to use one class per model in a design. In many cases, several classes have to be used for every model. On the other hand, for simple applications it might be sufficient to use only a small number of classes that combine the aspects represented by different models. In this case, the models represent different facets of a single component (Anderson *et al.*, 2000; Calvary *et al.*, 1997).

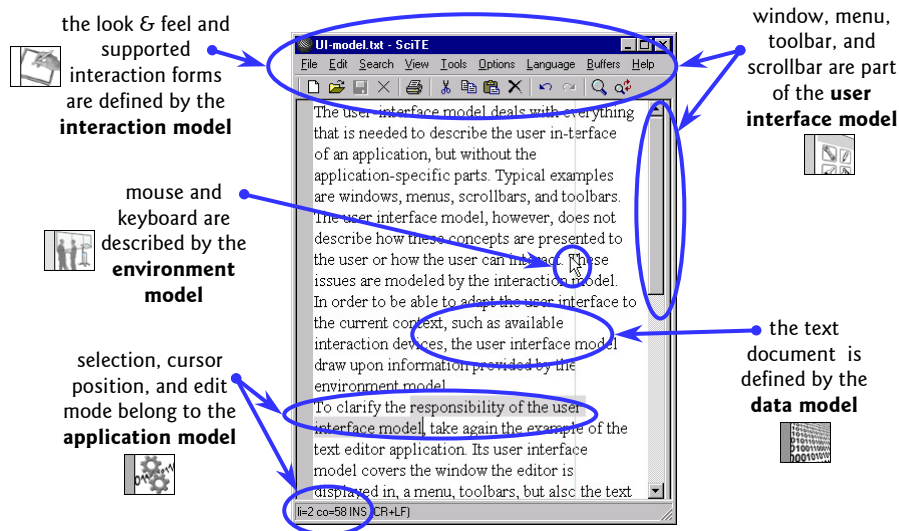


Figure 4-4. The text editor sample application's interaction model visualizes information that is defined by data, application, user interface, and environment models.<sup>16</sup>

↓ Section outline

In the following, these five models are presented in more detail, including their relationship to the previously identified requirements. Concrete examples how these models have been applied are given in chapters 8 and 9. As a simple example for illustrating the different concerns, a text editor application is used (fig. 4-4). The example is not intended to be specific for ubiquitous computing or roomware environments.

#### 4.3.1 Data Model: Information Objects

The *data model* specifies the kind of data the users can create and interact with. It is independent of the currently used devices.

A common approach in application modeling is to separate the application model from the data, domain, or business object model (see examples in chapter 3). The data model relates to the *information dimension* identified by Jacobson *et al.* (1992), while the application model represents the *behavior dimension*. This way, both data and application models can be reused independently.

Different applications can be specified and implemented for one kind of data. This reuse can save much time if the current application domain has complex data structures or algorithms. Conversely, application models can be reused for different types of data, if the interface be-

<sup>16</sup> The screenshot shows the SciTE text editor, which is freely available from <http://www.scintilla.org>. It was not developed using the BEACH model.

tween the application and the data has been defined very carefully at an appropriate level of abstraction.

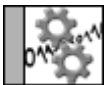
The data model defines the classes and functionality of all objects that can be part of a document. According to an object-oriented view, data objects combine document state with methods to change the state. In the context of cooperative work (req. UC-1), it makes sense to choose a fine-grained model to gain more flexibility in defining different aspects of collaboration, like the degree of coupling. In (Anderson *et al.*, 2000) the model facet represents the data model.

Following an object-oriented approach, the data model will usually consist of a network of multiple connected objects. For hypertext-like documents, e.g., it is popular to define one main containment hierarchy with additional connections defined by hyperlinks.

Example 4-1:  
Data model

*Let's take the example of a text editor application. The data a text editor works with is the text documents it can edit. This includes the text itself, but also its formatting, such as fonts, styles, and text sizes. In general, the data model covers everything that is traditionally stored in document files. A detailed example of a data model is given below as part of the description of BEACH's data model (see section 7.2).*

Depending on the actual application, data objects are not restricted to represent what is classically seen as a "document". In (Edwards and LaMarca, 1999) a much broader view on documents is described. If, for instance, physical devices, people, or tasks are treated as special kinds of "documents", a uniform interface can be used. The term "domain model" is sometimes also used for the concept of a data model (ParcPlace-Digitalk, Inc., 1995; Schuckmann *et al.*, 1999). The word "domain" stresses that the model represents artifacts of a given application domain, which may not necessarily be documents. Although the term "domain model" can be used interchangeably with "data model", this thesis uses the latter term in order to provide a clearer contrast with the application model.



### 4.3.2 Application Model: Application Behavior

The *application model* provides functionality to work with, create, and modify the information that is defined by the data model. To ensure the reusability of the application model in ubiquitous computing environments, it is independent of the current environment, such as available interaction devices.

Application models describe all platform- and interface-independent application aspects, such as manipulation of data objects. It also includes the necessary editing state. As application models define the *behavior* of the application, they specify control objects as defined in (Jacobson *et al.*, 1992). The ontological model of the conceptual model for groupware as defined by Ellis and Wainer (1994) covers aspects of both data and application models.

Example 4-2:  
Application model

*Looking at the example of the text editor, its application model covers what is needed to edit texts. This includes the cursor position, current selections, editing modes (such as insert or overwrite mode), and the reference to the document to be edited, plus the functionality that uses these abstractions to actually modify the associated document. BEACH's application model is described in section 7.2.2.*

To use different application models for the same data model, the data model must remain unaware of any application model, but only represent the document state.

It has proven helpful to choose a rather fine granularity for some application models<sup>17</sup>. This way, low-level application models with a well-defined functionality (for example, to edit a

---

<sup>17</sup> To make it even more confusing, fine-grained application models are called "value models" in (ParcPlace-Digitalk, Inc., 1995) to distinguish them from more complex application models. As this separation refers to a different *level of abstraction* as opposed to a different *concern*, in this thesis, only the term "application model" is used.

simple text field) can be aggregated into more complex models at a higher level of abstraction (for example, an editor that can manage complete workspaces). Usually, a whole hierarchy of application models composed of generic, reusable parts and custom parts constitutes an application (Schuckmann *et al.*, 1999). This way, the application model often forms a hierarchy that is isomorphic with respect to the containment hierarchy of its associated data model (ParcPlace-Digital, Inc., 1995).

Using small application models turned out to foster a new conception of what is regarded as an application. The application model is seen as a description of *additional* semantics for a data model, instead of the conventional view of data as a supplement that applications will edit. This change in viewpoint, therefore, leads to an *information-centric* perspective of application models (Winograd and Guimbretière, 1999; Grimm *et al.*, 2002; Esler *et al.*, 1999).

In the context of ubiquitous computing environments, it is essential that developers do not include user interface and environment aspects in the application model. Enforcing a strict separation between application model and device-dependent aspects makes it possible to reuse application models with different user interfaces and within a different environment.

#### 4.3.3 User-Interface Model: Interface Objects

The *user-interface model* deals with everything that is needed to describe the user interface of an application, but without the application-specific parts. Typical examples are windows, menus, scrollbars, and toolbars. The user interface model, however, does *not* describe how these concepts are presented to the user or how the user can interact, as these issues are modeled separately by the *interaction model*. In order to be able to adapt the user interface to the current context, such as available interaction devices, the user interface model draws upon information provided by the *environment model*.

As traditional operating and window management systems are suited for a traditional desktop PC, their user interfaces have drawbacks when used with devices without a mouse and keyboard, or those having different forms and sizes. For instance, if a menu bar were always at the top of the screen in a wall-sized display (such as DynaWall, see section 2.1), users would find it difficult to reach (Pier and Landay, 1992). Alternatively, toolbars take up a lot of precious screen space on a small device, such as a personal digital assistant.

Accordingly, the user interface model could define alternative user interface concepts suitable for different interaction devices (req. H-1), for example, rotation of user interface elements on horizontal displays. Multiple-computer devices (req. U-2) and multi-device interaction (req. UH-2) make it necessary to have user interface elements that can be distributed and shared among different devices. To choose an appropriate user interface, the user interface model can draw on information provided by the environment model.

An explicit model of an appropriate user interface addresses all issues related to the available hardware and the current environment, making applications and documents device and environment independent. Therefore, the user interface aspects have to be separated from information and behavior of applications. This is related to the *interface dimension* identified by Jacobson *et al.* (1992), but refined to acknowledge the additional requirements of ubiquitous computing environments.

Still, the BEACH conceptual model further distinguishes the *user interface* from the *interaction*, to allow accessing a (possibly shared) user interface with different modalities and different devices. The user interface model must not enforce a dedicated presentation or interaction style; this is the responsibility of the interaction model. Rather, the user interface model concentrates on the *elements* offered for interaction. These elements can be a device-independent representation of user interface widgets or interactors. Models of device-independent user interfaces are developed in the context of user interface generation (Ponnekanti *et al.*, 2001; Banavar *et al.*, 2000; Nichols *et al.*, 2002). These tools use a device-independent specification of the user interface to generate a user interface that matches the currently available interac-



#### 4. A Conceptual Model for UbiComp Applications

tion capabilities. To be able to do this, information that is described in the environment model is necessary. In contrast to BEACH, these tools do not distinguish between user interface and interaction models; the generated user interface is always used with the same interaction form. Some of today's application framework or toolkits already support some means of separating user interface and interaction. For instance, the Swing toolkit (Walrath and Campione, 1999) for Java and the VisualWorks Smalltalk application framework (ParcPlace-Digitalk, Inc., 1995) allow the look & feel of the user interface to be switched. This is a simple form of this separation. Figure 4-3 illustrates the dependencies between data, application, environment, and user interface models.

##### Example 4-3: User interface model

To clarify the responsibility of the user interface model, let's take again the example of the text editor. Its user interface model covers the window the editor is displayed in, a menu, toolbars, but also the text pane with scrollbars. However, for all of these widgets, it does not specify the way it is presented to the user. It rather abstracts from the interaction issues. This way, other interaction forms can be supported. Interaction issues are specified by the interaction model instead (see below). In addition to the definition of widgets, the user interface handles the input focus. This is possible, as the user interface model is aware of the available interaction devices, such as keyboard or mouse. Available devices are described by the environment model (see below). Defining the input focus in the user interface model has the benefit of being able to change to focus using different interaction modalities. In this simple example application, the keyboard focus can be changed by clicking with the mouse on a widget. The user interface model for roomware components provided by the BEACH framework is described in section 7.3.



#### 4.3.4 Environment Model: Context Awareness

The *environment model* describes relevant context information, such as available devices, physical environment, and the logical context.

One major property of ubiquitous computing environments is the heterogeneity of the available devices. To provide a coherent user experience (Prante, 2001), the "system must have a deeper understanding of the physical space" (Brummit *et al.*, 2000). This raises the need for an adequate model of the application's physical environment. The environment model covers three different aspects of the relevant environment: the hardware environment, the physical environment, and the logical context (fig. 4-5).



Figure 4-5. The environment model provides information about the platform, available devices, the surrounding physical environment, and the logical context. This may include information about the presence of other people and their tasks.

Therefore, the environment model is the representation of relevant parts of the *real* world. This includes a description of the devices themselves, their configuration, and their capabilities. This is the direct *hardware environment*, which the user interface model can employ in



adapting to different devices (req. H-1). This part corresponds to the platform model defined by the Plasticity framework (section 3.2.1, page 40), or Aura’s notion of environment (section 3.3, page 45).

In addition, the environment model can include other aspects if these aspects influence the behavior of the software. Necessarily, it has to be possible to measure their relevant properties with sensors. Depending on detected changes in the *physical environment*, the software can trigger further actions to reflect the current situation (req. U-4). An example of this is the way ConnecTables establish a common workspace when placed next to each other (see section 8.2).

Besides the physical environment, other contextual information, such as the current task, project, or coworker presence, could influence the behavior of the software—insofar as this information is available to the application. We refer to this type of contextual information as the *logical context* of the application (Schmidt *et al.*, 1999).

Software with functionality depending on physical objects and their properties, or other aspects of the user’s environment (req. U-3) is called *context-aware* (Salber *et al.*, 1999). There is a strong need for context-aware applications in ubiquitous computing environments, as the large number of available devices, services, and tools can be a burden for the user if the complexity for explicit interaction becomes too high. An environment designed to support the users’ needs must aim at *implicit* interaction (Schmidt, 2000). This can be accomplished by using changes in the real world’s state to trigger software functionality.<sup>18</sup> Therefore, the environment model must be capable of expressing relevant information, such as spatial relationships between physical objects.

However, currently, it is difficult for software applications to grasp the physical environment and logical context. Further work must establish how to capture sufficient information about the current environment and to define appropriate models (for example, as by Sousa and Garland (2002)).

*Current software systems have only a very limited environment model. The text editor assumes that every computer it is running on has exactly one mouse, one keyboard, and one monitor. However, other applications are aware of audio hardware or special peripheral devices such as graphic tablets or scanners. In the context of ubiquitous computing, it is essential that applications do not assume a fixed set of available interaction devices to be present. BEACH’s environment model includes information about roomware components. It is described in section 7.1.*

Example 4-4:  
Environment model



#### 4.3.5 Interaction Model: Presentation and Interaction

The *interaction model* defines how users and the software can communicate. This includes the specification of how information is presented to the user and how the user can invoke functionality. Typically, the interaction model describes how the user interface is rendered onto the screen and what happens if users click buttons of a mouse or press keys. By strictly separating interaction issues from the rest of software systems, especially from the user interface model, other forms of interaction can be employed without needing to change existing code.

To support different forms of interaction (req. H-1, UH-3), it is crucial to separate interaction issues from all other aspects of an application. The interaction model is the only place that specifies presentation aspects or interaction style. This way, a software system can adapt its presentation for different contexts, for example, by using a pop-up menu instead of a list box if display space is scarce. It is also possible to choose a different representation—when no display is available, voice-based interaction might still be possible.

<sup>18</sup> However, using detected context to trigger functionality always has the danger of relying on misinterpreted information, which can be very annoying for users.

The interaction model as it is defined here extends existing definitions. Beaudouin-Lafon (2000) defines the interaction model as a “set of principles, rules, and properties that guide the design of an interface. It describes how to combine interaction techniques in a meaningful and consistent way and defines the ‘look and feel’ of the interaction from the user’s perspective”. This view places the interaction model at the meta-level only. In this thesis, the interaction model also refers to the *instantiation* of Beaudouin-Lafon’s definition, i.e. the set of *concrete* interactions that are possible for a given software system. Dewan and Choudhary (1995) define an interaction model as “describing the nature of the state maintained by the user interface”. In contrast, we see the interaction model as describing the means of how to interact with the user interface. Interestingly, the user interface model of the conceptual model for groupware by Ellis and Wainer (1994) mixes aspects of the interaction model (view of information objects), and the environment model (views of participants, views of context), but *not* the aspects that the BEACH conceptual model places in the user interface model.

As an appropriate interaction style depends on the available interaction devices and the associated user interface, a suitable interaction model can be chosen depending on the environment and user-interface model. The interaction model defines a way to communicate with all other basic models, as shown in figure 4-3. This is necessary, as all models can define aspects and functions that can be represented for and accessed by the user.

Example 4-5:  
Interaction model

*The relationships between the five basic models can be illustrated by looking again at the example of the text editor. The interaction model is responsible for input and output of the application. Figure 4-4 shows a text editor that is displayed using a Windows look for widgets. The same application could use a different interaction model in different contexts, e. g. a Motif look for widgets when run under Linux. In addition to the visualization of the user interface, the interaction model shows the data model (the text that is edited) and the application model. The selection is visualized by highlighting the selected text; the cursor position and the edit mode are shown in the status bar at the bottom of the window. This shows why the interaction model can directly interact with all other basic models.*

*To handle user input, the interaction model processes mouse and keyboard input. By separating user interface and interaction model, a different feel can be used. In addition, different input modalities, such as pen gestures or speech input, can be added for the same user interface by defining an appropriate interaction model. As a detailed example in the context of roomware environments, BEACH’s interaction model is described in sections 7.4 and 7.5. Chapter 8 explains how new forms of interaction can be added to the BEACH framework.*

In ubiquitous computing environments, it is often desirable that multiple modalities can be flexibly combined (Myers *et al.*, 2002). The original “Put-that-there” system (Bolt, 1980) shows that one modality (there: speech input) may require additional information that is provided by other modalities (there: hand gestures). This common information, however, can be seen as part of the *user interface* or *application model* state. In this example, the gesture is used to set the selection and input focus for the operation that is given as speech command. This view has the benefit that different modalities are independent and can be exchanged or combined as desired.

When designing an interaction model, the software developer has to choose an architectural style that is appropriate for the supported interaction style. For visual interaction, an adapted version of the model-view-controller style (Krasner and Pope, 1988a; Schuckmann *et al.*, 1996) has proven successful. This way, the MVC’s “model” needs no information about how it is presented, or how users can trigger functionality. However, the “model” of the model-view-controller style is not further specified. It can refer to each of data, application, user interface, or environment model as defined in this section.

### 4.4. Second Dimension: Coupling and Sharing

The *coupling and sharing* dimension of the BEACH model refers to the degree with which models are shared among cooperating devices. Depending on the application



context of a software system, none, some, or all models defined for the five basic concerns have to be shared.

Whenever multiple devices are involved in a software system, the question arises, which parts of the system should be local to a device or shared among several. This has to be clarified for both the application code and its state. While *distributing code* among devices is a technical question unique to every application, *sharing state* has conceptual implications, which this section addresses.

Today, many applications still run entirely local to a single computer, or access only data that is distributed over a network. Aiming at synchronous collaboration, traditional CSCW or groupware systems have two crucial aspects: *access to shared data* and the ability to *couple the applications* of collaborating users (Dewan and Choudhary, 1995). Obviously, this coupling must apply to both data and application models for software running in a distributed environment (Schuckmann *et al.*, 1999).

In the context of ubiquitous computing environments, we must extend this view. In addition to data and application, different devices and applications must exchange information about the physical environment, such as the presence of nearby users or other available interaction devices. The user interface can be distributed among several machines (req. U-2) or among complementing devices (req. UH-2). The sharing dimension of the BEACH conceptual model explores these additional issues.

Based on the separation of concerns that has been previously identified, Dewan's definition of coupling (given in section 2.5, page 29) can be refined. *Coupling* can now be defined as *sharing the same interaction, user interface, environment, application (editing), or data state* among several users or devices. Coupling can thus simply be implemented as accessing the same user interface or application model. This is an important benefit of using shared user interface and application models.

Definition: coupling

Depending on how much state is shared, the *degree of coupling* can be controlled. If all user interface and editing state is shared, a tightly-coupled collaboration mode is realized; if only the same data model is shared, users work loosely coupled (req. C-2). This is related to the coupling model described in (Dewan and Choudhary, 1995). Examples of how collaboration can be modeled are given in section 7.6.

Even if some models are not coupled, one can profit from sharing environment, user interface, and application models. As the information encapsulated in the models is accessible to all clients, it is possible to provide *awareness information* in the user interface. Typical for CSCW applications is the provision of workspace or activity awareness (Gutwin *et al.*, 1996; Nomura *et al.*, 1998). This can easily be realized if the application model including all editing state is shared (Schuckmann *et al.*, 1999). While tightly coupling the user interface can be inconvenient (Gutwin and Greenberg, 1998; Stefik *et al.*, 1987b), shared user interface information provides a means of giving additional awareness hints to remote users.

Beyond the provision of awareness in CSCW systems, sharing the environment model allows a new kind of awareness for ubiquitous computing environments. The information embodied in the environment model can be used to give environmental awareness.

This section discusses the aspects of sharing of the basic models. Before starting a detailed discussion, it has to be noted that "sharing" can be implemented in many different ways. In the case of collaborating devices with quite varying properties—especially in terms of memory, performance, or network connection—a shared object does not need to have the same implementation for different platforms (see Manifold and Pocket Dream Team presented in section 3.4.4). For example, a shared "image" object is likely to have a different implementation on a high-end desktop PC than on a PDA. At the conceptual level, however, both implementa-

↓ Section outline

tions represent the same shared object.<sup>19</sup> This section extends the example of a text editor application to be a cooperative editor to illustrate aspects of sharing.



#### 4.4.1 Sharing the Data Model: Collaborative Data Access

Sharing the data model enables collaboration using common documents in a distributed environment.

To access and work collaboratively with shared data (req. C-1) it is widely agreed that a shared model for documents reduces the complexity in dealing with distributed applications (Phillips, 1999). While there are well-established models defining a shared data model providing only read access (e.g. the World-Wide-Web), it is much more complicated to allow simultaneous modifications at a fine granularity.

Example 4-6:  
Sharing the data  
model

For a shared text editor, sharing the data model is essential, as this is the information that should be edited by multiple concurrent users (fig. 4-6). The software must ensure that all users access a consistent data model.

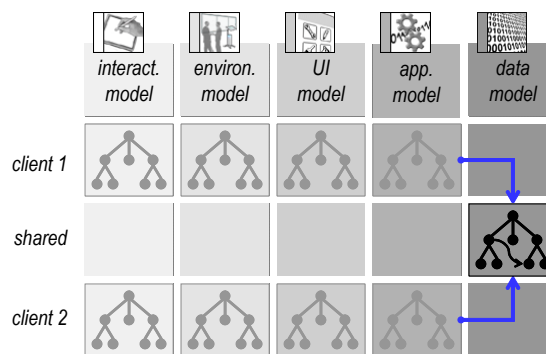
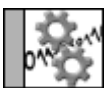


Figure 4-6. A shared data model is essential for synchronous collaboration.

Most popular toolkits and frameworks for computer-supported cooperative work provide some mechanism to manage a shared-object space. In toolkits with a centralized architecture (Patterson, 1991), the document is necessarily shared. Replicated (or semi-replicated (Phillips, 1999)) systems create a shared-object space by synchronizing the replicated objects (Urnes and Graham, 1999; Anderson *et al.*, 2000; Schuckmann *et al.*, 1996). In later versions of GroupKit (Roseman and Greenberg, 1992; Roseman and Greenberg, 1996b) shared “environments” have been introduced as shared data structures that can trigger callbacks upon changes.

Application designers thus have to decide to what degree or for which parts of their application shared access to data is desirable or necessary. In the example of a team member sitting in a CommChair and working with another member at the DynaWall (section 2.2), both users have access to the same information objects (i.e. data model) and can modify them simultaneously.



#### 4.4.2 Sharing the Application Model: Workspace Awareness & Degree of Coupling

Sharing the application model allows tight coupling of editing state and awareness about other users’ actions.

<sup>19</sup> In the case of sharing between PC and PDA, it is likely that conceptually different implementations of interaction and user interface models are used. The example given in section 9.2 uses different interaction and user interface models, and a combined implementation of the data and application model, although the data model is conceptually shared with a different implementation on the PC.

As an easy way of getting information about the editing state of other users, it has been proposed to share the data model as well as the application model (Schuckmann *et al.*, 1999).

Sharing the editing state allows accessing information about who is working on which document, providing awareness information to collaborating team members. For example, action indicators (Gutwin and Greenberg, 1998) can provide visual feedback for actions performed at a DynaWall by a user sitting in a CommChair.

*Theoretically, it would be sufficient to share the data model in order to implement a shared text editor. In practice, it has been found that collaboration is inefficient if users are not aware of other users' actions (e.g. Gutwin *et al.*, 1996). Taking again the example of a text editor, sharing the application model opens the opportunity to provide awareness indicators for other users' actions (fig. 4-7a). The text cursors or selections of remote users can be accessed and visualized if the application model is shared. In addition, a shared application model can be used to model different collaboration modes. For instance, the text editor uses a loose coupling of cursors and selections (fig. 4-7b); i.e. all participating users can have separate cursor positions and personal selections. The reference to the current document is shared by all users. This enables a tightly-coupled navigation, as if one user switches the document, all others will follow.*

*In BEACH, different instances of the workspace application model are used to allow specifying different rotations of the workspace for two users working at an interactive table (section 7.6.3, req. H-1).*

Example 4-7:  
Sharing the  
application model

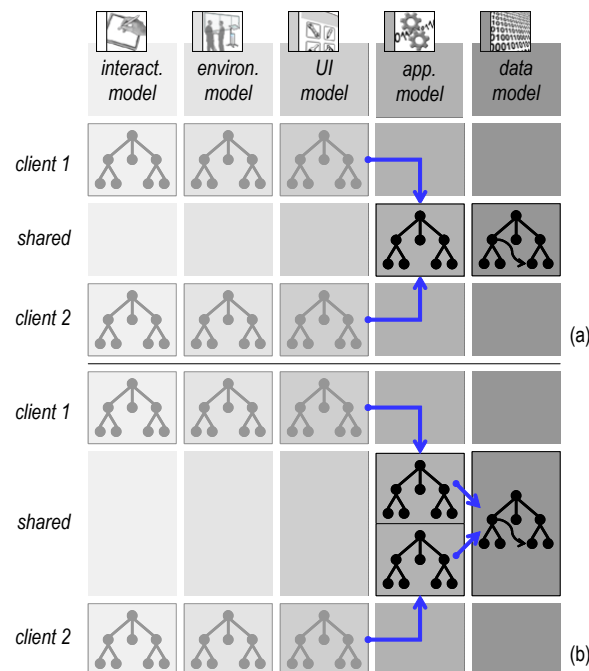


Figure 4-7. Local application models enable separate cursor positions, but no awareness and coupling. (a) Sharing the application model enables tightly-coupled collaboration and activity awareness. (b) If separate instances of the shared application model are used for different clients, some aspects of collaboration can be loosely coupled.

By changing the state of the application model, the software can control possible work modes such as the degree of coupling (req. C-2). This means it controls which parts of the editing state are shared by which users, and where private values are allowed. Users working with the same application model can work tightly coupled with rich awareness information (Schuckmann *et al.*, 1999). Tightly-coupled work could for instance include a coupled scroll position, coupled selection, or coupled navigation. If separate instances of the application model or different application models are used, users can still work loosely coupled when they modify the same data.

Again, the application designer has to decide whether or not a tightly-coupled work mode should be supported or how much awareness information is advantageous.



#### 4.4.3 Sharing the User Interface Model: Distributed and Coupled User Interfaces

Sharing the user interface model allows the coupling of the user interface among distributed devices. In the context of ubiquitous computing, a shared user interface model reduces the effort necessary to implement cross-device user interfaces and multiple-computer devices.

If one user interacts with different devices at the same time (req. UH-2), it is useful to coordinate their user interfaces. This is only possible if all involved devices can access information about the current user-interface elements. Depending on how much state collaborating users share, software can control the degree of coupling. Sharing all user interfaces and application state produces a tightly-coupled collaboration mode; sharing only the same data model creates a loosely coupled environment.

In addition, a shared user interface model forms the basis for providing awareness about the user interface state of other participating users. This is analogous to a shared application model that can be used to give workspace awareness.

Example 4-8:  
Sharing the user  
interface model

In our example of the cooperative text editor, a cooperative scrollbar could be implemented by sharing the user interface model (fig. 4-8). This way, the scrollbar could work either in a tightly-coupled mode that ensures the same scroll position for all users, or in a loosely coupled mode it could provide awareness about the other users' scroll positions.

In a roomware environment, a user sitting in a CommChair in front of a large DynaWall can view all information at the chair and on the wall at the same time (section 2.2). Consequently, the user would benefit if she could modify the information visible on the wall and remotely control the entire user interface. How this is implemented in BEACH is explained in section 7.6.2. Another example of how user interfaces can be coupled is the "join" operation of "join and capture" (Olsen et al., 2001).

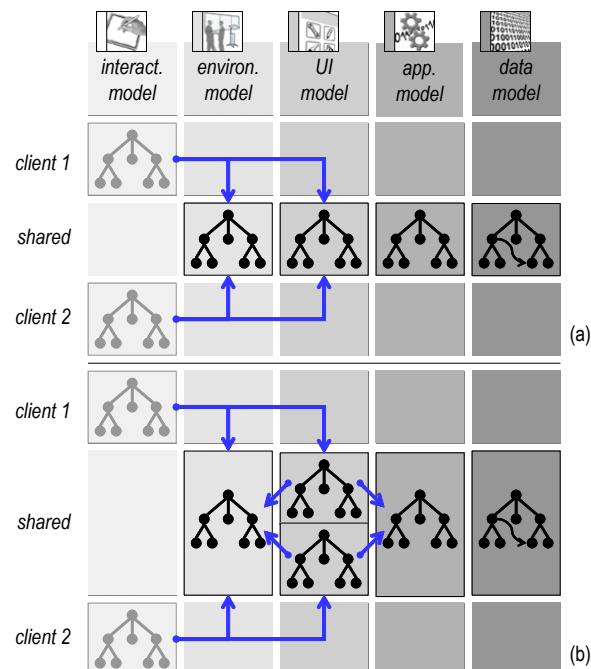


Figure 4-8. Local user interface models enable independent window and scroll positions, but no awareness and coupling. (a) Shared user interface can cross the boundaries of several devices. (b) Independent instances of shared user interface models allow for awareness. As the user interface relies on context information, the environment model must also be shared.

Furthermore, some devices actually have several embedded computers (req. U-2, e.g. the DynaWall, see section 2.1). When a visual interaction area crosses the borders between displays that are physically placed next to each other, but connected to different machines, it is desirable that the user interface elements be freely movable between the different displays (Streitz *et al.*, 1999).

In these cases, the involved machines must share user interface elements. However, if an application's user interface is local to a single machine, it is not necessary to implement it as shared objects.

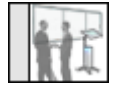
#### 4.4.4 Sharing the Environment Model: Environmental Awareness

A shared environment model is the basis for giving environmental awareness. It also eases the maintenance of the representation of the environment, as different devices can contribute the aspects they can detect.

When several people and devices physically share a common environment, it is obvious that the applications used in such situations should also have a shared model of how their environment looks.

In ubiquitous-computing environments, many devices have sensors that grasp some aspects of the physical environment. By combining all available information and making it accessible to other devices, each application draws on context information that it can use to adapt its behavior (req. U-3). Thus, a shared environment model can serve as the basis for environment or context awareness—similar to the workspace awareness, which is enabled by a shared application model. This is especially important in figuring out which users and interaction devices are currently present and available.

As the environment model of the sample text editor is too simple, we use here the example of *ConnecTables* (see section 8.2) to illustrate the benefit of sharing the environment model. When someone places roomware components, such as two *ConnecTables*, next to each other, the *ConnecTables* update their shared environment model using the information detected by sensors. As soon as *BEACH* observes this change, it triggers functionality to connect the two displays to form a homogeneous interaction area. Currently, the involved sensors are attached to computers built into the *ConnecTables*; future work could replace or augment this set up. A sophisticated object tracking system, for example, as described in (Brummit *et al.*, 2000) involves computers integrated into the environment. Here, a shared environment model enables arbitrary computers to update the information (fig. 4-9).



Example 4-9:  
Sharing the  
environment  
model—*Connec-*  
*Tables*

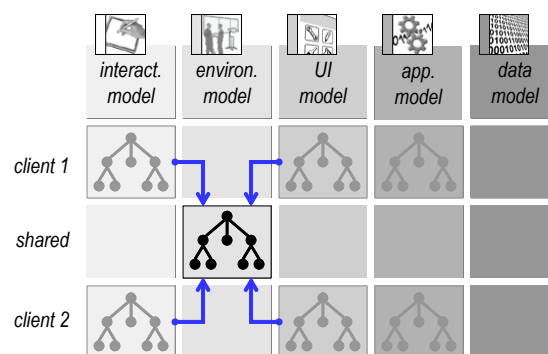


Figure 4-9. Local environment models can represent the local platform and devices, but not much of the surrounding context. A shared environment model allows drawing on information detected by remote devices and adapting to a broader context.



## 4.4.5 Sharing the Interaction Model: Disaggregated Computing

In contrast to the other models, the interaction model cannot be shared completely as it includes components that communicate with the devices attached to the local computer. In the context of ubiquitous computing, however, a shared interaction model can publish the interaction state of local devices to enable disaggregated computing. A local interaction model can be used to adapt the presentation and interaction style to the local context of a device.

The advantages of implementing the data, application, user interface, and environment models as shared objects to give several users or devices the possibility to access these objects simultaneously have been discussed. In contrast, some objects of the interaction model must necessarily exist local on each machine. This is necessary because interaction model objects communicate with the interaction devices that are attached to the local computer.<sup>20</sup>

In a ubiquitous computing environment however, the computer to which an interaction device is attached should become irrelevant, leading to what is called “disaggregated computing” (Shafer, 2001). Systems such as PointRight (Johanson *et al.*, 2002b) or Mouse Anywhere from EasyLiving (Brummit *et al.*, 2000) route input events to remote computers and introduce proxy device drivers. These examples show how an interaction model can be partially shared. The sharing can be partial only, as the device drivers remain local to a machine.

Another benefit of a local interaction model is the ability to adapt the interaction style according to each client’s local context, especially to its physical environment and interaction capabilities. An extensive example of how local interaction objects can be used to adapt to their local context is given in (Tandler *et al.*, 2001).

Example 4-10:  
Sharing the  
interaction model

*The cooperative text editor could use a shared interaction model to display telecursors, i.e. representations of the cursor positions of remotely participating users (fig. 4-10). This example, in fact, illustrates the relationship between the interaction and the environment model. While the environment model describes which devices are attached to which station, the interaction model represents their current interaction state, such as the cursor position for a mouse.*

*On the other hand, a local interaction model can be used to adapt the interaction to the local context. For instance, the shared text editor could use a Windows look & feel (fig. 4-4) for clients running on Windows, but a Motif look & feel for clients on UNIX, without implications for the collaboration. In the context of ubiquitous computing, further adaptation might be required. If a user sitting in a CommChair is collaborating with others working at a DynaWall (fig. 2-4), the information displayed at the wall must be scaled to fit into the screen of the CommChair.*

<sup>20</sup> Almost by definition, shared objects can never have a reference to a local object. This would require the local object to be accessible from every machine that has access to the shared object—which would result in the local object being shared. However, the opposite is very well possible: A local object can access any shared object (COAST, 2000b).

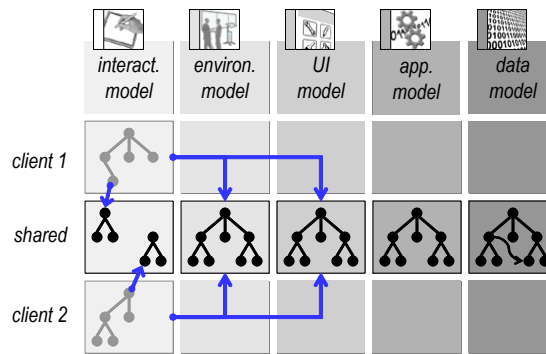


Figure 4-10. Local interaction models enable independent presentation and user control for each device. While the complete interaction model cannot be shared, sharing some of its parts enables disaggregated computing.

#### 4.5. Third Dimension: Conceptual Levels of Abstraction

Separating software into *levels of abstraction* is a common software engineering technique. It reduces the complexity of each level and ensures interoperability, reusability, and extensibility. Introducing levels of abstraction into a software system is seen as its *vertical* structure. The BEACH model proposes to separate four conceptual abstraction levels: the core, model, generic, and task level.

The third dimension of the conceptual model is the level of abstraction. Separating software into levels of abstraction is a common software engineering technique (see section 3.1); it reduces the complexity of each level (Nigay and Coutaz, 1991) and helps ensure interoperability (Hong and Landay, 2001).

For example, a core functionality of the interaction model, such as handling physical interaction devices, belongs to a very low level. Based on this functionality, higher levels define abstractions, such as widgets or logical device handlers. High-level interaction components use these abstractions to define the user's access and interaction possibilities for some other model at the same level of abstraction.

While the C2-architecture (see section 3.4, page 52) places different functionality at different levels (Taylor *et al.*, 1996), we prefer to see the level of abstraction as being orthogonal to functionality. As different functionality should be separated by different basic models, software components implementing one model can belong to different levels. For example, core functionality of the *interaction model*, such as the handling of physical interaction devices, belongs to a very low level. Based on this functionality, abstractions are defined, e.g. widgets or logical device handlers. High-level interaction components use these abstractions to define the user's access and interaction possibilities for some other model being at the same level of abstraction.

In practice, the number of levels that are actually used may vary. As mentioned in section 3.1, in the context of framework development, it has been recommended to define three layers as part of the functional view of the architecture (Succi *et al.*, 1999), the environment layer, the domain-specific layer, and the application-specific layer. These represent three different *conceptual levels of abstraction*. Handling environment and platform issues belongs to the *core* level, domain-specific functionality represents the *generic* level, and application-specific functionality is located at the *task* level. Similar levels are defined in (Tarpin-Bernard *et al.*, 1998) in the context of CSCW systems.

Still, besides the three commonly acknowledged levels, one additional level is needed to represent common abstractions for all basic concerns in an application-, domain-, and platform-independent way (fig. 4-11). This is called the *model* level in the BEACH conceptual model.

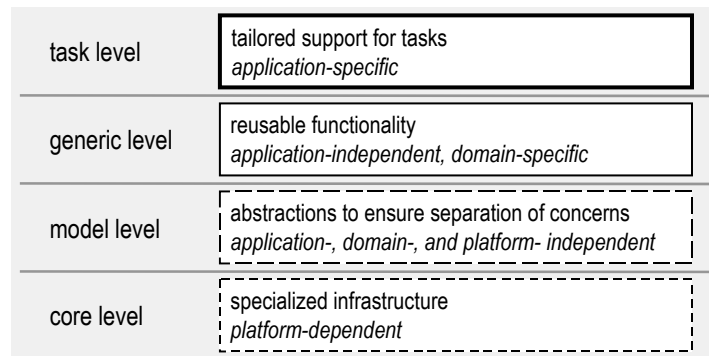


Figure 4-11. Four conceptual levels of abstraction: core, model, generic, and task level

Please note that the term *level* is used in contrast to *layer* to denote a conceptual level of abstraction. A layer is a software technique to structure software architecture and *can* be used to reflect different levels of abstraction in architecture and implementation; a layer can be used to separate different *concerns*.

The remainder of this section discusses these levels, starting at the bottom with the core layer.

#### 4.5.1 Core Level: Specialized Infrastructure

The *core level* defines the platform-dependent low-level infrastructure. It encapsulates the platform-dependent issues in order to ensure portability and reusability.

The core level provides functionality that will make the development of higher-level functionality more convenient and portable by abstracting from the underlying hardware platform and encapsulating platform-dependent details. Functionality normally provided by the (meta-) operating system, middleware infrastructures, or groupware and user interface toolkits resides at this level.

Roomware applications require additional functionality that is unavailable from off-the-shelf libraries or toolkits. This functionality includes support for multi-user event handling and low-level device and sensor management. In the BEACH framework, this also includes implementation of the shared-object space and the dependency mechanism (see chapter 6).

#### 4.5.2 Model Level: Abstractions to Ensure Platform-Independent Separation of Concerns

The *model level* defines application- and domain-independent abstractions to ensure platform-independent separation of concerns.

The model level provides application-, domain-, and platform-independent abstractions that can serve as the basis for the definition of higher-level abstractions. These abstractions can be implemented on top of the core level. This implies that the implementation of the model level maps the platform-dependent abstractions defined at the core level to the platform-independent abstractions constituting the interface of components at the model level.

Components at the model level typically define abstract classes that allow different implementations for different platforms, e.g., using the *abstract factory* or *bridge* pattern as defined in (Gamma *et al.*, 1995). For the platform-independent implementation of user interface and interaction models for instance, it is quite common to use an abstract GUI framework, such as Java AWT, Swing (Walrath and Campione, 1999), or the VisualWorks GUI framework (ParcPlace-Digitalk, Inc., 1995). These frameworks provide good examples of components at the model level.

Here, for example, the BEACH framework implements the model-view-controller style for the interaction model (see section 6.8).

↓ Section outline

task
generic
model
core

task
generic
model
core



### 4.5.3 Generic Level: Reusable Functionality

The *generic level* defines application-independent reusable functionality for a given application domain.

One important goal of every software system is to provide generic components suited for many different situations and tasks (req. S-1). Each application domain has common concepts and algorithms that can be applied by a number of software systems.

Therefore, software developers should group generic models and concepts that apply to a whole application domain at a generic level. In this way, the software developer must think about generic concepts, which will lead to the implementation of reusable elements.

At the generic level, the BEACH framework defines generic document elements, such as work spaces, text, scribbles, or hyperlinks (see chapter 7).

### 4.5.4 Task Level: Tailored Support for Specific Tasks

The *task level* is concerned with application-specific support for specific tasks. It covers abstractions that are unique to small application areas and are not likely to be further refined.

When a conceptual application model defines only generic elements, this restricts the application's usability to some extent. Some tasks require specialized support (req. S-2). Therefore, the BEACH conceptual model has a task level, which groups all high-level abstractions unique to small application areas. For example, we have implemented support for creative sessions on top of the BEACH framework (Prante *et al.*, 2002).

However, in order to increase the amount of reusable components, the application designer should aim to minimize application-specific code. Ideally, an application needs to do no more than glue together existing software components.

To show how the related models and architectural styles match with the BEACH conceptual model, the next section compares the BEACH model against related work.

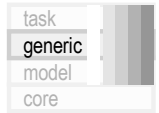
## 4.6. Comparison with Related Work

In this section, the conceptual model proposed in this thesis is compared with related models. Although results from the different research areas (mentioned in section 1.1) influenced this work, we focus here only on human-computer interaction and ubiquitous computing. It is also discussed which related models could be used as an architectural style to implement applications based on the BEACH model. For this discussion, MVC and PAC-AMODEUS have been selected as HCI models; MPACC and iROS for ubiquitous computing, and Dewan's generic architecture and the Chiron-2 architecture for CSCW.

### 4.6.1 Comparison with Software Frameworks

In section 3.1, categories of software frameworks have been explained. In order to help understand the BEACH model, the relationship between the BEACH conceptual model and the different kinds of frameworks and components of interactive systems is discussed here. Figure 4-12 illustrates which parts of an application can be implemented using which category of framework or component. Typical components of interactive applications are discussed by Myers (2003).

Often, software architectures use *layers* to separate different *concerns*. For example, data, business, and presentation are typical software layers that correspond to the data model, application model, and a combined user interface and interaction model. The layers defined by the Arch model (see section 3.2.1) mix concerns and levels of abstractions. However, the figures in this thesis use the level of abstraction as the vertical dimension. This does not imply that the abstraction levels are implemented as software layers.



↓ Section outline

#### 4. A Conceptual Model for UbiComp Applications

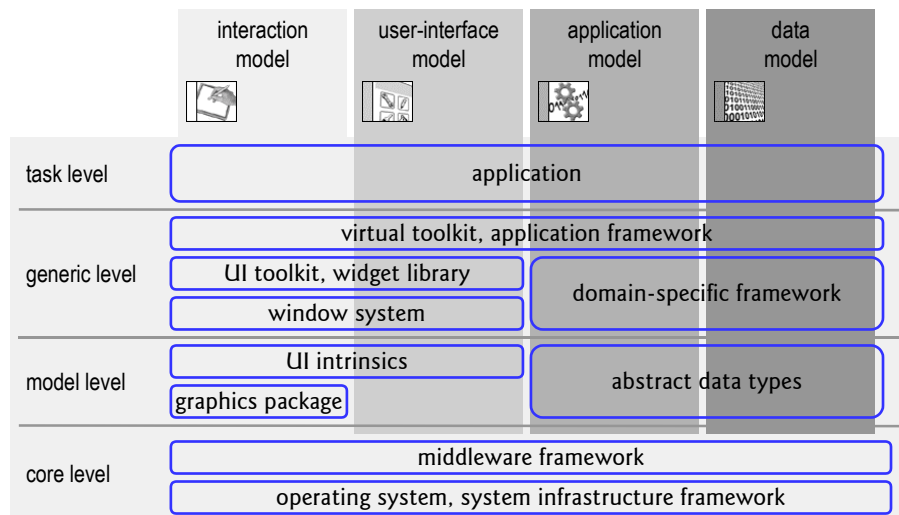


Figure 4-12. The different categories of software frameworks can be used to implement different parts of an application that is structured according to the BEACH conceptual model.

On top, at the *task level*, the software application is placed. It can reuse functionality that is provided at the *generic level*. Application frameworks address the development of complete application, providing a generic application “without functionality” that can be specialized. Virtual toolkits abstract from concrete user interface toolkits. Domain-specific frameworks provide solutions for an application domain, such as digital electronics, aerodynamics, or financial transactions. Domain-specific frameworks typically concentrate on data types and algorithms for their domain; that’s why they can be seen as defining data and application models. To help create user interfaces, widget libraries or user interface toolkits can be used. Window systems belong to the generic level of user interface and interaction models, as they define generic means of managing display space for applications, which is a task of the user interface. The *model level* contains device- and application-independent components. There, components such as device-independent graphics packages, UI intrinsics, or collections of abstract data types can be placed. At the bottom of the *core level*, operating systems handle the communication with the hardware platform. System infrastructure frameworks can be used to develop hardware-related functionality. Above, middleware frameworks abstract from operating system issues, and often provide support for distribution.

##### 4.6.2 Comparison with HCI Models

The HCI models do not address aspects of sharing and distribution. Therefore, the figures in this section only show the first and third dimension of the BEACH model, i.e. different concerns as the horizontal structure and levels of abstraction as vertical structure.

##### Model–View–Controller (MVC)

The model–view–controller paradigm (see section 3.2.1, page 38) is concerned with the separation of the interaction issues from the application and domain logic. Therefore, it concentrates on interaction, application, and data model (see figure 4-13). The user interface model can be seen as part of MVC’s non-specific model, as views and controllers are also provided for user interface elements, such as windows, toolbars, and scrollbars. However, the idea to define the view as an observer on the model has proven very successful, and it can be used in applying the conceptual model. Whether it is helpful to strictly separate input and output behavior depends on the modalities used.

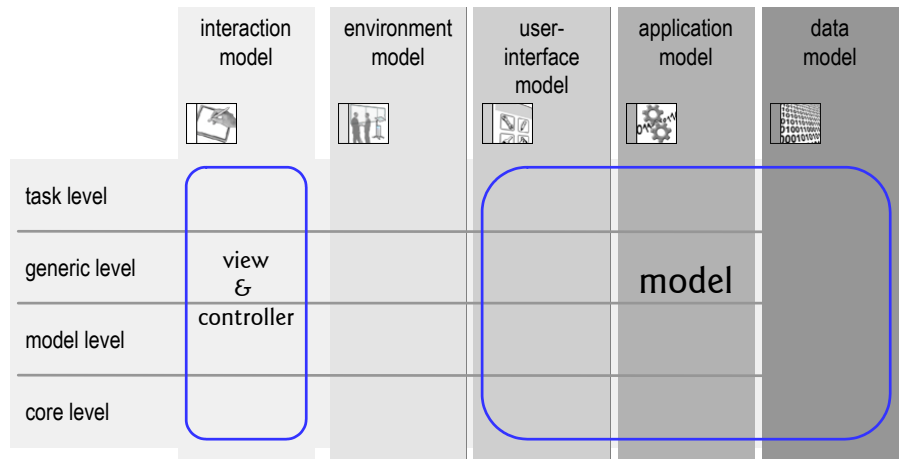


Figure 4-13. Comparison with Model-View-Controller

#### Agent-Style User Interfaces: PAC-AMODEUS

The PAC-AMODEUS style (see section 3.2.1, page 39) combines the Arch reference model (see figure 3-1) with the PAC style to structure Arch's dialogue component. Figure 4-14 shows which parts of the BEACH conceptual model map to PAC-AMODEUS. The data model corresponds to the *functional core* defined in the Seeheim model and Arch. The *functional core adapter* introduced in Arch represents a higher level of abstraction. As the other layers defined in Arch represent different concerns rather than different levels of abstraction, they represent different models in the BEACH conceptual model. The *physical interaction* belongs to the core level of the interaction model. The *logical interaction* provides generic components and defines abstractions for interaction and user interface elements. Therefore, it can be positioned at the model and generic level of the interaction and user interface model.

Finally, the *dialogue* component is structured—according to PAC—in presentation, abstraction, and control (see figure 3-3). These represent different concerns; accordingly, they belong to different models. The *presentation* facet is placed at the task level of the interaction and user interface model, as it defines application-specific presentation functionality. The *abstraction* facet defines application functionality and, therefore, it can be seen as an equivalent of the application model. The *control* facet is responsible for passing of messages between the other facets and between PAC agents. This is a responsibility orthogonal both to concerns and to levels of abstraction. Therefore, the functionality can be found in interaction, user interface, and application model. From the point of separation of concerns, care must be taken that the dependencies (shown in figure 4-3) between the basic models are not violated. The dependencies ensure that, e.g., different user interfaces can be used for the same application, or that the same user interface can be with different interaction styles.

#### 4. A Conceptual Model for UbiComp Applications

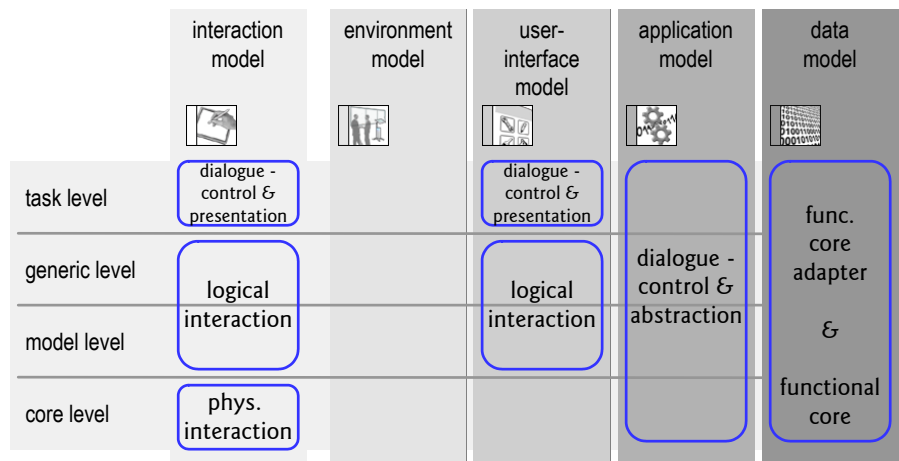


Figure 4-14. Comparison with PAC-AMODEUS: The dialogue component includes aspects of interaction, user interface, and application models. PAC-AMODEUS has no notion of environment model.

#### 4.6.3 Comparison with UbiComp Models

##### Gaia's MPACC: Application Model for Active Spaces

As part of Gaia, the MPACC application model is defined (see section 3.3, page 46). It is also an extension of the MVC paradigm. *Presentation* and *controller* are components belonging to the *interaction model* (figure 4-15). The *adapter* belongs to the *user interface model*, as it transforms the underlying abstractions into a format that fits the interaction model. The *coordinator* can be seen as a low-level service at the *core level*, which coordinates all other models. The MPACC model does not distinguish further between user interface, application, and data model. No explicit notion of an *environment model* is included.

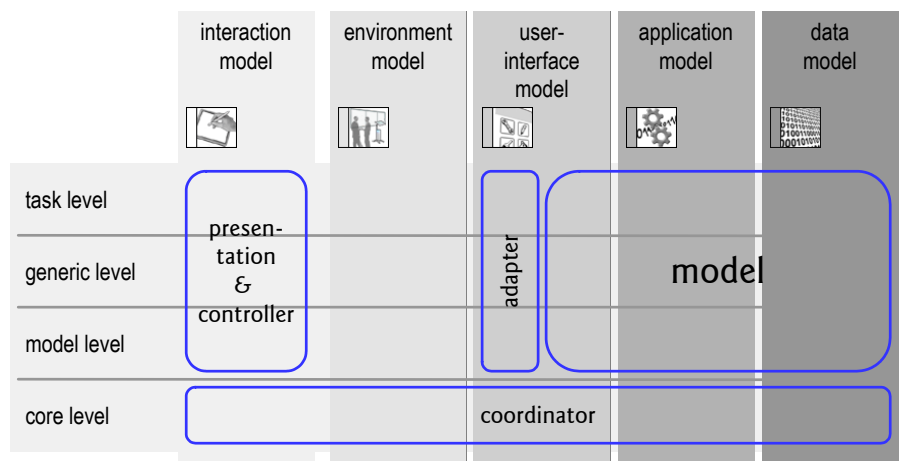


Figure 4-15. Comparison with MPACC (model, presentation, adapter, controller, coordinator): Adapter and coordinator are components introduced to support UbiComp environments.

##### iROS: Infrastructure for Interactive Rooms

As part of the Interactive Workspaces project at Stanford University, the meta-operating system "iROS" has been developed (see section 3.3, page 46). It defines a set of low-level services to facilitate the development of applications for ubiquitous computing environments. The major subsystems of iROS are the Event Heap, the Data Heap, and ICrafter. These are shown in

figure 4-16, together with a sample application called Room Controller. However, iROS itself has no underlying conceptual model such as the BEACH model.<sup>21</sup>

The Event Heap (Johanson and Fox, 2002 or Johanson and Fox, 2004) provides the core infrastructure of iROS, being responsible for coordination of applications and services. It thus can be seen as base component at the *core level*, similar to the BEACH *core.sharing* component shown in figure 5-5. The Data Heap—used to store persistent information—can also be placed at the core level and related to the *data model*. The Context Memory is a part of the Data Heap, holding information about which services and devices are available in the environment. Therefore, it can be seen as core service of the *environment model*. The Service Discovery keeps track of running services, relating it to the *application model*.

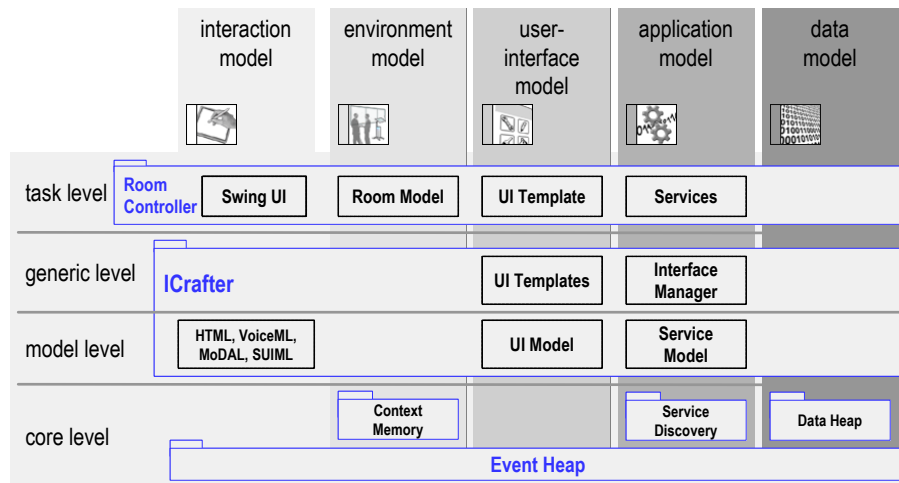


Figure 4-16. Using the BEACH conceptual model to structure the iROS meta-operating system

ICrafter is a well-known service, which is also part of iROS (Ponnekanti *et al.*, 2001). It handles the generation and selection of user interfaces for services. When a client device requests access to a service, ICrafter tries to find the best matching user interface from a set of predefined user interface descriptions for that device, with respect to the device's interaction capabilities. If no matching interface is available, a plain interface can be generated from the definition of the service's functionality and a set of service-independent user interface templates. To accomplish this task, ICrafter defines abstract models for services and user interfaces at the *model level*. Interface description languages, such as HTML or VoiceML, are used to construct the *interaction model* from a user interface specification. At the *generic level*, on the one hand, the collection of service-independent user interface templates form a generic *user interface model* for interface generation. On the other, the Interface Manager is a generic *application model*, constituting the main ICrafter service, responsible for interface selection and generation.

However, ICrafter does not support a clear separation of user interface and interaction model, as it is not possible to define a device independent user-interface description, which can be mapped on arbitrary interaction models. This restricts the range of automatically generated user interfaces, as the user interface cannot have its own state, separate from the application state. For example, the user interface cannot define a cursor position or input focus in a de-

<sup>21</sup> In fact, there is a discussion in the Interactive Workspaces project whether the BEACH model could be adopted to guide authors in developing applications built on top of iROS (private email from Brad Johanson, Aug 8, 2003).

vice-independent manner. Consequently, this information cannot be shared between collaborating devices, as it is described by Olsen *et al.* (2001).

The Room Controller is a sample application running on top of iROS. It consists of services controlling devices within the current room. It offers an interface to turn lights, projectors, and display surfaces on or off. The services controlling devices are part of the *application model*, defining the functionality that can be executed. Handcrafted user interface templates exist as *user interface model*. Templates, e.g. for Java, Palm OS, or HTML, are instantiated based on a Room Model (the *environment model*) to present the currently controllable devices in the room. The Swing UI for the Room Controller is an example of a supported *interaction model*.

##### 4.6.4 Comparison with CSCW Models

Comparing the conceptual model with CSCW models, it is ascertained that the models offer complementing views on the overall architecture of an application. While the BEACH model has the focus on a semantic structure defining important concerns and levels of abstraction, CSCW models deal with distribution architectures or architectural styles to implement distributed systems.

##### Dewan's generic architecture

For example, Dewan's generic architecture—also being adopted by PAC\*—defines different layers separating different levels of abstraction and different concerns, but these layers are used to illustrate different *distribution strategies*. The distribution architecture defines which parts of an application are centralized with shared access, which parts are replicated, and which parts are kept local to a single client. The BEACH model, instead, abstracts from the distribution architecture and only highlights the importance of sharing information, providing the freedom to select (maybe dynamically) an appropriate distribution architecture.<sup>22</sup>

Additionally, the hardware used to present information is also included in Dewan's architecture, as layer zero (see fig. 3-5). The BEACH model, in contrast, takes the approach of including a *representation* of the relevant parts of the available hardware, as part of the environment model. This acknowledges the fact that the available hardware has to be considered when selecting interaction (and also computation) strategies, but stresses the independence of the overall systems from the actually available hardware.

##### Chiron-2

Chiron-2 defines an architectural style about how to model the communication between the software components of an application. It proposes to use communication buses that mediate messages between components residing at different layers. This ensures a high degree of independence between the components and eases the selection of the distribution architecture.

The BEACH model does not propose the usage of an explicit architectural style for the communication or other aspects of the implementation of different basic models. Nevertheless, some architectural styles obviously fit better than others. Chiron-2, although it has not been used in the implementation of the BEACH software framework (see chapters 6 and 7), seems to offer support for the necessary independence between the components implementing the basic concerns.

---

<sup>22</sup> An approach how the distribution architecture can be linked to the conceptual architecture is described in (Anderson *et al.*, 2000).

## 4.7. Discussion of the Conceptual Model

The final section discusses the conceptual model presented in this chapter. It shows its strengths, but also its limits, and discusses alternative approaches. Important aspects are type-set in bold.

The BEACH conceptual model is a **generic model** providing a structure for all kinds of applications for roomware and ubiquitous computing environments. This **wide applicability** is gained by defining structural elements and common concerns at a rather high level of abstraction. As the BEACH model combines models from the research areas of HCI, UbiComp, and CSCW, it can also be applied to applications in just one of these areas by ignoring the parts necessary only to the other areas. In contrast, the **current state-of-the-art models cover only a part** of what is necessary to enable synchronous collaboration in roomware environments (see the discussion in section 4.6).

In conclusion, why does the model look like this? To answer this question, we can look at the three dimensions of the model, illustrated in figure 4-2.

The BEACH model deals with **five basic concerns**. But, why does the model use exactly these five concerns? As shown in section 4.3, all **models represent important aspects** of roomware applications. Every concern should be separated to be able to adapt applications independently for all concerns. Therefore, all models are necessary.

However, some other models mention **additional aspects** explicitly. The conceptual Plasticity framework (see section 3.2.1, page 40) separates the environment model (here understood as context information relevant to the user's current task) and platform model as separate concerns. The architectural framework developed in the Aura project (see section 3.3, page 45) divides the environment model (concerned with the device configuration) from the task model. The BEACH model, in contrast, makes no explicit distinction between different kinds of context information that might be relevant for an application. This way, the developer is free to put the focus on the kind of information that is needed. The model is also kept simpler, specifying only the basic concerns for synchronous roomware and ubiquitous computing applications.

Some other models explicitly define connectors handling the communication between software components. For example, PAC (see section 3.2.1, page 39) defines the "control" facet of a PAC-agent to be responsible for routing messages. Chiron-2 (see section 3.4, page 52) introduced communication buses to pass events between components. These two are examples for architectural styles. Since the BEACH conceptual model aims to be open to implementation with a wide range of different styles, the developer can choose which architectural style for message passing fits best to the given application.

Concerning the number of proposed levels of abstraction, one can also ask whether or not **all defined levels are necessary**, or, whether important levels are missing. The reason for defining the proposed levels has been discussed in section 4.5. The architectural framework defined in the Aura project (see section 3.3, page 45) restricts the model level to a model of the software environment. Also, it makes no distinction between generic and task levels. The explicit notion of a generic level helps increase the possibilities for reuse, as it forces the developer to think about generic elements of the given domain area. Therefore, it has been proposed in the area of software framework development (see section 3.1, page 35) to separate three levels for environment, domain-specific, and application-specific. Here, the explicit notion of a level defining platform-, application-, and domain-independent models—the model level—is missing, which is the key element to ensure the separation of basic concerns and the basis for extensions.

On the other hand, there is **no need to define additional levels**. Models that propose more or a flexible number of levels (or layers) often mix levels of abstractions with the introduction of explicit software layers to implement different *concerns*. For example, the five layers defined by

the Arch reference model (see section 3.2.1, page 38) belong to the interaction, application, and data models—and are separated into two levels of abstraction only. The layers implemented in the Amulet framework (see section 3.2.2, page 41), mix core level, interaction model, and user interface models. Dewan’s generic architecture (section 3.4) allows an arbitrary number of layers to model distribution and communication aspects of distributed systems. This gives no advice about which layers are actually needed. However, using the BEACH model it is still possible to implement a single conceptual level as multiple software layers. Alternatively, the developer is free to define sub-levels if necessary.

As an additional remark, it should be pointed out that **separation of concerns and levels of abstraction are two independent properties** of a system structure (Parnas, 1972). This allows seeing them as independent dimensions.

The third dimension of the BEACH model—coupling and sharing—addresses the aspect of collaboration, especially synchronous collaboration. In section 4.4, the benefits of sharing the basic models have been discussed. Generally, it can be summarized that **sharing offers the capability of providing awareness**. Besides sharing the data—which is unavoidable for synchronous collaboration—a shared application model can be used to provide collaboration awareness and control coupling. A shared environment model offers environmental awareness (including context awareness) among several independent devices. The shared user interface model builds the base for **distributed interaction**.

While the dimensions of concerns and level of abstraction are independent, the combination of the sharing dimension with the others is more constrained. For instance, the **sharing has to be homogeneous between all levels of abstraction for any concern**. It simply makes no sense to share, e.g., an application model at the task level, if the generic application model it is build upon is local. Besides, it is also not possible to reference a local model from a shared one, as (per definition) all references of a shared model are shared as well, and a shared reference obviously cannot point to a local object. This implies that, if an application model is shared, the data model has to be shared as well—according to the references shown in figure 4-3. This is the same constraint that led to Patterson’s “zipper” model and Dewan’s generic architecture (see section 3.4).

---

↓ Next part

The next part shows an application of the BEACH conceptual model. It starts in the following chapter with applying the BEACH conceptual model to develop the BEACH software architecture for roomware components.



## Part II. The BEACH Software Architecture and Framework: Support for Implementing Roomware Applications

---

Part II shows an application of the BEACH conceptual model in the context of roomware environments. Chapter 5 analyzes the concrete needs of the roomware components developed by the i-LAND project and derives the aspects of the BEACH model that require tailoring. This is used to design the BEACH *architecture* for the software infrastructure for roomware environments. In order to support developers when designing and implementing roomware applications, two software frameworks are developed providing a design and implementation of the reusable parts of the BEACH architecture. The BEACH *Model framework* covers the core and model layers (chapter 6). The BEACH *Generic Collaboration framework* implements the generic layer of the BEACH architecture (chapter 7). Using the architecture and these frameworks, developers can implement applications that acknowledge the specific properties of roomware environments much more efficiently, which is discussed in Part III. This part describes the architecture and frameworks in a bottom-up fashion.

---



## 5. Software Architecture for Roomware Applications

---

This chapter presents the design of the BEACH architecture. The BEACH architecture constitutes one major part of the software infrastructure for the roomware components developed as part of the i-LAND project. It is designed to enable scenarios such as the ones given in section 2.2. The BEACH architecture was developed based on the BEACH conceptual model presented in the previous chapter. As basis for the design of the architecture, this chapter analyzes the concrete needs of the roomware components and derives the aspects of the BEACH model that require tailoring. The results are used to define key abstractions at several levels that reduce the complexity of software development for roomware environments. While traditional software systems often use layers to separate software concerns such as data, application and presentation, the BEACH architecture defines software layers according to the levels of abstraction identified by the BEACH conceptual model. This enables to provide low-level functionality as part of the software infrastructure. The BEACH architecture can thus be used as proof-of-concept showing the applicability of the BEACH conceptual model.

---

The conceptual model introduced in chapter 4 was designed for the software infrastructure for ubiquitous computing environments as a major application area. To be widely applicable, it leaves many options for implementation. This chapter presents the BEACH software architecture that was developed as part of this work. It comprises the platform for the *roomware* applications developed in the i-LAND project (section 2.1). This is an example of how the conceptual model can be applied in the design of frameworks and architectures. The architecture of a software system plays a key role as a bridge between requirements and implementation (Garlan, 2000).

With respect to the requirements described in chapter 2, this architecture offers the flexibility necessary for supporting collaboration with heterogeneous devices and ensures extensibility for future devices. Although this architecture was designed as the software infrastructure for roomware components, it has generic elements that are applicable to other ubiquitous com-

puting environments as well. On the other hand, due to the focus on the roomware components of i-LAND, several issues that are relevant for a universal ubiquitous computing software infrastructure have not been addressed. For example, the implementation as a software framework aims at constructing single applications; for a universal UbiComp infrastructure, some of the functionality included in the framework would be provided as services by the infrastructure (Hong and Landay, 2001; Sousa and Garlan, 2002; Johanson *et al.*, 2002a). Also, roomware components have a rather homogeneous software platform; in the general case, the heterogeneity of devices in an UbiComp environment has to be addressed explicitly.

### 5.1. Architecture Overview

The architecture conforms to the guidelines proposed in the conceptual model, according to the three design dimensions (shown in figure 4-2).

- The five basic models separate the *basic concerns* of roomware applications, as identified above.
- A software layer is introduced for every *conceptual level*. In contrast to defining layers for the basic concerns, this way, the lower abstraction levels can be realized as part of the software infrastructure or as a software framework.
- All basic models except the interaction model are implemented as *shared objects* to enable distributed access. In contrast, the interaction model is implemented with local objects to be able to adapt to the local context. This gives maximum flexibility for the design of applications, as discussed in detail in section 4.4.

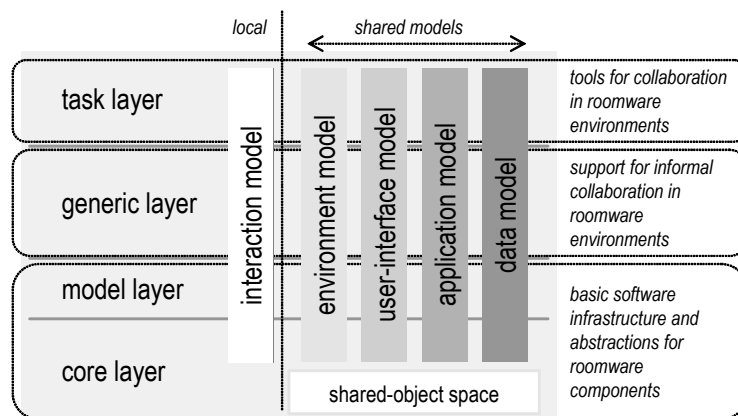


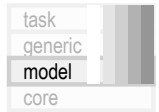
Figure 5-1. Following the BEACH model, the software architecture is vertically organized in four layers defining different levels of abstractions and horizontally by five models separating basic concerns. Crucial for synchronous collaboration is the shared-object space provided by the core layer.

The resulting structure of the architecture is visualized in figure 5-1. Parts of the functionality of the core and generic layers are described in (Tandler, 2001b) in the context of architectures for ubiquitous computing environments. The idea to refactor the reusable parts as a software framework is discussed in (Tandler, 2001c). The remainder of this chapter explains these aspects, organized by the layers of the architecture.

The layers are presented bottom-up, starting with the model layer. The core layer is presented last; it provides low-level abstractions, which are encapsulated by the model layer. A detailed understanding is therefore not necessarily required for the description of the upper layers.

This chapter concentrates on showing how the conceptual model can be applied in the design of a software framework.

The figures in this chapter use the separation-of-concerns and level-of-abstraction dimensions of the BEACH conceptual model to illustrate the software architecture. The degree-of-coupling dimension is not visualized. Arrows denote usage relationships between software components. The notation is summarized in appendix A on page 207.



## 5.2. Model Layer: Domain-Independent Abstractions for Roomware Components

The BEACH conceptual model suggests extracting domain- and platform-independent abstractions to ensure strict separation of concerns to improve extensibility and interoperability (section 4.5). Consequently, the BEACH architecture defines a separated model layer that provides common abstractions for all applications in roomware environments. As discussed in section 4.4, sharing the environment, user-interface, application, and data models provides the necessary flexibility to create roomware applications. Therefore, the architecture ensures that instances of these models are always realized as shared objects. This enables roomware developers to think and design at a high level of abstraction, not caring about distribution issues in the roomware environment. Figure 5-2 shows the components of the model layer. A component is used to specify the basic abstractions for each of data, application, user interface, and environment model. Each component implements the basic models by defining both abstract and concrete helper classes.

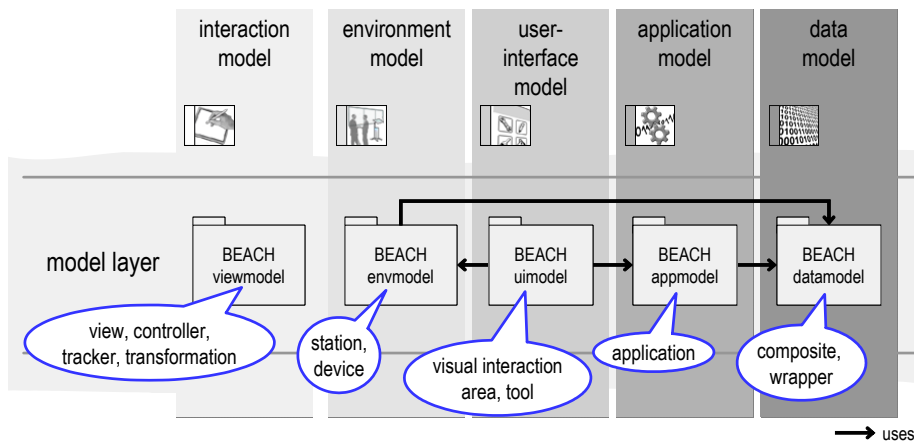


Figure 5-2. Components of the model layer. Every basic concern identified in the conceptual model is implemented by a corresponding component.

The *application model* (BEACH *appmodel*) defines the basic structure of applications. The *data model* characterizes the foundation for the structure of data. *Composites* are abstractions for part-whole hierarchies, providing a common interface for all hierarchies. *Wrappers* can be used to dynamically add functionality to other objects. *Relation* wrappers describe the relationship between objects.

The *visual interaction model* (BEACH *viewmodel*) defines an adapted version of the model-view-controller (MVC) concept as basic architectural style to be used for interaction aspects. MVC has been selected because the roomware components support visual-based interaction. Decoupling input and output allows the creation of different input styles, such as mouse, pen, or finger, for the same visual representation.

The view model defines abstractions for views, controllers, and trackers, and includes a framework for view management, transformations, and event handling. The *view* is the abstraction for any visual representation of a model. By creating local presentations of information as observing shared models, the presentation of common information can be locally adapted. In order to ensure flexible adaptation of presentations, *transformations* are used to modify the output generated by view. *Controllers* map the user's input actions to the behavior

that is invoked in the user interface. This is extended by *trackers*, which track interaction sequences and map these to the appropriate functionality.

Interaction models are always local objects, as they are computed by every device independently, depending on its local context. This way, views can use different scale factors depending on their display size, or show a different part of a common workspace, as in the case of multiple-computer devices (req. U-2), such as the DynaWall (see section 2.1).

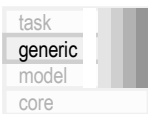
Other interaction modalities besides visual presentation can be easily integrated in this architecture. In section 8.3, an additional component to support auditory feedback is described. For multi-modal interaction, new components like a multi-modal integration component (Oviatt *et al.*, 2000) could be added as part of the interaction model.

The *user interface model* (BEACH *uimodel*) is also designed for visual interaction, due to the focus on visual-based interaction of the roomware components. The main abstractions it defines are the visual interaction area and the tool. The *visual interaction area* represents a part of display space that can be used to render a visual representation of a tool. A *tool* describes the user interface for an application model (or a set of application models).

The low-level abstractions of the *environment model* (BEACH *envmodel*) are stations and devices. The term *station* refers to computers running a BEACH client. Stations can have attached *devices*, which can be both explicit interaction devices—like displays, keyboards, or pens—and sensors that are used for implicit interaction (Schmidt, 2000). While a concrete representation of the environment is modeled at the generic level, introducing station and device as basic abstractions at the model level is important for the extensibility of the overall system. This way it can be ensured that the model level is truly device-independent, but that it is possible to add support for arbitrary devices.

Concerning the dependencies between the components, the architecture mainly follows the dependencies between the basic models as defined in the conceptual model (fig. 4-3). In addition, the environment model uses the data model. This way, all environment model classes can define specialized document classes. If physical elements, like devices or people, can be treated as special kinds of documents, a uniform interface can be provided, reusing concepts and interfaces for documents (Edwards and LaMarca, 1999).

On the other hand, at this layer there is no direct dependency between the interaction models and the other models as it is present in the conceptual model. The reason is that the interaction is based on a more generic notion of “model” defined in the core layer. In general, all abstractions defined by the components at the model layer can be used by the upper layers.



### 5.3. Generic Layer: Informal Collaboration Support for Roomware Environments

One important goal of every software architecture is to provide generic components that are useful in many different situations and for different tasks (req. S-1). The generic layer contains components that support many work situations in roomware environments. It builds on the abstractions defined by the model layer. Figure 5-3 shows the generic components and their connections.

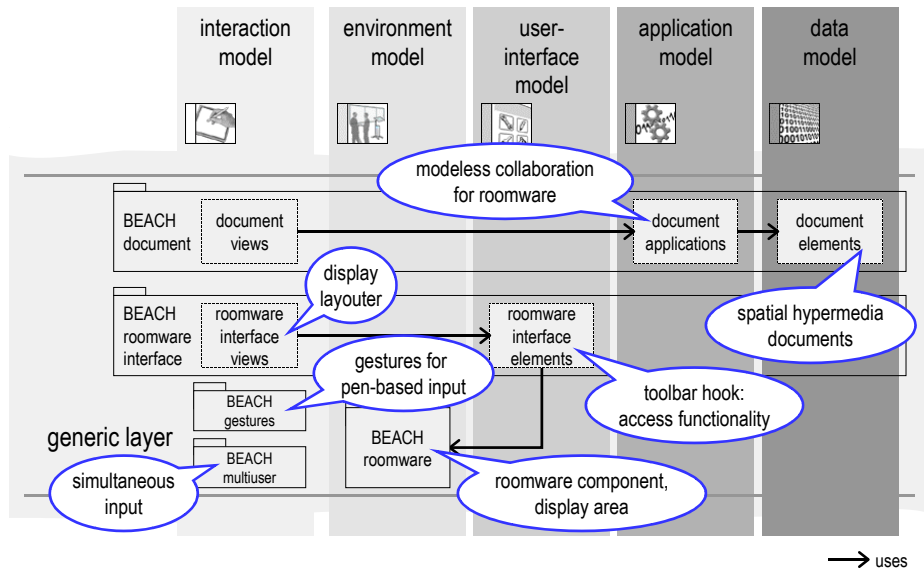


Figure 5-3. Generic components for roomware environments that are defined as part of the BEACH architecture. The dotted boxes represent the functionality within a software component that belongs to a specific model.

The generic layer defines two groups of components that are independent of each other, as shown in figure 5-3. One is concerned with the *document*, the other with *roomware components*. Both groups span several of the basic models, as typically an application model is defined to work on a specific data model, and an interaction model defines the interaction for a specific application model. As these groups are independent, they should not be seen as building sub-layers for the generic layer. They rather form entities, being next to each other at the same layer.

Both groups' interaction models rely on the *interaction style* for roomware components that support multiple users at one roomware component and gestures to ease pen interaction. This section now explains the generic components.

### 5.3.1 Interaction Style for Roomware Components

As identified in one of the requirements (req. C-3), *multiple users* may interact with one device synchronously and cooperatively. Consequently, the BEACH architecture defines a component (BEACH multuser) that extends the basic interaction capabilities to be able to handle concurrent actions. This is especially needed for roomware components such as the InteracTable, which work well for small group meetings.

Since the currently existing prototypes of roomware components offer pen-based interaction, *gesture support* is the major interaction style. The gesture component (BEACH gestures) supports assembling of strokes from low-level events, recognition of gesture-shapes, and provides an appropriate event dispatching mechanism for gesture events.

Further details about the core event handling mechanisms provided by the BEACH framework and the extensions for multiple users and gesture interaction are discussed below (section 5.5).

### 5.3.2 Roomware Components

The BEACH architecture defines two software modules to support roomware components. First, a concrete instantiation of the *environment model* (BEACHroomware) of roomware components is defined that provides a representation of roomware components, their interaction devices, sensors, environment, and relationship to other roomware components. Two abstractions are defined. The *roomware component* abstracts from the computers that are used to construct what is perceived as a single device by users. The *display area* has a similar job: it abstracts

from the displays and defines a homogeneous area that actually might be built out of several displays. This is important to ensure flexibility for applications to work with arbitrarily configured displays that might be present in the environment (req. U-3) and to dynamically update to changes in this configuration (req. U-4).

The display area is used by the second module, the *roomware user interface* component (BEACH-roomwareInterface). It defines user interface elements that are adapted to the needs of the different roomware components (called “roomware interface elements” in figure 5-3). Two user interface elements are *segments* and *overlays* that are provided as concrete implementation of the abstract concept of a visual interaction area (defined at the model layer), which is appropriate for roomware components. To be able to interact with these user interface elements, the component also defines views, controllers, and pen gestures for the interaction with these elements (denoted “roomware interface views” in figure 5-3).

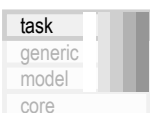
In order to be able to make the functionality that is provided by modules and services available in the user interface, the roomware interface offers a hook to add functionality to the user interface. The concept of a *toolbar* is used that can dynamically add and remove commands that can be invoked.

### 5.3.3 Generic Document Elements

The basis for documents created with BEACH is a hypermedia data model. The generic *document elements* include hand-written input (scribbles), texts, images, and links as basic objects constituting information. Workspaces are used to structure information (the equivalent of a page in other hypermedia systems). By including hand-written input and free spatial arrangement of document elements within the workspace, the infrastructure aims to support informal meetings, a major application area for roomware components (see section 2.2).

The *document applications* component defines fine-grain application models for the document elements. They add editing behavior to the document elements, and specify additional editing state, such as the cursor positions for all users currently modifying a text object. With respect to collaborative work, the application model defines no global editing modes.

The “*document pen interaction*” component defines the possibilities for pen-based interaction with the document application models. Similar to the interaction model for the roomware user interface, this defines view, controllers, and pen gestures for all application models. Note that in the BEACH architecture the interaction with a document element is always mediated by a specific application-model object.



## 5.4. Task Layer: A Platform for Roomware Applications and Extensions

The generic components that are proposed by the BEACH architecture are useful in many different situations. For some tasks, it is helpful if specific support is given (req. S-2). Therefore, the architecture has a task layer, which provides a place for modules to add further model elements and to extend the functionality of existing components.

The BEACH architecture provides support for modules at the core level. This way, modules can be plugged into BEACH that can add new functionality and services without having to change existing code. Figure 5-4 shows three modules that have been developed for BEACH, the document browser and two modules for creativity support, BEACHcreativity and PalmBEACHsupport.



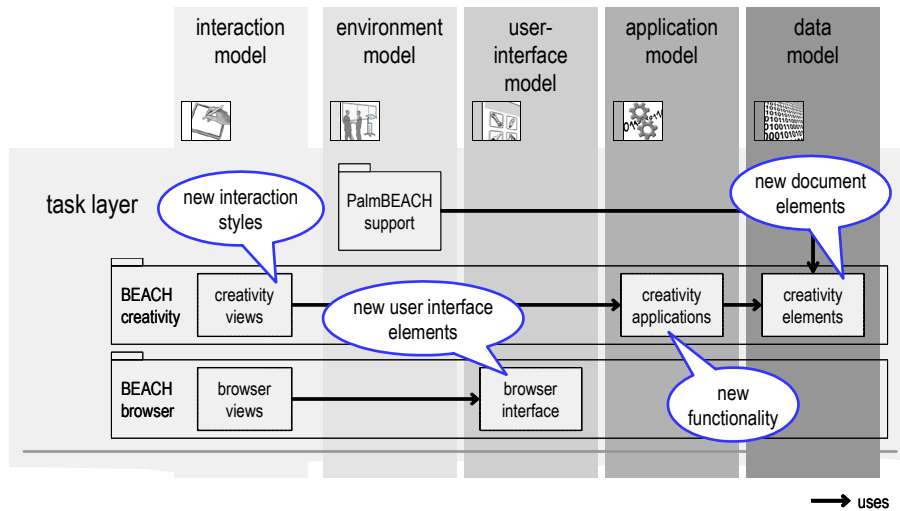


Figure 5-4. Components defined by the document browser and creativity support modules. The task layer represents the place for application-specific definitions.

The *document browser* defines a connection between the user interface and the document that is currently being viewed or edited. It can specify which part of the document is shown, also offering the possibility to navigate in the document to a different workspace. The document browser provides no visible interaction elements in BEACH. Instead, it stays in the background and reacts to commands invoked via gestures or toolbars.

At present, many tools have been implemented on top of the BEACH framework. For example, the *BEACH creativity module* provides support for creative teams to collect ideas during brainstorming sessions (Prante *et al.*, 2002).<sup>23</sup> As the roomware components allow informal interaction, they contribute to the atmosphere needed for creative meetings. To support smooth transitions, ideas generated between sessions can be collected using a PDA and transferred to a public roomware component (e.g. a DynaWall) in the next meeting. With its basic structure, this module is similar to the generic document module (fig. 5-3).

Other modules provide, e.g., extensions for context-awareness that is integrated with the data model (Flucher, 2001). The “Passage”-system enables the transport of BEACH documents using physical objects (Konomi *et al.*, 1999 and section 8.1). The ConnecTables explore the transition between individual and cooperative work by dynamically combining displays to form a homogeneous interaction area (Tandler *et al.*, 2001 and section 8.2). How auditory feedback for interactions has been added is described by Müller-Tomfelde and Steiner (2001) and in section 8.3.

### 5.5. Core Layer: Specialized Infrastructure for Roomware Components

The aim of the core level of the BEACH conceptual model is to provide functionality that will make the development of the higher levels more convenient or possible. Figure 5-5 shows the components that are provided by the core layer of the BEACH architecture. The dashed line separates two sub-layers. Model-independent components are first of all the *shared-object space* to enable synchronous access to distributed objects, *dependency detection* and automatic update, and support for *modules and services*. It is important that the basic support for sharing is provided at core level, as all other components must be able to define and access shared objects (Myers *et al.*, 2000, p. 17f).

<sup>23</sup> This and other examples are explained in chapter 9.

To enable interaction, the *event-handling* component (BEACH core events) enables event dispatching for different interaction devices. As basis for the visual interaction model, the BEACH core views component defines view objects that use the dependency mechanism to observe shared model objects and are automatically updated when the shared object's state changes.

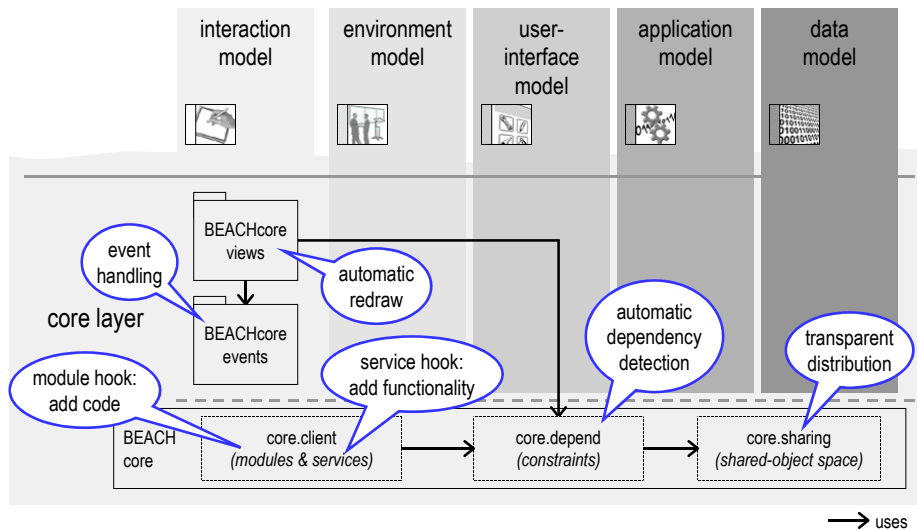


Figure 5-5. Components of the core layer. They provide a shared-object space, constraints and enable the handling of custom modules and services. On top of this, core support for visual presentation and event-based interaction is realized.

Some parts of this functionality are actually implemented by the runtime-environment of the implementation language, by the operating system, and by other toolkits. The remainder of the section explains the functionality of the components.

### 5.5.1 Module and Services Interface

To be able to plug in new components that offer new functionality without having to change existing code requires a provision of appropriate hooks already at the core layer. Hooks or hot spots have been introduced as places in a software architecture that are intended to be adaptable by applications (section 3.1).

In BEACH, modules are allowed to add:

- *components*, which may define or extend data, application, user interface, environment, or interaction models,
- *services*, which encapsulate new active behaviors that are not directly triggered by the user, but can start new threads of execution and become active due to external influences (like sensors), and
- *hooks*, which allow other modules to add new kinds of features. This is used, e.g., by the roomware interface module that defines a hook to plug in new toolbars in the user interface.

Reflection is used to check for the presence of modules. On startup, existing module objects are queried for components and services.

Services are notified on startup and shutdown of BEACH on the station they are running on. It is up to the service what actions to take. For the communication with sensors, a service might start a process to poll a sensor for values, and act if the sensed value changes. Another service could install an observer to watch for specific state changes, e.g. of the environment model.

### 5.5.2 Shared-Object Space and Distribution Architecture

In order to provide computer support for synchronous collaboration, a platform for distributed access to shared objects is needed (req. UH-2, UC-1, U-3, and U-2). As mentioned above, the BEACH conceptual model makes no assumptions about the underlying distribution architecture. This section, therefore, does not focus on the properties of different groupware frameworks and toolkits; it rather highlights the important features of the software infrastructure of a roomware environment.

To implement sharing, the BEACH architecture uses objects as basis for sharing, rather than simpler models such as tuples (Ahuja *et al.*, 1986; Johanson *et al.*, 2002a). Using objects allows for building complex data models; tuples, in contrast, are a very simple model. When heterogeneous devices and services interact, a simple model has the benefit of being easily adaptable. This is an important feature for ubiquitous computing. However, we chose to use objects for sharing, as synchronous collaboration often requires more complex data models that cannot be mapped onto tuples in a straightforward manner. Objects, instead, can be used not only to share data, but also to share application, user interface, and environment state information. This was shown above to be useful for synchronous collaboration in ubiquitous computing environments.

The BEACH architecture uses a replicated distribution architecture, as some roomware components are connected via a wireless network with a rather low bandwidth: currently 10 Mbps are shared by all connected clients (fig. 5-6). After an initial replication, only incremental changes to the shared objects must be transmitted, thus reducing necessary communication. A server synchronizes all replicas of shared objects and ensures persistency. To minimize the coordination overhead, objects are grouped in “clusters” as the atomic elements for replication.

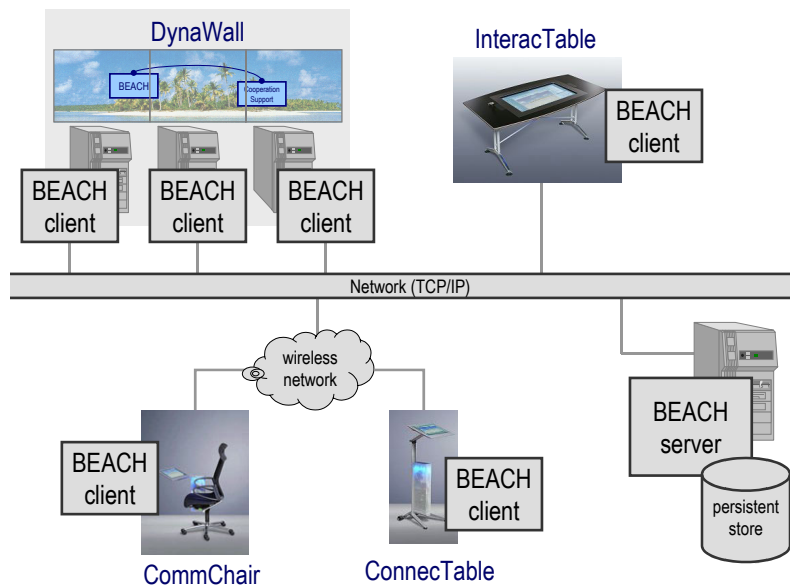


Figure 5-6. BEACH clients running on different roomware component are synchronised by a server. Mobile components communicate with the server via a wireless network.

In our implementation, it is assumed that all collaborating clients have a permanent network connection to allow for timely synchronization. However, for environments with mobile devices having no permanent network access, special support for versioning and merging would have to be provided. Transactions are used to guarantee consistency in spite of concurrent changes to objects. As a roomware or ubiquitous computing environment is highly interactive, it is important to ensure a fast response of the user interface. By committing locally and allow-

## 5. Software Architecture for Roomware Applications

ing control flow to proceed before server confirmation, *optimistic transactions* offer a significant speedup whenever conflicting actions are unlikely or harmless.

### 5.5.3 Dependency Detection and Automatic Update

As changes to shared objects can be initiated by an arbitrary computer for a variety of reasons, it is very important that mechanisms are provided to trigger updates automatically when the state of shared objects changes (req. UH-3, U-4).

Therefore, it is possible to define *computed values* for local objects. For computed values, a declarative description of the computation is used. The dependencies between computed values and attributes of shared objects are automatically detected and re-computation is triggered whenever these attributes are changed (Schuckmann *et al.*, 1996; Schümmer *et al.*, 2000). When, e.g., the attribute ‘color’ of a workspace is set to ‘blue’ while a view for this workspace is open somewhere, this view will be repainted, regardless who changed this value on which device. This is very similar to the constraints used in systems like Amulet (Myers *et al.*, 1997), but works also in a distributed setting.

Both triggering of re-computation and using shared events are techniques that allow for synchronization. Nonetheless, shared state (e.g. implemented as a shared-object space) together with a dependency or constraint mechanism enjoys several key advantages, which make it a preferable technique in our situation.

First, it is straightforward to make the state of objects persistent. This way, the overall state of tools and applications can be captured, which is helpful to recover from crashes, or to migrate applications to a different device (Coen *et al.*, 1999). If a client is crashed or migrated, it can simply be restarted (optionally on a different device) and its saved state is retrieved from the server. In the context of synchronous collaboration, sharing state helps overcoming the “late-comer” problem (i.e. clients joining an already running session), which has been discussed in the CSCW literature (ter Hofte, 1998).

Second, specifying constraints or dependencies enables a declarative programming style, which has several advantages (Myers *et al.*, 1992). For example, it can be left to the constraint system, which (and when) computed values have to be updated. This takes a huge burden from the developer and can help improve overall performance by allowing only necessary computations to be triggered. However, triggering re-computation can be seen as an automatic and implicit creation of events upon state changes.

### 5.5.4 Automatic View Update

For distributed interactive systems, it is desirable that the state of the replicated objects be consistent among all clients. Moreover, the associated interaction models should also give an up-to-date representation of their attached model objects, because otherwise the users might not realize that the shared state is actually synchronized.

To support visually oriented interaction models, the BEACH framework is capable of treating display space as a special kind of computed value (see above). The core view model provides the re-computation mechanism (also used for computed values) for the re-painting of a view’s part of the display by recording all accesses to shared model objects that occurred when painting the view. When any value accessed by a view to render the visualization is changed, the re-painting of this view is triggered automatically in order to ensure an up-to-date representation. This mechanism can be easily adapted to support different kinds of interaction models (req. H-1).

### 5.5.5 Event Handling

One characteristic of roomware environments is the presence of different interaction devices that allow different forms of interaction (req. H-1). While output devices can vary too much to provide more specific support at the core level than the automatic dependency detection

and update mechanism described above, most input devices can be abstracted as using events to provide notification of state changes. Our experience has been that it is convenient to use an event-based model, although the question has been raised if events are an appropriate interaction model for recognition-based systems (Myers *et al.*, 2000, p. 8)]. In the case of the roomware components, it is convenient to use events, as the device drivers already send events to the application.

#### Different Families of Events

To enable the integration of new interaction devices, new types of events can be added by new components. As an alternative to the standard mouse button-up or -down events, these could be event types like pen-up, pen-moved, or pen-down in the case of a pen-based system. All event types being generated by each kind of interaction device can be thought of as belonging to one *family of events*. Using this terminology, one can say that each interaction device has one associated family of events it generates.

#### High-Level Events

To support an adapted user interface for roomware components equipped with a pen, the events generated by the device drivers can first be assembled into higher-level events. As it is very intuitive to draw strokes with a pen instead of just clicking on a document, pen events can be combined as strokes. For these strokes, gesture events can be generated depending on the shape drawn with the pen (like tap, line, circle (Geißler, 1995)).

To track several concurrent event sequences, the concept of “trackers” (Demeyer, 1996; ParcPlace-Digitalk, Inc., 1995) is extended by the generic layer to function in the multi-user case (see BEACH multiuser in figure 5-3). A tracker is an object receiving events directly, without using the view hierarchy for dispatching. This is the same mechanism as is used by Myers’ multi-user interactors (Myers, 1999). BEACH is capable of handling several trackers at the same time by keeping a mapping of input device IDs to the different trackers, which will get all events from this device.

As recognition-based systems inherently have to deal with ambiguity, a hierarchical event model has been proposed (Mankoff *et al.*, 2000a; Mankoff *et al.*, 2000b). This could be easily realized on top of this event mechanism.

#### Different Event Dispatching Strategies

Different devices produce different kinds of events that might need to be handled differently. Therefore, the event-handling component allows specifying different dispatching strategies for different classes of events (Henry *et al.*, 1990).

For example, mouse or pen events, being closely connected to the visual interaction model, are normally sent to the topmost view (that wants to handle this event) at the position specified by this event.

In contrast to mouse events, which refer to a specific point, a gesture event is associated with a stroke, which could cross the bounds of multiple view objects. Therefore, a dispatcher for gesture events has been implemented as part of the generic layer (see BEACH gestures in figure 5-3) that is capable of selecting the right controller by considering all views that are “close” to the drawn shape.

### 5.6. Discussion of the BEACH architecture

This chapter presented the BEACH architecture, which applies the BEACH conceptual model using the roomware components created as part of the i-LAND project (section 2.1), in order to enable scenarios such as the ones given in section 2.2. At several points, architectural decisions have been made to tailor the flexible aspects of the conceptual model for the concrete requirements of roomware components. In this discussion, “roomware components” always

refers to the first generation of roomware components created as part of the i-LAND project. When other roomware components (or other devices) are used, it might be necessary to support different architectural properties.

As illustrated in chapter 2, the properties of roomware components that influenced the architectural decisions are:

- *visual* and *pen-based* interaction
- standard *computation capabilities* in terms of memory resources and processing speed
- a *permanent network connection* among each other
- a *slow wireless network* connection for mobile components
- the roomware components are operated inside *dedicated meeting rooms*
- the application scenarios cover *dynamic collaboration*, *informal meetings*, and *creative or design tasks*

In discussion sections, the important aspects are typeset in bold.

The roomware components focus on *visual interaction*. Therefore, the interaction model is based on the **MVC architectural style**. The user interface model uses the concept of **visual interaction areas** to specify the space on a display, which is used to display different application models.

The visual presentation of information on roomware components is augmented by a *pen-based interaction* style. Here, the separation of view and controller helps in modifying the input independently from the output. At the generic layer, **gesture recognition** is the major component for handling ambiguous pen input.

The computers currently built into the roomware components are standard PCs, or have comparable characteristics. In contrast to PDAs as target platform, the roomware components impose no special requirements on the software in terms of *computation and memory capabilities*.<sup>24</sup> This provides the chance to keep the **strict separation of basic concerns** at the architectural (and therefore also framework) level. Every client is fast enough to **render its own view** of shared objects, and has enough memory to hold replicates of all objects it is currently accessing.

As basis for distribution, a shared-object space is used. **Sharing state** in contrast to shared events enables continuous persistency. **Constraints** are used to ensure consistency of presentation with the state of shared objects. To **share objects** instead of tuples allows for representing complex models and enables sharing not only for information, but also for application, user interface, and environment state.

For the communication between roomware components, a *permanent network connection* is assumed. This is necessary to ensure the timely synchronization of replicated objects that is needed for synchronous collaboration. Using a **replicated distribution architecture** has benefits compared to a centralized architecture, as the mobile roomware components (such as the CommChair) are equipped with a wireless network that has a much lower bandwidth than a standard LAN. If replicates are used, only incremental update messages have to be sent. To speed up feedback in the user interface, **optimistic transactions** provide fast feedback, while ensuring consistency among all clients.

As the roomware components are designed to be used in *dedicated meeting rooms*, it can be assumed that a prepared infrastructure is available. This makes it possible to rely on a **server as part of the infrastructure**, to keep the primary copies of all replicated objects and to ensure

---

<sup>24</sup> An example of how the BEACH conceptual model can be applied for PDAs is described in section 9.2.

persistence. Due to the relatively static setup of some roomware components,<sup>25</sup> the focus of the BEACH architecture was not on support for mobile devices that can fluidly change environments. Hence, features such as code migration or service discovery are not included in the architecture.

When people are collaborating in a roomware environment, documents are modified concurrently on different devices. For such *dynamic collaboration situations*, it is helpful to free the developer from thinking about sending update messages manually when views have to be refreshed. Therefore, the presentation is coupled to the **shared model objects** using a **constraint mechanism**. As the shared-object space ensures the synchronization of replicated objects, the presentation is automatically refreshed, as soon as synchronization messages are received.

The roomware components aim to support an *informal meeting* style. This is reflected in the generic document elements by including **hand-written scribbles** and **free spatial arrangement** of document elements within a workspace.

Finally, *creative meetings* constitute a major application scenario. Accordingly, a sample module provides **tools to support creativity**.

The main *abstractions and hooks* defined by the BEACH architecture are summarized in figure 5-7, using the notation suggested in chapter 4. The second dimension of the BEACH model, the degree of coupling and sharing, is shown by the gray scale of the borders of the boxes. The numbers in the figure refer to the section in this thesis where this topic is discussed in detail.

The BEACH software architecture is organized by layers representing the levels of abstraction identified by the BEACH model. This allows implementing **lower abstraction levels as part of the software infrastructure**. This is not possible if the layers represent different concerns, such as data, business (i.e. application), and presentation (user interface and interaction) as commonly realized in today's architectures.

---

<sup>25</sup> Especially the DynaWall cannot be easily moved to a different room, as it is physically integrated in the meeting room. Even, to transport an InteracTable to another location is still more effort than desirable.

## 5. Software Architecture for Roomware Applications

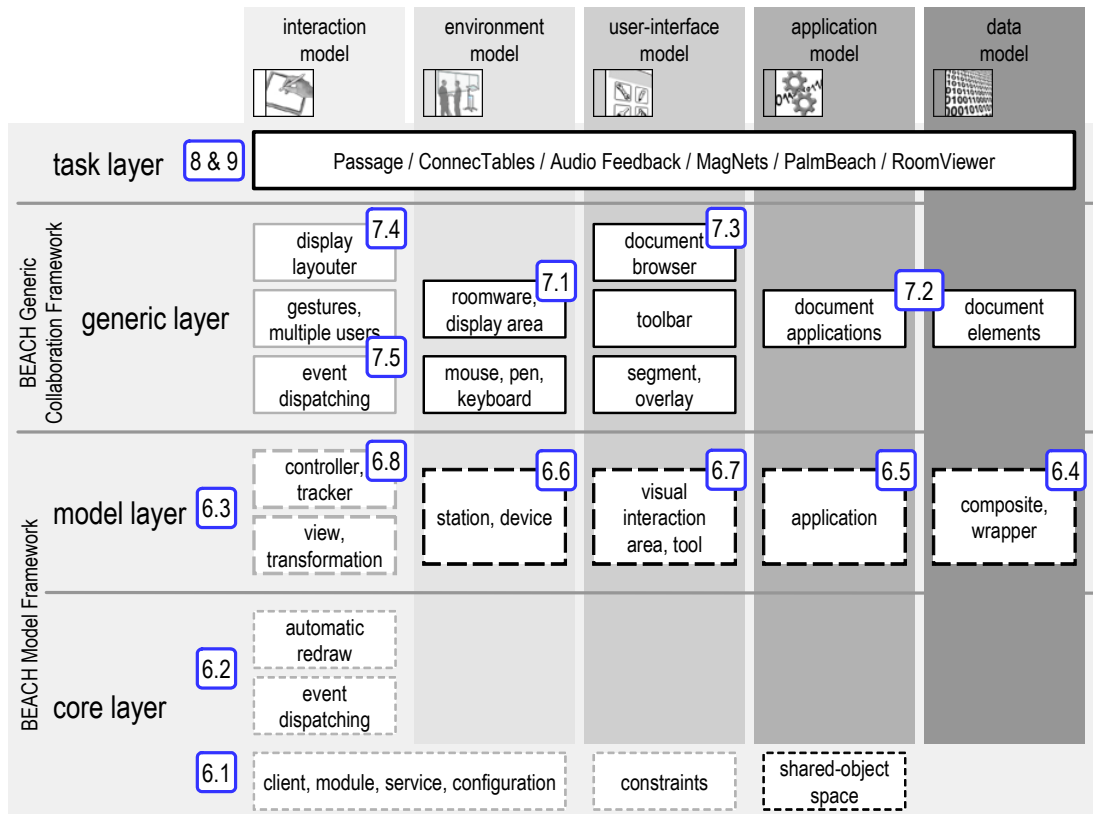


Figure 5-7. Overview of the main hooks and abstractions defined by the BEACH framework. The task layer lists the applications discussed in this thesis. The numbers refer to the section in this thesis where this topic is discussed.

↓ Next chapters

In the next two chapters, it is explained how the BEACH architecture is implemented as two object-oriented software frameworks. Besides conceptual model and software architecture, the framework constitutes the third level (see section 1.4) at which this dissertation is contributing.



## 6. Software Infrastructure for Roomware Environments

---

This chapter describes the design of the BEACH Model framework, which implements the core and the model layer of the BEACH architecture. It serves as a proof-of-concept to show how the conceptual model and architecture can be successfully applied. The core layer implements platform-dependent functionality. The model layer provides abstractions that enable extensibility. Hooks are defined for the aspects that are designed to be extended by upper layers.

---

This chapter describes the design and important implementation details of the BEACH Model framework. It implements the *core and model layer* of the BEACH architecture presented in the previous chapter. In this way, it can be used as the basis for implementing both generic and specialized components for roomware applications. The BEACH Model framework (and therefore necessarily BEACH applications as well) are implemented using VisualWorks Smalltalk (Cincom, 2002). The BEACH framework was first presented in (Tandler, 2001c).

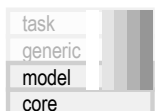
The description of the design is organized bottom-up, starting with the model-independent core components. The following chapter focuses on the generic layer. The design presented here is at some parts abstracted from the “real” design of the BEACH framework to contain only the essential parts and thus make it more understandable. The “real” design had to consider more constraints from the implementation platform that are not relevant to explain the application of the BEACH conceptual model and the BEACH architecture.

### 6.1. Design of the Core Layer

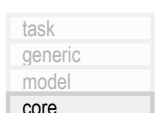
This section explains the design of the components specified for the core layer of the BEACH architecture. The architecture was presented in section 5.5 and illustrated in figure 5-5.

The core level of the BEACH architecture defines six low-level abstractions that are explained in this section:

- The *client* represents an instance of BEACH running on a device (section 6.1.1).
- A *module* allows plugging in of new code into the BEACH architecture (section 6.1.2).



↓ Chapter outline



- A *service* encapsulates functionality that can be added by modules (section 6.1.1).
- The *configuration* provides a common place to define parameters for modules and services and a single interface to modify all available parameters (section 6.1.1).
- The *shared-object space* defines the basic abstraction for implementing transparent distribution (section 6.1.3).
- *Constraints* are used for automatic dependency detection and re-computation (section 6.1.3).

### 6.1.1 Beach Client: Modules, Services, and Configuration

The BEACH client is responsible for handling services, configurations, and modules. In addition, it also is responsible for initializing the shared-object space and establishing the connection to a server, as described below. Modules and Services represent two essential hooks that are defined at the core level. *Modules* allow the integration of arbitrary software components, while *services* encapsulate functionality that can be turned on and off. The *configuration* takes care of handling the selection of values for various hooks and parameters.

#### Service Management

A *service* encapsulates any kind of functionality that can be started and stopped.<sup>26</sup> Many services like to act on startup and shutdown of the BEACH client. It is up to the service what actions to take. A service can be passively used by other components, or it can actively start its own thread of control. A couple of subclasses are defined to support different kinds of services (fig. 6-1).

The `SingletonService` can be used for services that have exactly one instance. (The name is derived from the “Singleton” design pattern (Gamma *et al.*, 1995).) Its single instance can be accessed by the class method `current`.

The `ProcessService` adds the ability to spawn a process that remains under the control of the service. When the service is shutdown, the forked process will also be terminated if it is still running.

While all service objects are local objects belonging to the local client, the `SharedService` can be attached to a shared object that is used to store its state. This way, it is possible to change the service’s state remotely by simply modifying the shared object at another client. This, e.g., is used by the environment model to implement the startup and shutdown of the devices connected to the station (section 6.6).

---

<sup>26</sup> Examples of how the service hook has been used are given in sections 6.6 (device service), 7.4 (display application), 8.1 (sensor management), 8.2 (ConnecTables), and 9.2 (PalmBeach).

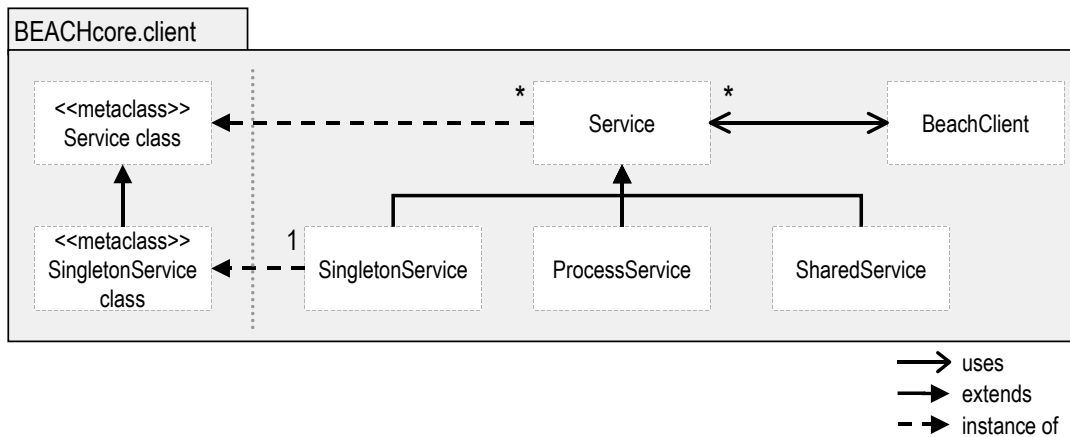


Figure 6-1. Different kinds of services

To influence the order in which services are started, each service can specify *prerequisite services* that have to be running in order to start this service. If a service is shut down, it is ensured that all dependent services are shut down before.

### Configuration

The `BeachClient` also handles the configuration of services and modules by offering an interface to define and access configuration entries, grouped by categories. The configuration object is the single point to control the configuration of the overall system, as proposed in (Demeyer, 1996; Demeyer *et al.*, 1997), and is similar to MPACC's coordinator (see page 46). At startup, all defined configuration entries are set to the values provided in the client's configuration file.

This has been implemented to have a simple mechanism to inform each client about the (static) context in which it is running. It is implemented in such a way that nearly all configuration values can be changed dynamically. One exception is the hostname the server is running on; this value is only read at startup and cannot be modified afterwards, as switching to a different server would require shutdown of the client first.<sup>27</sup>

### 6.1.2 Modules and Hooks for Extensions

Using a reflective programming language like Smalltalk for the implementation of a framework, one can benefit using reflection (see page 37) to detect modules and allow a wide range of adaptations. The interface for modules is implemented by a single class, `Module`. New modules are defined by simply including a new subclass of class `Module` in a new software component. Due to the ability of Smalltalk's class objects to query existing subclasses, `Module` can easily check for the presence of modules—just by querying all its subclasses.<sup>28</sup>

To be able to make any initialization, class `Module` is designed to extend class `SingletonService`. This allows treating modules as a specialized form of services if they have to perform startup or cleanup functions. Using the service's prerequisites mechanism, modules can thus also specify other services—or modules—that should be started before.

In order to determine the initial set of services, the `BeachClient` checks the existence of loaded modules on startup. Using Smalltalk's meta-object protocol (see page 37), all subclasses of

<sup>27</sup> The current implementation actually restarts the client automatically if the server is changed. This causes no inconvenience as the complete state of each client is stored persistently by the server, and switching to another server and back to the previous resumes exactly where stopped.

<sup>28</sup> Some examples of modules that have been created are presented in sections 6.6 (environment model module), 7.1.1 (roomware module), and 8.2 (ConnectTables).

class `Module` are asked for services that should be automatically started together with the BEACH client.

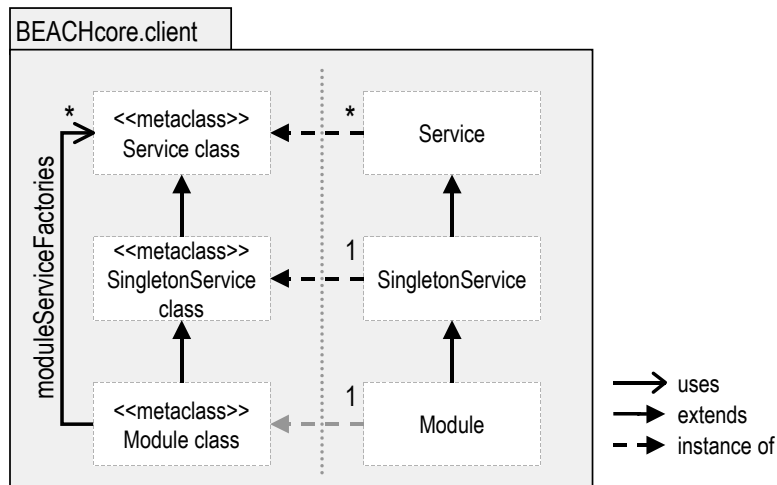


Figure 6-2. Classes `Module` and `Service`.

The remainder of this section presents an example of how modules can use the object-oriented techniques for extensions (discussed in section 3.1, page 37), to define hooks to be used by new modules.

Using these extension mechanisms, the technique of querying existing subclasses of class `Module` can be used by higher-level components to inquire the existence of present functionality. For example, the user interface model (component `BEACHuimodel.toolbars`) uses the modules to check for additional toolbars (see section 7.3.4).

Example 6-1:  
Toolbar hook

Besides sub-typing and inheritance, a very powerful extension mechanism is the possibility to add methods and/or attributes to a class defined in another module. If the user interface model module wants to allow modules to add new toolbars, an easy implementation would be to send a `toolbars` message to every loaded module. Each module then has the possibility to return a collection of toolbars to be included in the user interface. The problem that occurs is that not all modules necessarily understand the new `toolbars` message, which results in runtime (or compile-time) errors if the message is sent to these modules.

One approach to solve this problem is to use reflective features available in the implementation language as described above. Then, every module would be asked if it responds to the `toolbars` message, before actually sending it. While this approach works technically well, it decreases the maintainability of the software system, as all software development tools analyzing message passing between classes have a hard time in examining these messages.

A better solution is to use an extension to the module class. The user interface model module adds the `toolbars` message to the `Module` metaclass—which is actually defined in the `BEACHcore.client` module (fig. 6-3). Every subclass that uses the `uimodel.toolbars` module can then redefine the `toolbars` method in order to return new toolbars.

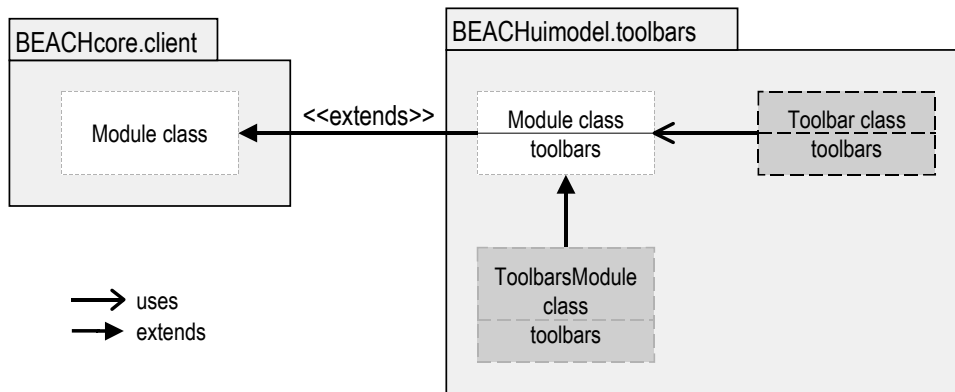


Figure 6-3. Example how to use the extension mechanism to add a `toolbars` method to the `Module` metaclass.

### 6.1.3 A “BEACH at the COAST”: Shared-Object Space and Constraints

As discussed in section 5.5.2, a shared-object space is the major core functionality for cooperative ubiquitous computing applications. In section 3.4, several existing toolkits and frameworks for CSCW applications have been presented. Each of these systems has advantages and disadvantages. In practice, it depends therefore on the target applications and the actual environment which framework should be chosen (Urnes and Graham, 1999). As the shared-object space that constitutes the basis for the BEACH framework, the open-source framework COAST has been selected (COAST, 2003; Schümmer *et al.*, 2000; Schuckmann *et al.*, 1996). The reason was, on the one hand, because it has the properties needed in a roomware environment, on the other hand, because it has been developed and used in previous projects<sup>29</sup> at Fraunhofer IPSI. Using COAST, the available competence could be reused and flexible support was possible. This section therefore discusses the properties of COAST that are relevant for the implementation of shared-object spaces in ubiquitous computing environments. The COAST framework itself has not been developed as part of this dissertation. Instead, the new application domain introduced in this thesis had an impact on the development of the framework. Figure 6-4 illustrates the relationship between BEACH and COAST.

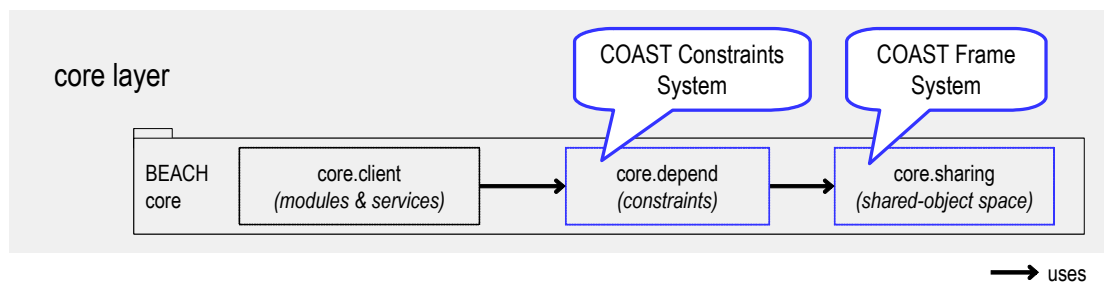


Figure 6-4. Relationship between BEACH and COAST. The COAST framework is used to implement the shared-object space and the constraint system.

<sup>29</sup> The first version of COAST was implemented as part of SEPIA (Haake and Wilson, 1992). It was extracted as a framework for DOLPHIN system (Streitz *et al.*, 1994), and has been used in several other projects in the meantime.

### COAST Architecture Overview

COAST defines a simple, platform-independent, and efficient protocol for synchronizing replicated objects, quite similar to the one defined by XWeb (see page 54). In order to easily use this protocol, the COAST framework has been implemented for applications written in VisualWorks Smalltalk (Cincom, 2002; ParcPlace-Digitalk, Inc., 1995). It takes care of handling replicated objects and encapsulates all communication with the COAST server.

In fact, the COAST framework can be divided into three main parts for both the client and the server side. The *frame system* constitutes the object model for the shared-object space; the *replication and transaction management* takes care of the replication and synchronization and ensures consistency. These parts implement the shared-object space for the BEACH framework, for both clients and server. The third part for the client is the *dependency detection*, which is used to keep output interaction models (like view objects) up-to-date. The *persistence* component on the server side handles the storage and retrieval of all shared objects. By separating replication and frame system, the replication is transparent to the frame system and to software using the frame system. These components are explained in this section. For more detailed description of COAST, please refer to (COAST, 2000a). A more extensive overview over the current version of the COAST framework is given in (COAST, 2000b).

### Frame System

Frame systems have been used in the construction of interactive software for several years now. Frames were originally used within the domain of artificial intelligence for knowledge representation. In contrast to objects that model their relationships among each other using simple references, frames use slots to express their relationships. In addition to storing values, slots are capable of carrying supplementary semantics such as bi-directional references (called “inversion”) and cardinality constraints.

Slots were introduced in object-oriented programming languages by CLOS, the Common Lisp Object System (Keene, 1989). As mentioned in section 3.2, the Amulet environment uses slots for the construction of interactive systems (Myers *et al.*, 1997; Myers *et al.*, 1998a).

The frame system monitors access to frames and slots to provide the declared properties or check constraints. That means the frame system has full control over the state of all frames. Based on this fact, the frame system can provide additional generic facilities such as transactions or persistent storage that can be used for any user-defined frame.

In COAST, slots can hold not only a single value. Instead, there are slot types providing basic abstract data types like Set, Ordered Collection, or Dictionary (Map/Hash). So called “inverting slots” use constraints to ensure a bi-directional relationship between objects. This way, it can be ensured that two objects always reference each other; if one reference is changed, the other one will be adjusted automatically.

In COAST’s frame system, every frame has a *type* attribute. The COAST framework uses this attribute when instantiating objects for a frame. It uses the type attribute to look for a matching class, and instantiates an object of this class, if the class is present. If no class is found, the frame is instantiated as an object of class `CoastUniversalFrame`. This approach is very comfortable when using object-oriented programming languages. However, it would be possible to always use a single frame class if desired, or, if more appropriate for different platforms or different languages. Being a white-box framework, COAST provides the class `CoastModel` as basic superclass for all shared objects.

### Replication and Consistency

The replication management component of COAST handles the remote access to shared objects. Unlike systems aiming at distributed processing (such as CORBA (Object Management Group)), in most systems supporting synchronous collaboration, the behavioral part of each

object (i.e. methods) is always available at each site, instead of using proxy objects. This allows much faster feedback of the user interface, which is crucial for interactive systems.

Instead, the state of shared objects is replicated. COAST uses a server-based architecture (fig. 5-6). The server always has the *primary copy* of replicated objects, while the clients get *secondary copies*. This implies that client state might be overwritten (or rolled-back, see below) in cases of conflicts. One benefit of a replicated architecture is that after an initial replication, only incremental update messages have to be transmitted. This requires much less network bandwidth, which is especially important for those roomware components connected using wireless technology that is currently much slower compared to standard Ethernet.

#### *Partial replication by Clusters*

Early versions of COAST always replicated the complete shared-object space to every client. It turned out that this was infeasible in practice when object structures got too large. Therefore, now all shared objects are grouped into *clusters* that represent the atomic entities for replication. This way, the necessary management overhead for partial replication is reduced, and performance can be gained if the objects are grouped together that are most likely used together. Good candidates are, e.g., all objects on one document page or workspace. If an object is accessed at a machine that has no replicate of this object yet, COAST transparently loads the necessary cluster.

#### *Consistency Control and Commutative Operations*

While working in roomware environments, concurrent interaction can occur at different roomware components. This might result in conflicts, if, e.g., one property of an object is modified by more than one user at the same time. To ensure consistency between all clients and the server, transactions are used. Transactions guarantee that all changes that occur within a transaction are accepted—or all changes are denied and correctly rolled-back, i.e. restored to the state before the transaction was executed.

If transactions arrive at the server, the server checks whether the state that the transaction was based-on is still valid, or if another transaction has already been committed that has modified some object's state the new transaction has used. If two transactions are independent of the order they are executed, they are said to be *commutative*. COAST uses a very fine-grained model to check for commutativity of transactions. This way, the application designer can gain a high degree of possible parallel interactions without losing the power of an automatic concurrency control.

COAST defines commutative operations on a per-slot basis. For every type of slot, all commutative operations are defined. A 'read' and a 'write' access to a slot are never commutative, because the read would result in returning different values when executed before or after the write. In contrast, two 'add' accesses to a slot of type 'set' are commutative, as after the execution both values are part of the set independently of the order they have been added. This allows a very fine-grained detection of conflicts, in contrast to locking whole objects or even collection of objects.

#### *Persistency*

The COAST server provides persistency for all shared objects. To be able to retrieve shared objects when a client has been shut down and is started again, a *name* can be assigned to some key-objects. This name can be used if a client is started to check for the existence of these objects. This is similar to the naming service provided in GaiaOS (see page 45).

Every change of a shared object's state is stored automatically by the server. Therefore, a crash of a client is not very harmful, as it will retrieve all its state from the server when started again.

#### *Constraint System: Automatic Dependency Detection and Updating*

For distributed interactive systems, it is not only desirable that the state of the replicated objects is consistent among all clients, but associated interaction models should also give an up-

to-date representation of their attached model objects, because otherwise the users would not realize that the shared state is actually synchronized. This part of the core layer of the BEACH framework can thus be associated with the interaction model.

### *Computed or Virtual Slots*

To ensure the up-to-date presentation, COAST provides a one-way constraint system that is used to couple local (interaction) models to shared objects. Local objects can have *computed* or *virtual slots* that cannot be modified directly like other slots. Instead, they have an associated specification *how to compute* the slot value. The specification is simply a code fragment (implemented as a Smalltalk block) that returns the value for the slot.

When a virtual slot is accessed the first time, COAST executes its associated computation specification and caches the returned result. Moreover, it registers all accesses to other slots that occur during the computation of the slot. The accessed slots are used to automatically build up dependencies between the slots. This way, all depended virtual slots can be invalidated if at least one of the recorded slots changes its value.

In COAST, the class `CoastVirtualFrame` is used as base class for all frames being able of having computed slots.

### *Lazy and Eager Slots*

To avoid unnecessary re-computations, virtual slots by default only re-compute their value when they are accessed the next time (and—of course—if their cached value has been invalidated). In addition, it is possible to declare a virtual slot as *eager*, which means that it is re-computed immediately at the end of the current transaction. The default behavior of virtual slots is called *lazy*. The concept of lazy and eager computation is common in many interpreted (mainly functional) programming languages like CLOS (Keene, 1989), Common Lisp (Steele, 1990), or ML (Leroy, 2001).

### *Display Updates*

To support visually oriented interaction models, COAST is capable of treating display space as a special kind of virtual slot. COAST's view model provides the same re-computation mechanism for the re-painting of a view's part of the display by recording all slot accesses that occurred when painting the view. In order to always have an up-to-date representation COAST eagerly triggers the re-painting of all invalidated views at the end of every transaction. However, this mechanism can be easily adapted to support different kinds of interaction models (req. H-1) as shown in section 8.3 below.

## Speed-up Possibilities for Distributed Interactive Systems

In COAST, all accesses to shared-object state must occur within a transaction to ensure consistency of all clients. As a roomware environment is highly interactive, it is important to ensure a fast response of the user interface (Coen *et al.*, 1999). Therefore, COAST offers several possibilities to speed up interaction, namely display transactions, event handling support, and optimistic transactions.

An essential issue is the execution of transactions. Avoiding delays waiting for the server's commit, *optimistic transactions* offer a significant speedup whenever conflicting actions are unlikely or harmless. In case of a rollback, all transactions that rely on the rolled-back transaction are also rolled-back. If a client has to rely on a transaction, it has to use a pessimistic transaction, which blocks the client's active process until the commit (or cancel) is received.

To update local objects like views, so-called *display transactions* can be used, which inhibit modifying access to shared objects. Therefore, they can be executed completely on the client's site, which is much faster as no messages have to be sent to the server.

When accessing local objects, the use of transactions may be discarded if speed is particularly crucial and consistency is not the main focus or conflicts are very unlikely. This has been introduced for faster event handling. If, e.g., mouse events have to be dispatched to the right view's controller, the bounds of many views have to be checked. The views' bounds are nor-



mally stored as part of the computed local-object state (see below), which would result in an up-to-date check for every access of every slot. However, it is very likely that the currently cached value is up-to-date, as mouse interactions normally occur *after* a view has been displayed—and during display transactions all computed slots that are accessed are updated. This speed-up illustrates that optimization often implies an increased responsibility of software developers. If no transaction is used, the framework cannot automatically ensure the consistency of computed slot values, thus forcing the developer to take care that all slots used while event dispatching are guaranteed to be computed during display update.

In COAST, views are never directly notified about changes by their controllers, as done in other systems, such as Smalltalk’s original MVC implementation. Instead, views use the dependency mechanism of COAST, registering the dependencies between the visualization and each slot used when drawing. Each slot notifies its observers about write accesses to this slot. This leads to a fine granularity for update messages that helps avoid performance problems.

For further optimization, COAST accumulates all invalidation messages (i.e. notifications about slot changes) to avoid the same re-computation being triggered twice. This is a technique that all window systems do when handling re-draw messages.

These mechanisms ensure that COAST applications are fast enough to run on standard desktop PCs. When starting developing BEACH, we used 200 MHz Pentium CPUs, yielding a reasonable speed for interaction. Concluding, we experienced that the overhead introduced by COAST’s slot dependency system is really worth it. In return, one gains a significant reduction of implementation effort and avoided bugs, because developers do not have to worry about sending update notifications manually and because consistency is guaranteed.

#### BEACH Shared and Computed Models

BEACH introduces a base class for all shared objects. The class `BeachModel` itself is a sub-class of class `CoastModel`, which is defined by the COAST framework. In addition to `CoastModel`, it has some minor enhancements, such as the ability to support shared prototype objects, to allow a prototype-instance style of programming if desired. The core property of the prototype-instance concept is delegation (Ungar and Smith, 1987). Using delegation, active properties can be realized, as described in (Dourish *et al.*, 2000), but this is not a main issue of this thesis.

Module `BEACHcore.depend` defines the base class for all computed models, `BeachComputedModel`. It uses the dependency mechanism defined by COAST to implement models that can observe other—especially shared—models. Computed models can define computed slots, which are re-computed whenever an observed model is changed (either eagerly or lazily). Computed models are always local objects to avoid conflicts with triggering re-computation.

## 6.2. Design of the Core Interaction Model Support

The core layer contains two components that are included to form the basis for interaction models.

- The `BEACHcore.events` component encapsulates the interface to the operating system’s event handling API and provides a hook to use different *event dispatching strategies*.
- The `BEACHcore.views` component offers a set of base classes for the development of a *visual-based interaction model* that use the constraint mechanism for automatic re-display.

### 6.2.1 Event Handling

The `BEACHcore.events` component (figure 6-5) provides a low-level infrastructure for handling events that are either sent to the application by the operating system or generated by a part of the application itself. (An example of application-generated events is given in section 7.5.2, page 144.)

task
generic
model
core

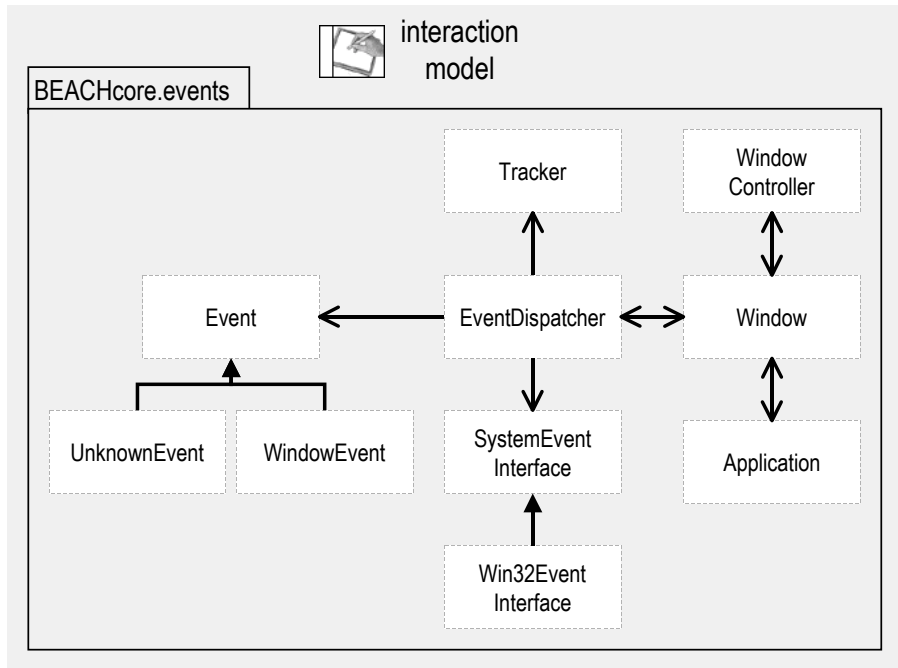


Figure 6-5. The core level support for the interaction model defines classes for event handling.

The class `SystemEventInterface` is an abstraction of the specific system's API. At runtime, a concrete subclass like `Win32EventInterface` is instantiated depending on the current platform. The class `EventDispatcher` retrieves events from the `SystemEventInterface` and creates event objects from a matching subclass of `Event`. If no matching subclass can be found, an instance of `UnknownEvent` is created. Class `WindowEvent` is the base class for events concerning the window, like window repaint requests.

The event objects are dispatched to an appropriate handler for this event—depending on the kind of event and its properties (see section 5.5.5). The handler could be a `Tracker` that is currently grabbing events of some kind. To select a handler, different event dispatching *strategies* can be used. For example, most events that refer to a specific position on the screen are sent to the `WindowController` of the `Window` at this position. Other event dispatching strategies are presented in sections 6.8.1 (mouse events) and 7.5 (keyboard and pen gesture events).

Although not directly concerned with event dispatching, `BEACHcore.events` includes the class `Application` that builds the low-level abstraction of a top-level application model that is associated with every window (ParcPlace-Digitalk, Inc., 1995). This is used in section 6.3 as the base for visual-based interaction.

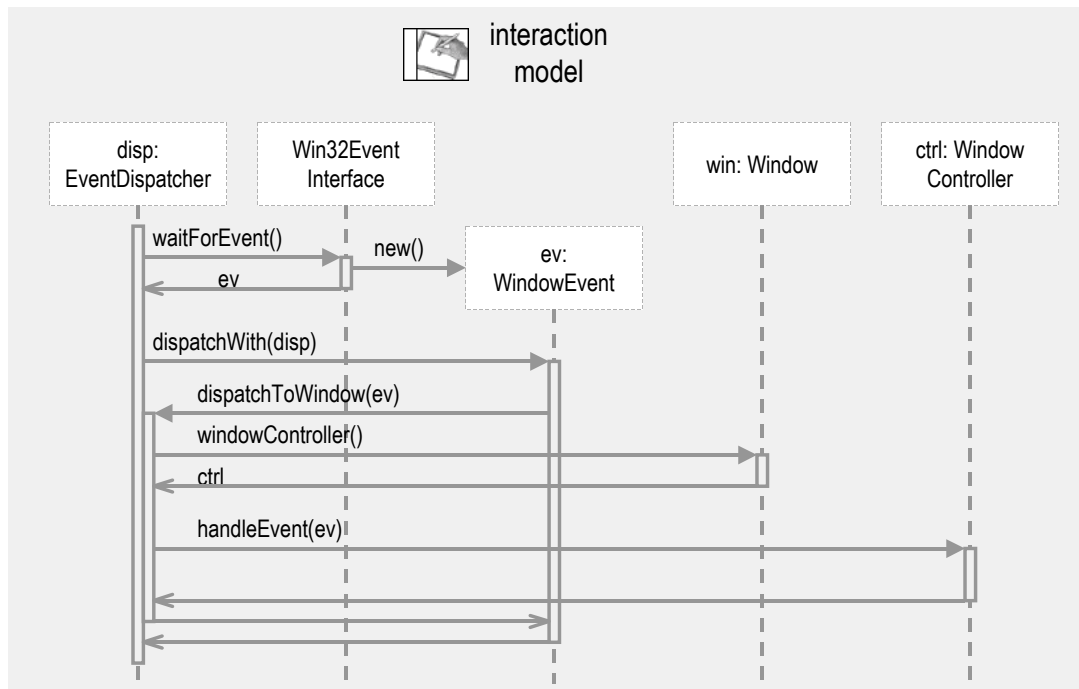


Figure 6-6. Event dispatching strategy for window events. Window events are always dispatched to the window controller.

Example 6-2: Event dispatching strategy

Figure 6-6 shows a sequence diagram illustrating the dispatching strategy used for window events. When the event dispatcher receives a new event, it uses double-dispatch (Gamma et al., 1995) (*dispatchWith(dispatch)*, *dispatchToWindow(ev)* in the diagram) to determine an appropriate dispatching strategy for the event. Dispatch strategies are implemented as methods of class *EventDispatcher*. The dispatch strategy method first detects a controller to handle the event. Then, it sends *handleEvent(ev)* to this controller.

### 6.2.2 Views

The *BEACHcore.views* component extends the low-level support for visual-based applications by defining abstractions to organize and visualize the available display space (figure 6-7). In fact, the many classes contained in *BEACHcore.views* are provided by the *VisualWorks* application framework and the *COAST* framework.

Every *Window* contains a top-level *VisualComponent*. A *VisualComponent* mainly manages one part of the window's display space, defined by its *bounds*, which can be painted using the method *displayOn()*. The *displayOn()* method is given an instance of the window's *GraphicsContext* as an argument. *GraphicsContext* provides a device-independent low-level interface to draw geometric objects.

*CoastAutomaticVisualPart* extends *VisualComponent* with support for the dependency mechanism provided at the core level (see section 6.1.3). A *CoastAutomaticVisualPart* can contain computed slots. More important, it uses the dependency mechanism for re-painting its visual representation (using *composeDisplayOn()* instead of *displayOn()*). This is an example of how constraints can be used to ensure a presentation that is always consistent with its associated models. In section 8.3 it is discussed how the same mechanism can be used to give an auditory presentation.

Finally, *CoastView* has an attached model that is visualized by the view, and a controller that reacts to events by manipulating the model.

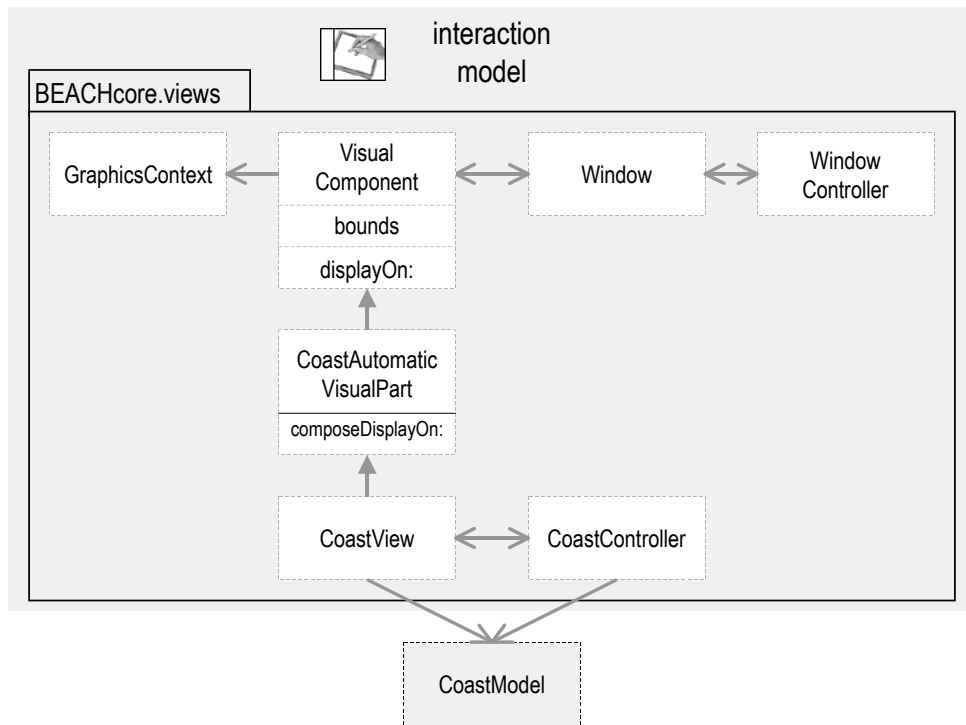


Figure 6-7. The base classes defined at the core level for the interaction model in BEACHcore.views. Class *CoastAutomaticVisualPart* uses COAST's dependency mechanism for redraw and it supports the definition of custom computed slots.



### 6.3. Design of the Model Layer

The aim of the model layer is to provide an abstract implementation of the five basic concerns (see section 4.3) to be used as the device- and application-independent basis for the implementation of the higher layers. Three abstractions are used in the design for all basic concerns:

- *Composites* use the composite pattern (Gamma *et al.*, 1995) as an abstract interface to part-whole hierarchies (see fig. 6-9).
- *Wrappers* can add behavior using the decorator pattern (Gamma *et al.*, 1995). *Relation wrappers* describe the relationship between objects (see fig. 6-9, 6-10).
- *Applications* are abstractions for the top-level models that represent each concern as a whole for a software application (see fig. 6-5, 6-13, and 6-14).

The BEACH framework implements the basic concerns by defining abstract base classes (fig. 6-8) and additional helper classes.<sup>30</sup> The base classes of environment, user interface, application, and data model ensure that all instances of their subclasses are part of the shared-object space.

For the visual interaction, the base classes for views and controllers are defined (Krasner and Pope, 1988a). These are always local objects, as they are computed by every device independently, depending on its local context. This way, views can use different scale factors depending on their display size, or show a different part of a common workspace, as in the case of the DynaWall (shown in fig. 2-1).

<sup>30</sup> Figure 6-8 also shows an interesting observation: Lower-level layers are usually drawn *below* higher-level layers. (That is where the name comes from.) For class diagrams, in contrast, it is common to place low-level abstractions (i.e. base classes) *above* their specialization (sub-classes).

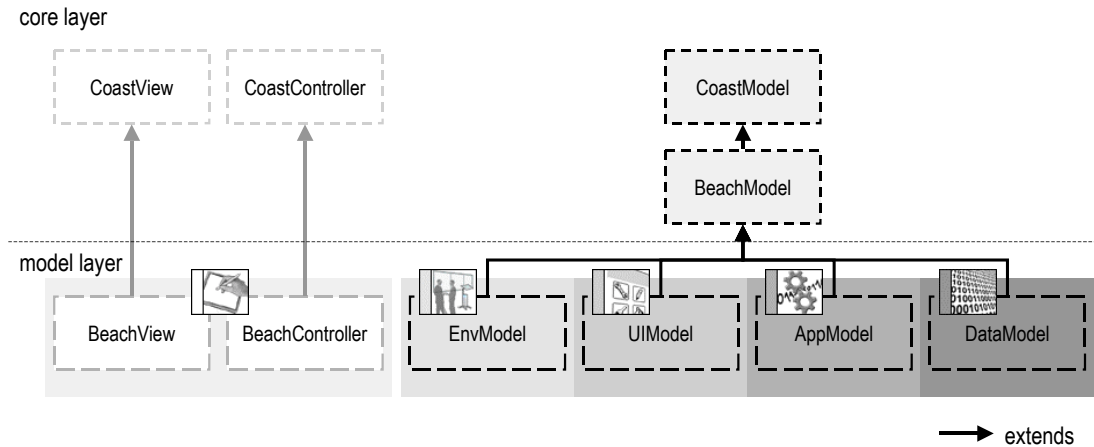


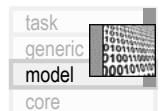
Figure 6-8. Base classes for all shared models. The graphical notation introduced for the BEACH model (see fig. 4-2) is used in class diagrams to ease the mapping of classes to the model’s dimensions. The grayscale of the background indicates the basic concern the class belongs to, the grayscale of the border the degree of coupling, and the border style the level of abstraction.

One abstraction that covers all basic concerns is the *application*. For a software application, the “application” abstraction represents the top-level model for each concern. Applications have to coordinate which sub-models are used. In addition, the applications specify the relationship among the concerns, e.g. the user interface application defines which application model to use, and the interaction application chooses the user interface application.

In sections 7.3 and 7.4, an example of how the application abstraction can be used is given, taking the example of visual interaction. In section 8.3, a model for audio feedback is defined that creates a different interaction style for the same user interface application.

The following sections of this chapter explain the design of the abstractions defined for the five basic concerns.

↓ Chapter outline



## 6.4. Design of the Data Model

The data model supports very simple, hierarchical document structures, as an elaborate data model is not the focus of this thesis (see 3.5.2). It defines base classes for the containment hierarchy and wrappers to be inserted between document elements<sup>31</sup>.

Figure 6-9 shows the classes defined by BEACHdatamodel. BEACHdatamodel defines class `DataModel` as a base class for all shared data model classes. The classes `DataModel`, `ContainerData`, and `AtomicData` define the basic structure of hierarchical documents, following the composite pattern (Gamma *et al.*, 1995). This way, clients can treat individual objects and compositions of objects uniformly. In addition, as composition is a very common structure, defining a simple protocol for accessing hierarchies enables the reuse of all operations that are performed on hierarchies.

Using the composite pattern to represent a containment hierarchy has the benefit of providing an abstract interface for navigation in the structure, while keeping the possibility to add further document elements transparently. This ensures extensibility (req. S-2) of the data model, as new types of document elements can be placed in the document hierarchy.

<sup>31</sup> The term document “element” is used to avoid confusion with (software) “component” or “object” in general.



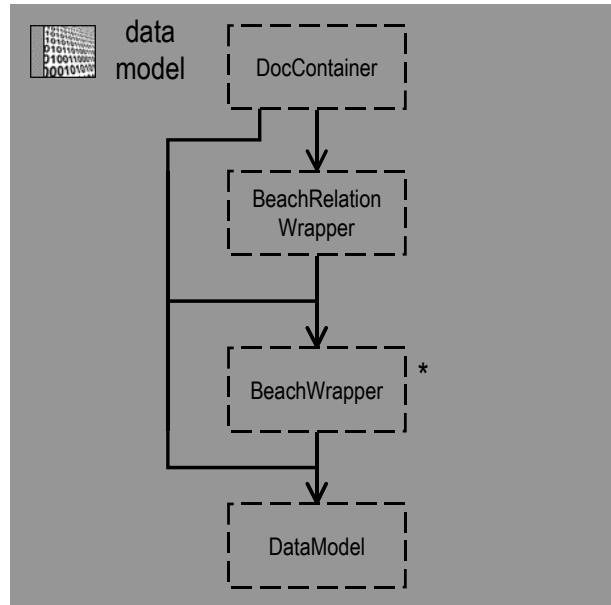
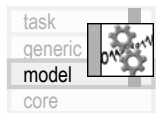


Figure 6-10. An arbitrary number of document *wrapper objects* can be added around a document element. Additionally, a *relation wrapper* can be inserted between a container and its sub-components to describe the properties of their relationship.



## 6.5. Design of the Application Model

Application models are used to describe the application behavior such as manipulation of document objects (see 4.3.2).

In general, two different classes can be distinguished, depending on their relationship to data models. The base class for all application models is class `AppModel`. Yet, some application models have a *direct* association to a data model to which they add editing capabilities. These are subclasses of class `DataApp` (fig. 6-11). Having a separate application model object for every data object leads to a fine-grained structure of application models that increases the reusability of the models (req. S-2), and also allows a fine-grained model for the degree of coupling (req. C-2). The issue of coupling modes is discussed in sections 4.3.2 and 4.4.2.

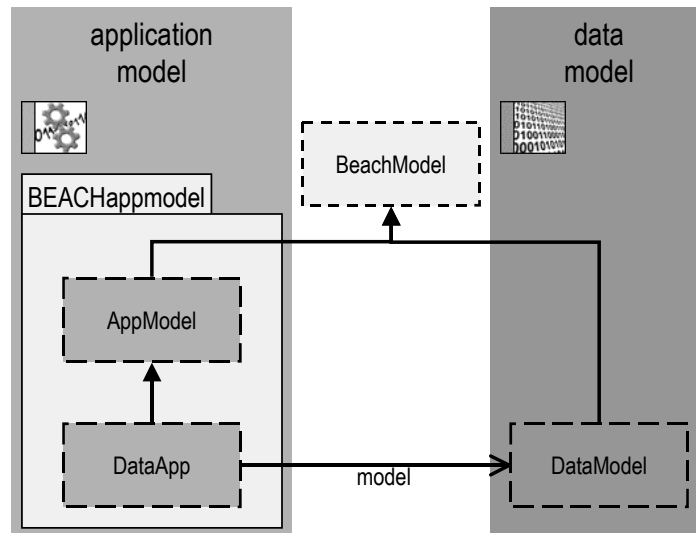
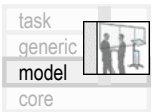


Figure 6-11. Classes AppModel, DataApp, and DataModel

To be able to define a fine-grained structure of application models that are likely to be reusable, BEACHAppmodel defines a set of base classes that reflect the structure of the data model.

In order to reflect the structure of the current document, the default behavior of application model classes is to keep the structure of the document, i.e. for every data object an application model object is created. Therefore, all subclasses of DocContainerApp observe (Gamma *et al.*, 1995) their associated container data model object and will add or remove application model objects, whenever document elements are added to or removed from the document container object. This is related to the “tree maintenance links” in ALV (Hill *et al.*, 1994).



## 6.6. Design of the Environment Model

In contrast to the models described so far that only define abstract classes; the environment model goes one step further by also providing concrete classes. Concrete classes are defined for the station, the device service, and the environment model module (fig. 6-12). The environment model defines two main abstractions:

- The *station* refers to a computer running a BEACH client.
- A *device* is anything that can be attached to a station and be used for interaction.

The module BEACHenvmodel is an example of how the module hook that is provided at the core layer (see section 6.1.2) can be applied. The module contains class EnvModelModule as a subclass of class Module. As all defined subclasses of class Module are detected using Smalltalk’s meta-object protocol, class EnvModelModule can define services to manage the station and its attached devices.

A computer running a BEACH client is called *station*. Stations can have attached devices, which can be both explicit interaction devices—like displays, keyboards, or pens—and sensors that are used for implicit interaction (Schmidt, 2000). The station has to be configured to reflect the local context, e.g. it has to know which devices and sensors are attached to it. To accomplish this, station and attached devices use the BEACH client’s configuration (see section 6.1.1). The EnvModelModule creates an instance of class Station on system startup to represent the local machine.

Class Station provides means of handling all attached devices. When the station is started, it creates one instance of class DeviceService for every attached Device. The device service is an example of how services can be used to add functionality (see section 6.1.1). The device service notifies the associated device on system startup and shutdown. This can be used to initialize and release resources associated with the device. As the device service has an associ-



ated, shared device object, `DeviceService` is a subclass of `SharedService`. The abstract subclasses `InputDevice` and `OutputDevice` are used to group input and output devices. Devices that have been implemented are described in sections 7.1 (display, mouse, and keyboard), 8.1 (sensors), and 8.3 (audio output).

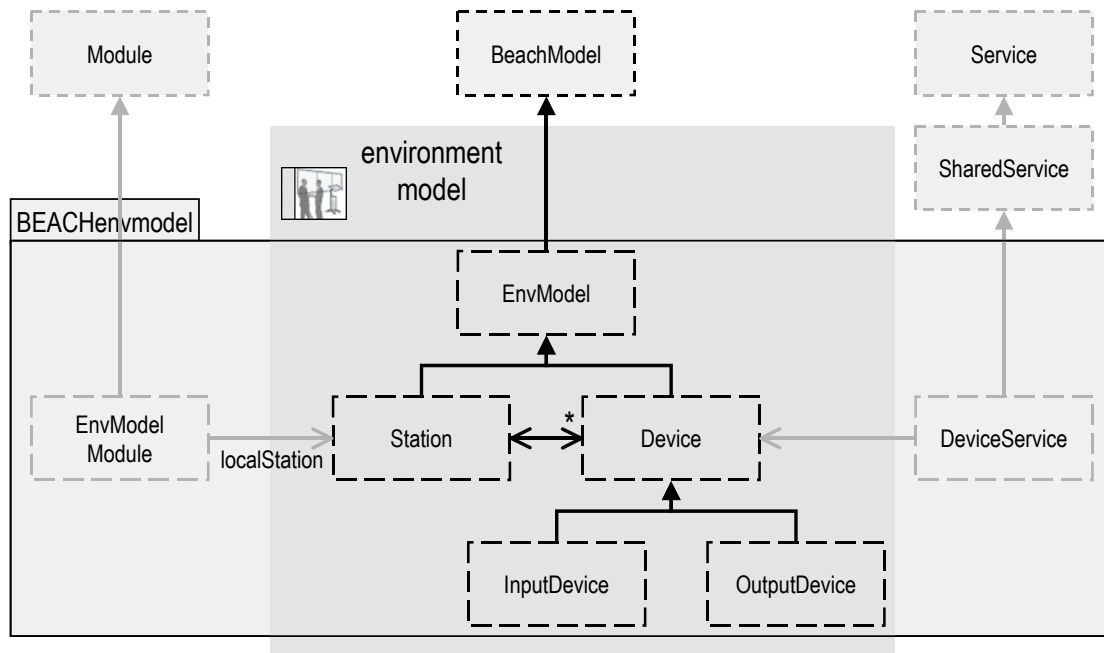


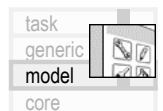
Figure 6-12. The environment model of BEACH defines classes for the stations and devices it can attach. The device service handles the communication with the physical device.

## 6.7. Design of the User Interface Model

The user interface model defines three main abstractions:

- The *user interface application* is concerned with the top-level functionality at the user interface part of an application (see also section 6.3).
- *Tools* provide the connection between the user interface of an application and the application's behavior.
- The *visual interaction area* represents an area on a screen that can be used to visualize a tool.

The module `BEACHuimodel` provides the base classes for user interface elements and for the overall user interface of software applications. Figure 6-13 shows the relationships between classes `UIApplication`, `VisualInteractionArea`, and `Tool`.



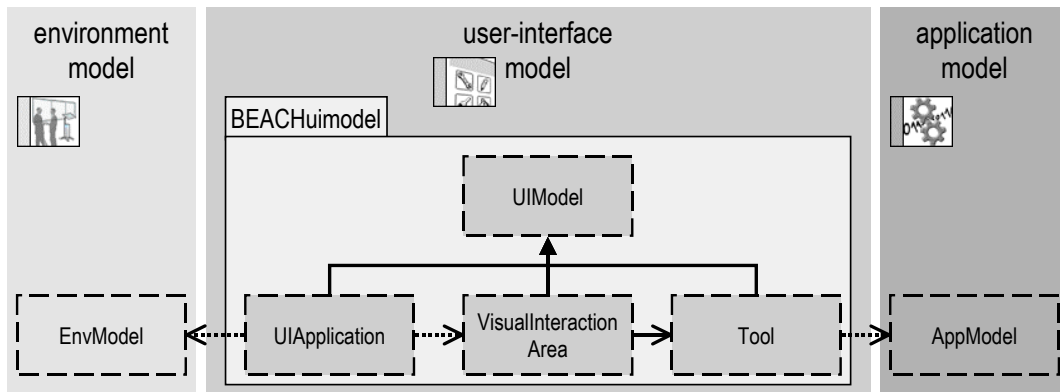


Figure 6-13. BEACH User Interface Model. The dotted lines, in this case, denote possible relationships in subclasses. Class `UIApplication` typically uses information provided by the environment model to configure the user interface, while tools encapsulate the interface to application behavior, i.e. the application model.

Class `UIApplication` represents the top-level user interface part of an application (see section 6.3). It constitutes the root of a user-interface-model hierarchy. Subclasses of `UIApplication` typically will draw on information provided by the environment model, in order to adjust the user interface according to the properties of available interaction devices and other context information (section 4.3.4).

In order to interact with the user interface of an application, at least one *interaction application* (see section 6.8) needs to be assigned to the user interface application. This way, different (or multiple) interaction applications can be used depending on the appropriate interaction style. An example of a visual interaction application is given in sections 7.3.1 and 7.4.1. Section 8.3 describes an interaction application for audio feedback.

Tools provide the connection between the interface of an application (the user interface model) and the application's behavior (i.e. the application model). A tool can directly refer to a specific application model object or to a group of objects, or it has only an indirect or connection to application model objects.

An example for a tool is the document browsers, which is used to switch between different documents (see section 5.4, page 93). In contrast, toolbars (see below) are examples of tools that are not directly connected to an application model. Instead, they group access to functionality that can be provided by arbitrary application objects.

Class `VisualInteractionArea` is the base class for all user interface elements that define a region on the display that can be used to visualize a tool. A concrete example of a visual interaction area that is used in all modern operating systems is the "window". Examples of visual interaction areas defined by the BEACH framework are the "segment" and "overlay", explained in section 7.3.3.

The reason why dedicated support for visual interaction is placed at the model level is that the roomware components rely heavily on visual-based interaction. Additionally, the functionality of visual interaction areas is independent of the application domain, which would require it to be placed at the generic level.

In addition to the base classes of the user interface model class hierarchy, module `BEACHuimodel` defines classes to represent commands and toolbars. *Commands* represent a function that can be triggered by the user interface (Myers and Kosbie, 1996; Gamma *et al.*, 1995). *Toolbars* are used to hierarchically group commands. Depending on the interaction capabilities of the used device, they can be displayed as a popup menu when invoked by a right-button mouse click or

as a toolbar with icons if a pen gesture is used.<sup>32</sup> This is related to the command groups that can be defined to describe the user interface of personal universal controllers (Nichols *et al.*, 2002). In section 6.1.2, it has been explained how new toolbars can be added to the BEACH framework by new modules (especially see fig. 6-3). The root toolbar is described in section 7.3.4.

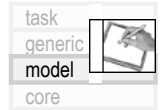
## 6.8. Design of the Interaction Model

The interaction model defined by the BEACH framework aims at visual-based interaction, as the currently existing prototypes of roomware components focus on a pen-based, visual interaction style. The interaction model defines four main abstractions:

- The *view* is the abstraction for any visual representation of a model.
- To increase the reuse of views, *transformations* are used to modify the output generated by view.
- *Controllers* map the user's input actions to the behavior that is invoked in the user interface.
- This is extended by *trackers*, which track interaction sequences and map these to the appropriate functionality.

For visual-based interaction, BEACH uses an adapted version of the model-view-controller (MVC) concept (Krasner and Pope, 1988a; Krasner and Pope, 1988b), tailored for distributed synchronous applications (COAST, 2000b). The model-view-controller concept separates the model with its state and application logic from the handling of input and output. The `BEACHviewmodel`, therefore, defines the base classes for views and controller, based on the support provided in the core layer (see section 6.2). An instance of `BeachView` can be opened on every model object, although in the current implementation of BEACH views are never opened directly on a data model, but rather on an application model for this document element. As mentioned earlier, view objects are always local objects, as they have to communicate with the local display to create a rendered representation of its model. As shared model objects can be modified by virtually every participating machine, views use the core level observer mechanism (described in section 6.1.3). It ensures that the views are automatically notified upon changes to the model's state—regardless whether the model has been changed by the local controller or by a remote machine. Instances of `BeachController` handle all events that are generated on the local machine by the operating system or by drivers for attached input devices.

In order to allocate a part of the display from the underlying operating system, class `Application` defined at the core layer is extended. The abstraction defined at the core layer to describe a part of the display is a `Window` that belongs to an `Application`. The abstract class `VisualApplication` is a subclass of class `Application`, inheriting the ability to open and control a window (fig. 6-14). In addition, it has an associated user interface application (section 6.7), specifying the underlying user interface. The dotted line between class `VisualApplication` and class `BeachView` in figure 6-14 emphasizes that, normally, subclasses of class `VisualApplication` define the root of the view hierarchy to be shown in the window. See section 7.4.1 for an example of how a concrete visual interaction application is defined.



<sup>32</sup> The name “toolbar” is actually not well chosen, as the collection of commands can be presented by the interaction model in an arbitrary form. The name was used because originally only toolbars were supported, and it was not changed afterwards.

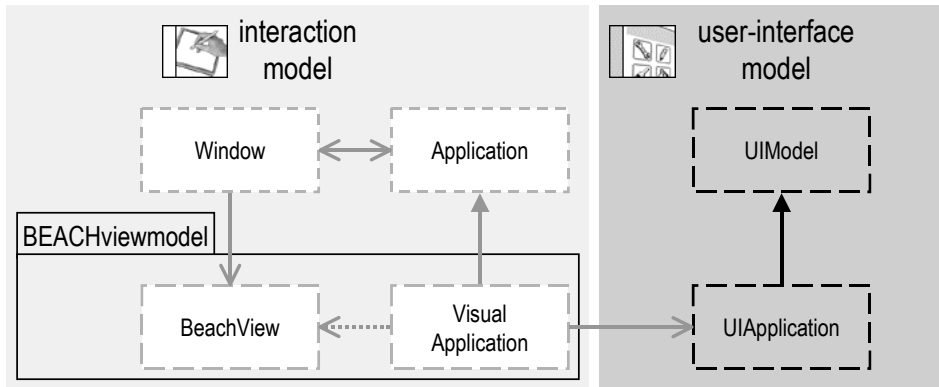


Figure 6-14. Relationships between interaction application, user interface application, window, and views

↓ Section outline

The remainder of this section further explains views and controllers, including defined base classes, wrappers, and an appropriate event dispatching strategy. Then, predefined trackers are presented. Finally, transformations of the rendering of views are discussed.

### 6.8.1 Views and Controllers

`BEACHviewmodel` defines base classes for views according to the hierarchical structure of views. A `DocAtomicView` is a view that has no subcomponents, `DocContainerView` cares about the creation of sub-views for the sub-elements of its model (similar to `DocContainerApp`, see section 6.5), and `BeachWrapperView` is the base-class for all wrappers in the view hierarchy.

To allow combining different interaction modalities and styles, view objects should have no local object state that is not computed from the state of their model (see also the observer mechanism, described in section 6.1.3). When view objects carry no information, but only specify output behavior, the interaction behavior can very flexibly be combined or exchanged, as no information is lost if view objects are destroyed. This is a key feature to realize different interaction styles.

#### Wrappers for Visual Output

The view model defines two base classes for wrappers. `BeachWrapperView` is the default base class for the views of wrapper models, e.g. wrappers in the document or application model. In contrast, *composed wrappers* (subclasses of `BeachComposedWrapper`) have no associated model, as they inherit directly from `CoastAutomaticVisualPart`. They can be used as wrappers additionally inserted in the view hierarchy. For instance, this can be used to add translation or transformation wrappers that modify the output of the contained view (see below).

In addition to the abstract classes, `BEACHviewmodel` defines two concrete classes for translation wrappers, which are commonly used in the view hierarchy to position a sub-view relative to its containing view. Class `PositionWrapperView` can be used as the view for all relation wrapper models that define the position of a document element relative to its container. Class `ComposedTranslationWrapper` has no associated model. Instead it gets the current translation from a *translation composer*, i.e. an object that answers the message `translationFor: aView`.

When translation wrappers are used, the wrapped sub-views are unaware of being translated, as the wrapper creates a translated *local coordinate system* for its sub-view.

#### Controllers Classes

`BEACHviewmodel` defines only one base class, `BeachController`, for all controllers, as the abstract controller does not need to behave differently for views with or without sub-views. The class `NoController` is a concrete class that ignores all events and can be used for views whose controllers should not react to any kind of input.

Every view in the view hierarchy has an attached controller. The controllers, however, are not arranged in a hierarchy themselves (fig. 6-15). Instead, the view hierarchy is used when dispatching events to controllers (see below).

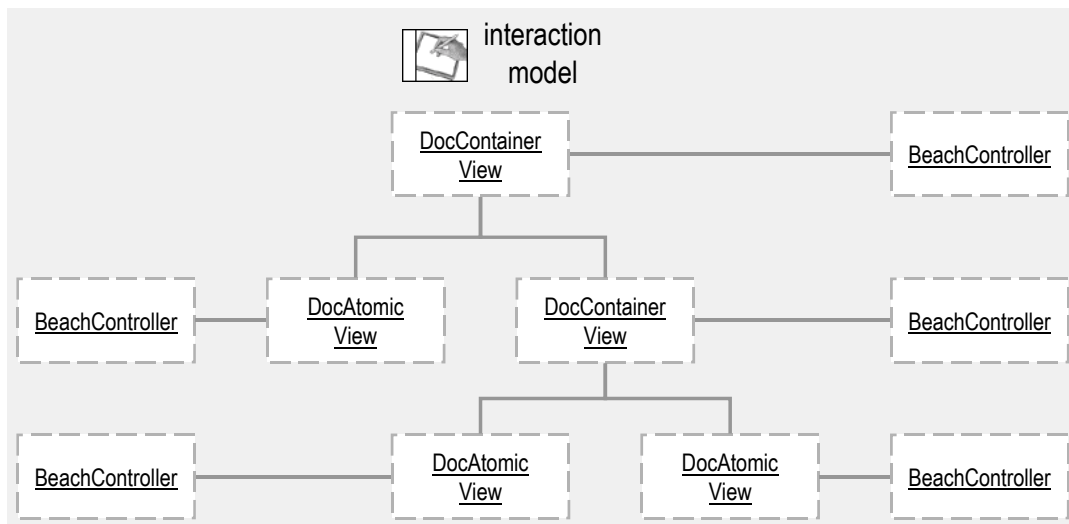


Figure 6-15. Example view hierarchy with attached controllers

#### Dispatch Events along the View Hierarchy

To be able to send events to the topmost view in the view hierarchy at a specific (mouse) position, class `EventDispatcher` is extended with a suitable dispatching strategy (see section 6.2.1). Method `dispatchToPoint` can be used for all events that specify a position inside the window to which they should be dispatched. It traverses the view hierarchy and looks for the topmost view that has a controller that wants to handle this event.

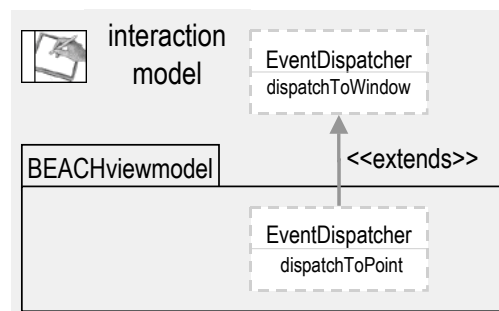


Figure 6-16. Class `EventDispatcher` is extended with a dispatching strategy for mouse events using the view hierarchy.

#### 6.8.2 Predefined Trackers

A tracker encapsulates an interaction sequence (see section 6.2.1), similar to Myers' interactors (1990). The view model defines trackers for generic sequential operations: moving, resizing, and scaling of graphical objects (fig. 6-17).

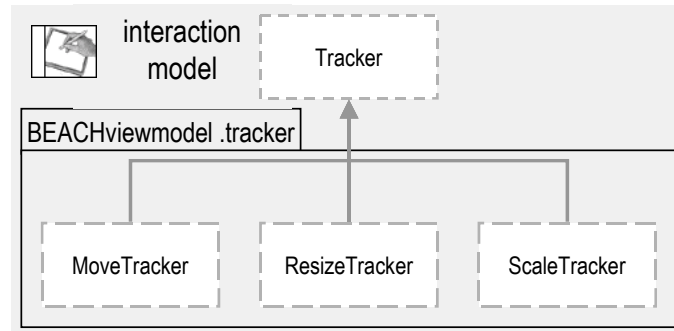


Figure 6-17. Predefined tracker classes. They handle interaction sequences for moving, resizing, and scaling objects.

### 6.8.3 View Transformations

Views should be displayable in different orientations and sizes depending on the current context (req. UH-1, UC-1). Therefore, the core model replaces the standard “graphics context” (that handles the drawing of views, see section 6.2.2) by an adapted version that supports transformations of the rendered output.

This is realized by optionally attaching a transformation object to a view object. This way, the *possibility* to transform output is separated from *how* the output is transformed (strategy pattern (Gamma *et al.*, 1995)). A transformation is an object that responds to messages for transforming points and graphic primitives like images. These transformations are applied by wrapper objects which are inserted into the view hierarchy and which “wrap” the view to be transformed without needing to change it. A similar idea is followed by introducing the *portals* in Pad++ (Bederson *et al.*, 1996) or the *internal cameras* in Jazz (Bederson *et al.*, 2000).

The view model defines three types of transformations, IdentityTransformation, RotationTransformation, and ScaleTransformation (fig. 6-18). Composed transformations use an arbitrary beach model as a “transformation composer”, i.e. an object to compute a rotation or scaling factor. This eases reuse and adaptation.

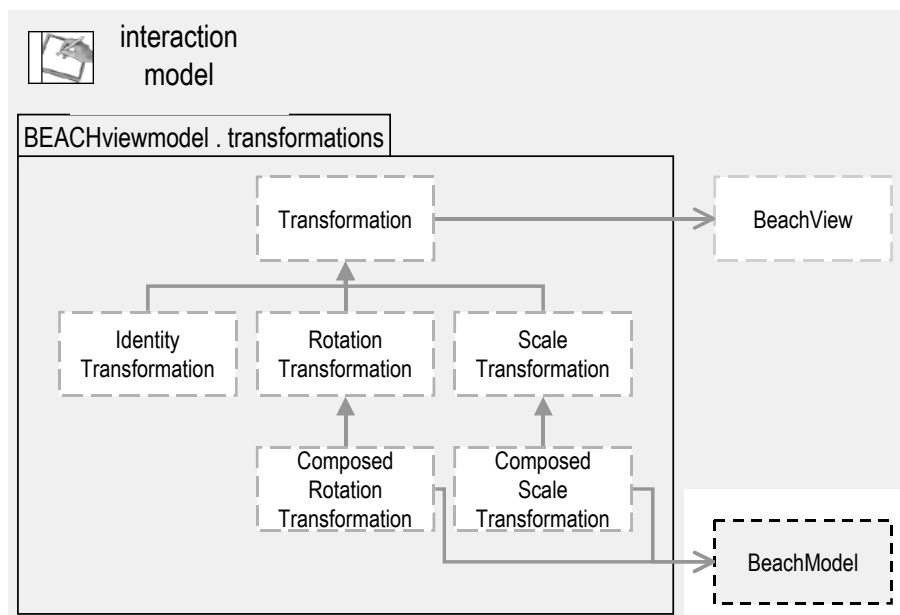


Figure 6-18. Transformation class hierarchy

### Transformation Wrappers and Global Transformation

To be able to modify the presentation of a part of the view hierarchy, a *transformation wrapper* can be inserted into the view hierarchy. It ensures that the rendered output is transformed using its associated transformation. This allows the flexible reuse of views in different contexts that require an adapted rendering.

Within a view hierarchy, multiple transformation wrappers can be used. Therefore, a view might have to be transformed several times before it can be displayed. To speed up drawing, especially to combine several transformations of the same kind (e.g. several rotations), a global transformation was introduced that combines all local transformations (fig. 6-19). Below an example is given of how the transformations interact with the view hierarchy.

Please note that the translation is handled independently from the other transformations. We observed that for the kind of visual-oriented applications that we built for roomware components, graphical objects were frequently re-arranged. Therefore, re-computation and re-painting of moved objects had to be very fast to ensure usability of the system. This led to two optimizations. First, the handling of the translation was separated from the other transformations, to save time-consuming re-transformations when the translation is changed. Second, the underlying graphical system provides native support for translations, which allows handling translations much faster.

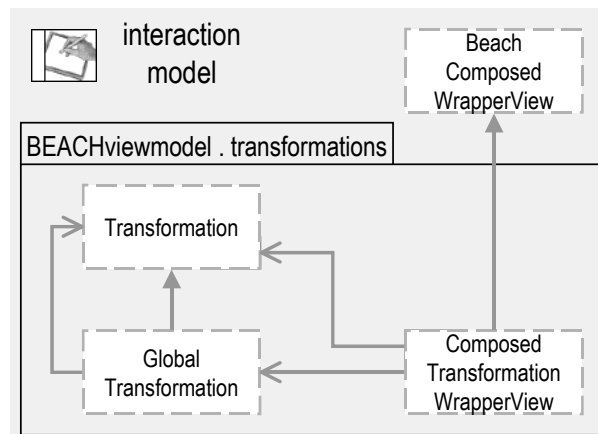


Figure 6-19. Transformations and views

### Example: Transformations in the BEACH View Hierarchy

Figures 6-20 and 6-21 give an example of how the transformations are used. It shows a part of the view hierarchy that is created by the generic *BEACH* elements. A detailed description of the view classes mentioned here is given in section 7.4.

Example 6-3:  
Transformations

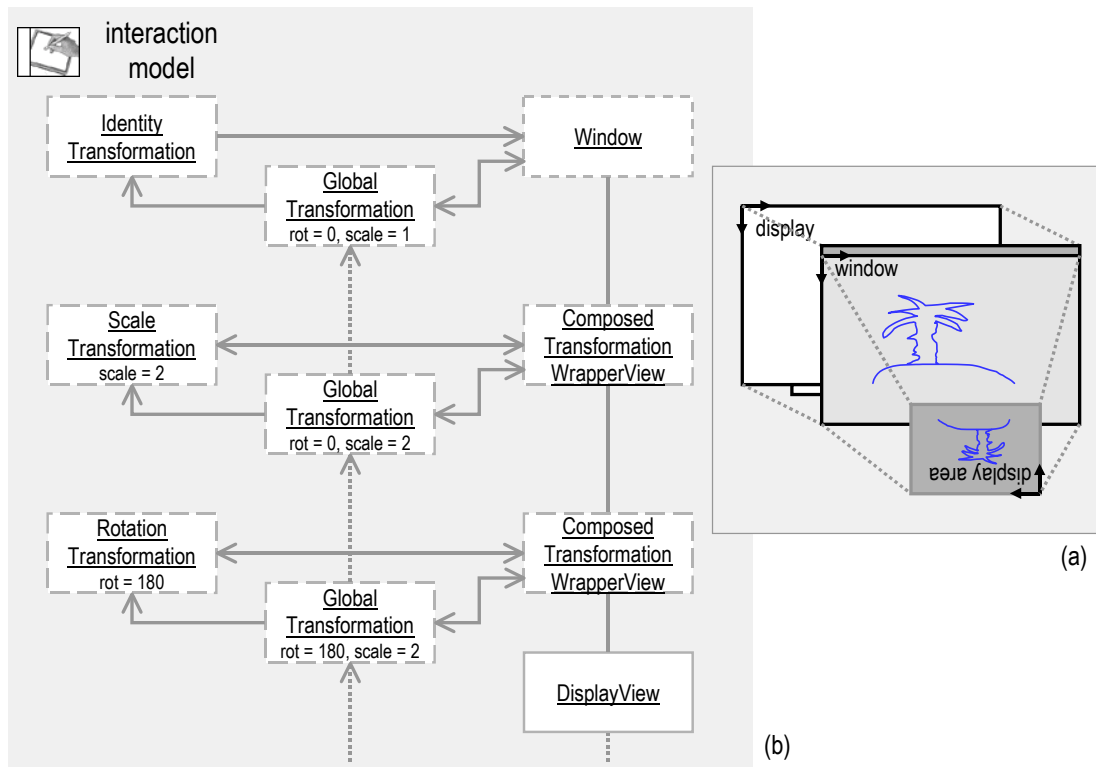


Figure 6-20. Transformations and views example—part 1. (a) The display in this example has a different rotation and resolution than the display area. (b) The display view is wrapped with a scale and a rotation wrapper, to be able to adjust the output properties of the display relative to its display area. The global transformation accumulates all previous transformation to speedup redraw.

The top view is always a *window* (fig. 6-20). The *window* creates the root global transformation, which is attached to an *IdentityTransformation*, as the window defines the global coordinate system. To allow scaling and rotation of a roomware component's display (see section 7.1), two composed transformation wrappers are inserted between the display's view and the window into the view hierarchy. One is used for the scaling, the other for the rotation. The wrappers obtain the current scaling factor and rotation from the display's display layouter (not shown in the figure 6-20, see figure 7-17 in section 7.4.2). In this example, the display is rotated by 180 degrees.

Each wrapper's global transformation accumulates all transformations that are upwards in the view hierarchy. Here, this is a rotation of 180 degrees, and a scaling factor of two.



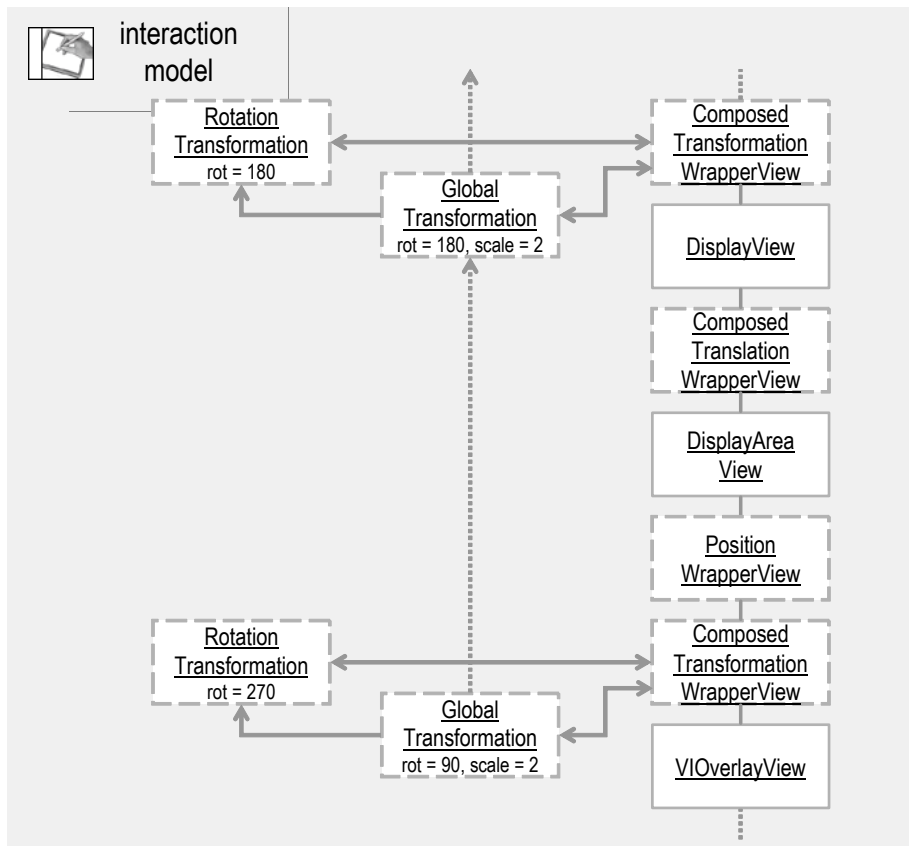
Example 6-4:  
Display area

Figure 6-21. Transformations and views example—part 2: The translation wrapper between the display and display area view is used to select the part of the display area that corresponds to the display. The global transformation merges the overlay's and display's rotation.

Figure 6-21 shows the continuation of the example. The display area (see section 7.1.1) combines all displays of one roomware component. Consequently, each display shows a partition of the display area. To accomplish this, a composed translation wrapper is used to move the local display's part of the display area to the visible area on the screen. Again, the current translation is computed by the display's display layouter.

Finally, this example shows the view of a single overlay (see section 7.3.3) that is displayed at the display area. As overlays can be positioned and rotated freely, a *PositionWrapperView* uses the translation specified in the model to adjust the view's local coordinate system, and a composed translation wrapper handles the rotation. Now it can be seen how the global transformations accumulate transformations. Given that the rotation for the overlay is set to 270 degrees in the model, the global transformation combines this with the display's rotation (set to 180 degrees in the example), yielding a normalized global rotation of 90 degrees.

### Drawing of Transformed Views

All views use a graphics context for creating their visual representation. The abstract class *GraphicsContext* defines methods that can be used for drawing different kinds of graphical objects, such as lines, boxes, text, or images. Subclasses of *GraphicsContext* define the specific implementation to render these graphical objects for different output systems. In order to handle the transformation, the *TransformationGraphicsContext* has been introduced. It uses its associated global transformation to transform drawing requests from the local coordinate system to the global one before passing them to the original graphics context.



The benefit of the shared-object space is that it enables a transparent distribution. Developing a distributed application using the BEACH framework adds no extra complexity compared to an equivalent single-user application. Using transactions and constraints with automatic dependency-detection enables guaranteed consistency among replicated objects and between the shared object and its (visual) presentation. Developers do not have to worry about sending update notifications among clients explicitly. This reduces the possible errors and the necessary development effort, which is the main goal of this dissertation.

The support for **modules** allows for adding code without modifying the existing framework. The presence of new modules is detected using reflective features. **Hooks** can be defined by modules to allow integration with other extensions. To plug in functionality, **services** allow the integration of functionality that can be started and stopped. Modules can identify services that should be started automatically at system startup. The **configuration** is responsible for handling the overall configuration of the system. Modules, hooks, services, and configurations are abstractions that ensure the extensibility of the developed software systems (req. S-2).

To ease implementation of the interaction model, the core level includes abstractions to open **windows**, handle **views**, and dispatch **events**. This ensures the independence of the underlying operating system, as the abstractions encapsulate platform-specific details.

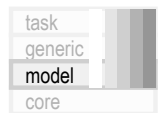
### 6.9.2 Properties of the Model layer

The *model layer* of the BEACH Model framework implements abstractions to separate the five basic concerns, which are independent from the platform and application domain. The *data model* is a simple model, using composites and wrappers to ensure its extensibility. In general, composites and wrappers are two simple, but powerful abstractions that help enable reuse and extensibility.

**Composites** define a uniform interface to work with any kind of hierarchy. This enables *extensibility*, as new types of elements can be added to the hierarchy without having to adapt the places in which they are accessed. Composites also ease *reuse*, as the same functionality for working with hierarchies can be employed for different kinds of hierarchies. For example, views offer mechanisms for visualizing a model and its sub-components. These mechanisms can be used for all models that can be traversed as a hierarchy. This technique was used to visualize the dependencies among services (that actually form a directed graph, not a hierarchy) for debugging purposes. **Wrappers** offer a significant improvement in *extensibility*, as new functionality can be added to existing models and existing behavior can be augmented on a per-object basis. This is, for instance, used when transformation wrappers are inserted into the view hierarchy. In addition, wrappers have a high potential of being *reused*, as the same wrapper can be combined with different objects, as long as they provide the same protocol; the transformation wrapper can be used to scale the visualization of the complete display area or to rotate the presentation of a workspace. Composites and wrappers are two techniques to realize a minilithic design (see section 3.2.2) that uses small components that can be flexibly combined.

The *application model* is used to add application behavior to information objects. To be able to use fine-grain application models, its implementation provides base classes that **link the structure** of data and application model. Thus, an application model can observe its currently attached data model object in order to create and remove sub-application models for sub-data models that are added and removed. The separation of data and application model enables reuse of both data and application models. The same data model can be manipulated by different application models, and the same application model can be used to edit different kinds of data models, as long as they rely on compatible protocols.

The *environment model* defines a minimal model of the environment. It includes the **station**, as abstraction for the device the client is running on, and a notion of **device**, to model all kinds of interaction devices that can be attached to a station. Using the device service, device driv-



ers for new devices can be integrated in the framework, which ensures the extensibility for new interaction devices and forms of interaction. The device hook is used to provide visual output (display device, section 7.1), acoustic output (MIDI device, section 8.3), mouse, keyboard, and pen input (section 7.1), and physical objects as input elements (section 8.1).

The *user interface model* provides the root abstractions for an application's user interface. The **user interface application** is the base class for the root object of the user interface. For the construction of the user interface, it draws on information made available by the environment model. A **tool** links the user interface with the application model. The **visual interaction area** encapsulates a part of a display that can be used to render a tool. These abstractions decouple the functionality and the space in which it is presented at a high level.

The *interaction model* uses the **model-view-controller** architectural style to render output and process input, allowing separate input and output of applications. As **views are state-less and computed** from the state of the other models, different views and support for other interaction modalities can be combined flexibly. Implementing views as local objects enables the adaptation of the visual representation to the local context. To decouple the visual representation from the properties of the underlying device and to enable reuse of views for adapted presentation, the rendering of views can be independently transformed using **view transformations**. As extension of controllers, which process a single event at a time, **trackers** are an abstraction that allows controlling event *sequences*. By defining support for typical interaction sequences, a small set of trackers can be used in a number of situations.

---

↓ Next chapter

The following chapter uses the BEACH Model framework to provide generic components for synchronous collaboration with roomware components. This focus there is on informal meetings and collaboration situations.

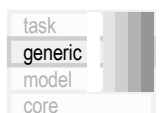
## 7. Generic Support for Collaboration in Roomware Environments

---

This chapter presents the part of the BEACH framework providing generic elements to support synchronous collaboration with roomware components. The focus is on informal meetings and collaboration situations. It implements the generic layer of the BEACH architecture and offers reusable components for roomware applications. The BEACH Generic Collaboration framework proves that the BEACH conceptual model helps identify reusable software components.

---

The previous chapter introduced the low-level part of the BEACH software framework core and the model layer of the BEACH architecture. The BEACH Model framework (presented in the previous chapter) is independent from an explicit application area, and is applicable for many different ubiquitous computing environments.



This chapter, now, presents the part of the BEACH framework providing generic elements to support synchronous collaboration with roomware components—tailored for meeting situations as described in section 2.2. Thus, it implements the generic layer of the BEACH architecture. As it is designed to be used with the roomware components we have built in the context of the i-LAND project, this part of the framework is strongly tailored to their needs. The description of the roomware components developed in the i-LAND project is given in section 2.1.

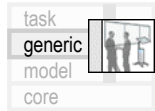
Currently, the roomware components support pen or finger as the main medium for input. Thus, BEACH emphasizes direct visual interaction. All roomware components have a permanent (in parts wireless, see fig. 5-6) network connection aiming to support synchronous collaboration, and no “slow” CPUs. This implies that BEACH is not designed for very small devices such as PDAs and for devices not having a permanent connection to the network.

Although being designed for roomware components, this part of the BEACH software framework can be used to illustrate how the proposed conceptual model and architecture can be applied in a concrete environment.

The BEACH Generic Collaboration framework offers a set of pre-defined components. The components defined at the generic layer can be used as a very simple application for collaborative whiteboard interaction already.

The following sections explain the modules defined by the generic layer as shown in figure 5-3.

↓ Section outline



## 7.1. Generic Environment Model: Roomware Components

The BEACHroomware module provides an environment model for roomware components, defining two abstractions and several concrete classes:

- The *roomware component* abstracts from the computers that are used to construct what is perceived as a single device by users.
- The *display area* has a similar job: it abstracts from the displays and defines a homogeneous area that actually might be built out of several displays.
- Mouse, pen, and keyboard are defined as basic *interaction devices*.

### 7.1.1 Roomware Components and Display Area

One important part of the representation of the physical environment is the configuration of roomware components. BEACHroomware extends class Station defined in BEACHenvmodel to be part of a roomware component (fig. 7-1), allowing multiple stations to be part of a roomware component (req. U-2, U-3).

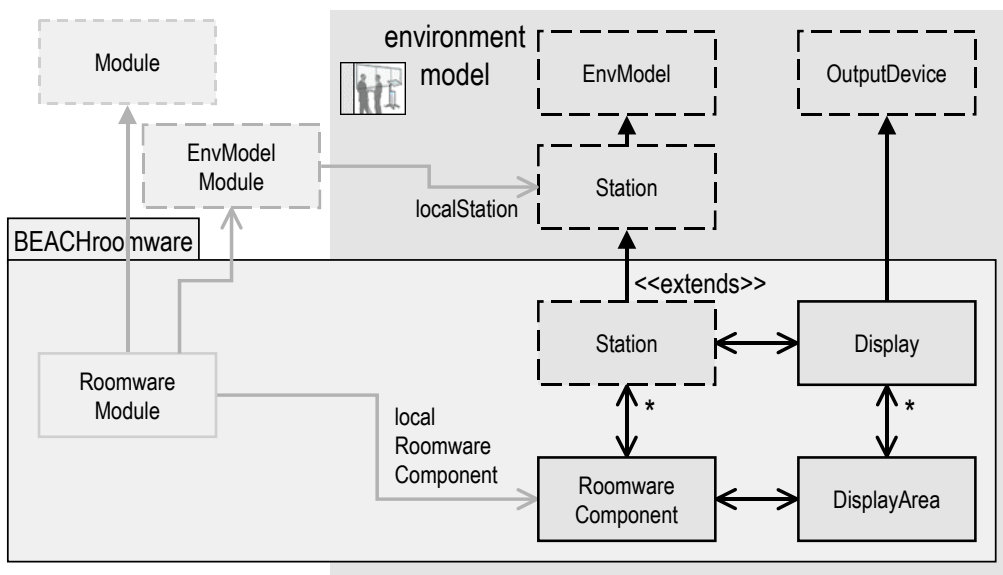


Figure 7-1. Class diagram showing relationships between roomware components, stations, displays, and display areas

Using the module hook (section 6.1.2), module BEACHroomware is represented at runtime by class RoomwareModule. Class RoomwareModule can be used to access the roomware component the client is currently running on. Therefore, class RoomwareModule references class EnvModelModule that manages the local station.

A roomware component consists of one or more stations. Each station can have a display. The displays of all stations belonging to a roomware component are combined into a display area, which represents the complete interaction area of the roomware component.<sup>33</sup>

<sup>33</sup> The “display area” is called “display” in (Swaminathan and Sato, 1997); the “display” as a physical device, is called “screen”.

Note that the display is implemented as an output device. This implies that it can be managed like any other device by the functionality provided at the model level (see section 6.6). Using the display's device service, the window that is used for BEACH's visual presentation can be opened and closed when the display device is started or stopped (see section 7.4.1).

Using the DynaWall (shown in figure 2-1) as an example, figure 7-2 illustrates the environment model that is constructed for this DynaWall at runtime. For each PC, a station object (`Station1` to `Station3`) is created that is associated with the object representing the DynaWall. Their displays are combined by the object that stands for the DynaWall's display area.

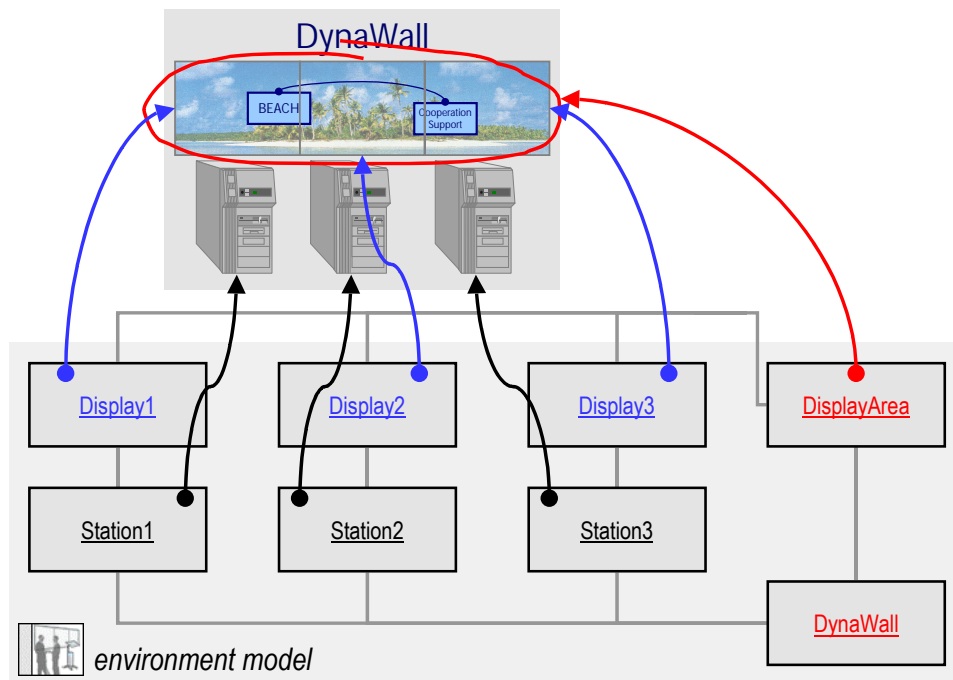


Figure 7-2. The representation for a DynaWall consisting of three computers that are combined to form a homogeneous display area. This allows the complete area to be used to show one large workspace.

If displays are added to or removed from the display area, the views showing it will immediately adjust the size of the available area (req. U-4) due to the dependencies between the environment model and the views. The interaction model for roomware components is described in section 7.4. An example of dynamically changing display areas is given in section 8.2.

### 7.1.2 Input Devices

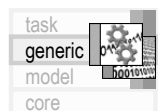
In addition to the model for roomware components, `BEACHroomware` defines two concrete classes for interaction devices, class `PointingDevice` (representing mouse or pen input) and class `Keyboard` (which is used in case the roomware component has also an attached keyboard).

## 7.2. Generic Data and Application Model: Document Elements

The basis for documents created with BEACH is a hypermedia data model. The instances of data model classes are always part of the shared object space, as this gives several users the possibility to access these objects simultaneously. Module `BEACHdocument` defines data, application, and interaction models for the document elements.

### 7.2.1 Document Data

The generic document elements are separated according to the structure defined by the atomic, container, and wrapper elements of BEACH's data model.



## Atomic Document Elements

Atomic document elements are shown in figure 7-3. Class `DocText` contains text typed by a keyboard. `DocGeometric` is a super-class for geometric elements. `DocScribble` models hand-written input (scribbles). Hyperlinks are constructed using an instance of class `DocReference` that is placed in a workspace and connected by an instance of class `DocLink` with another reference. The reference functions therefore as an *anchor* in terms of the Dexter hypermedia model (Halasz and Schwartz, 1994).

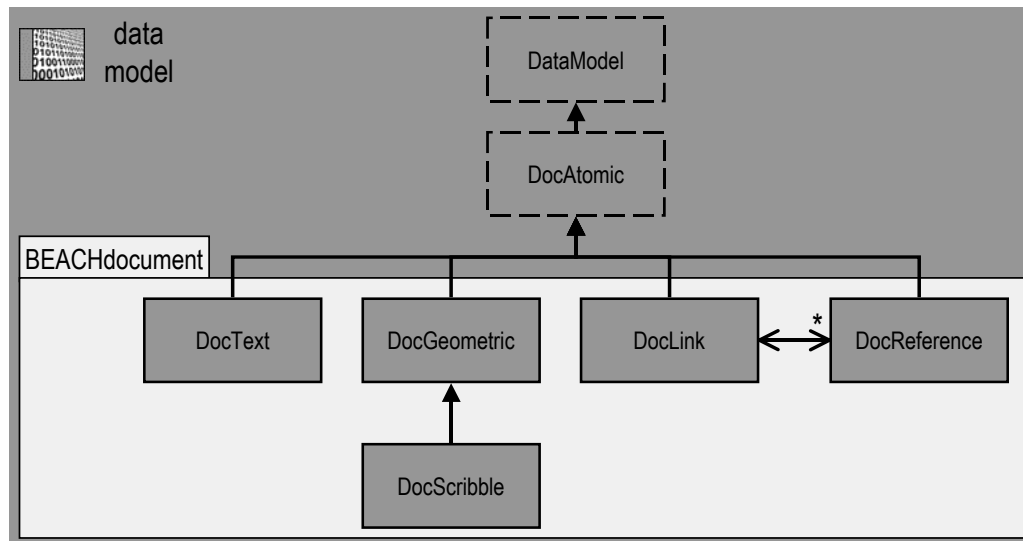


Figure 7-3. Atomic document elements that are defined by BEACH's generic document model.

## Container Document Elements

Like many hypermedia data models (Conklin, 1987), BEACH's generic data model builds on a containment hierarchy that is primarily composed from workspaces (the equivalent of a page). In addition, external objects such as images are also container objects, as they can contain annotations made by the user (especially scribbles). Figure 7-4 shows the class hierarchy.



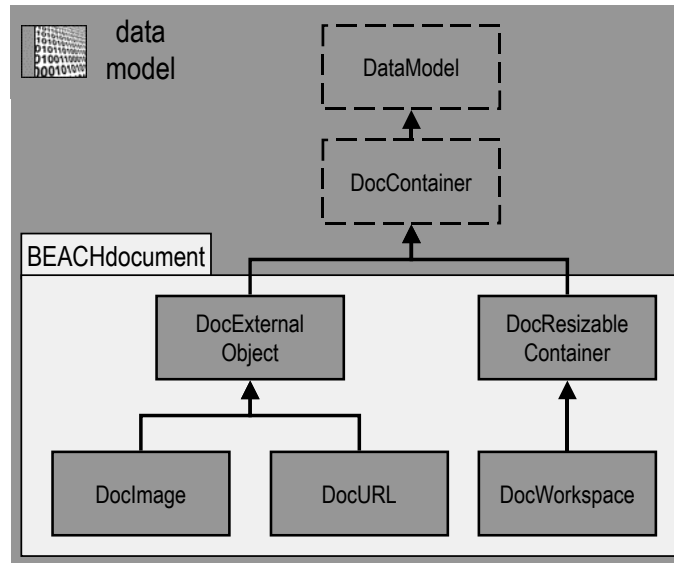


Figure 7-4. Container document elements that are defined by BEACH's generic document model.

### Wrapper Document Elements

Large visual display-surfaces that are provided by roomware components fit well for a *spatial hypertext* model like the one defined by VIKI (Marshall and Shipman, 1995; Marshall *et al.*, 1994). This makes it necessary to position document elements freely within container elements. As the position at which an element is placed describes the *relation* to its container, a relation wrapper is used (see section 6.4). A *PositionWrapper* is inserted between a workspace and its elements (fig. 7-5). An example is shown below in figure 7-12.

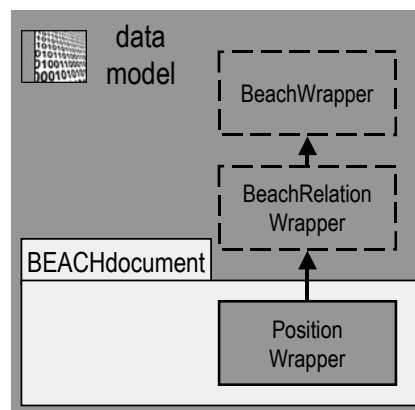


Figure 7-5. The position wrappers is the only relation wrapper that is defined by BEACH's generic document model.

Modeling the position as a relation wrapper gives flexibility in re-using the document element in different contexts, as they do not need to make an assumption about the environment in which they are placed. For example, the same classes can be used for elements placed inside a table with automatic adjustment of rows and cells, or in a tree- or map-like structure, which computes the position from the relationship to other elements.

### 7.2.2 Document Applications

In order to work with documents, a set of application models is provided by *BEACHdocument* that support editing and navigation.

## Atomic Document Application Models

As it turned out that modeless interaction seems to ease collaboration at multi-user devices (req. C-3) (Pier and Landay, 1992; Prante, 1999), not much editing state is actually added by the document application models. Figure 7-6 shows the atomic document application model classes.

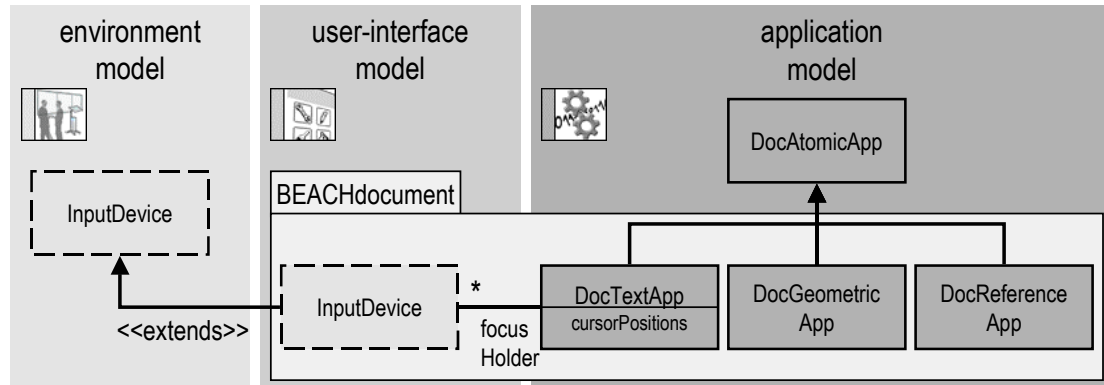


Figure 7-6. Atomic Document Application Models. The application model for text elements allows several input devices to provide input concurrently.

Class `InputDevice` is extended to refer to the application model that currently has the input focus. Class `DocTextApp` defines an association of all input devices with their input focus currently lying on this text element. This way, it is possible that multiple input devices edit a single text object concurrently.

## Container Document Application Models

As noted in section 6.5 above, all container application models automatically create sub-application models for the sub-elements of their associated data model. External objects are objects not directly handled by BEACH. Class `DocExternalObjectApp` allows starting of applications to edit the external object, like a paint program for images or a Web browser for URLs. It remembers the process ID of the started application to be able to re-import changes after the application is closed.

## Wrapper Document Application Models: Add Movement to Positions

Roomware components require new forms of interaction (req. H-1). For example, it is helpful for users at physically large interaction areas to ease rearranging of objects across large distances by permitting throwing of document elements across the display area (Geißler, 1998; Streitz *et al.*, 2001). We experimented with several different implementations of throwing across multiple displays that belong to the same display area (e.g. at the DynaWall, see section 2.1). According to the experiences we gathered, throwing works most smoothly if only one transaction is used to specify the throw-parameters—and the animation is handled independently by the local interaction models of every involved machine.

Therefore, class `PositionWrapperApp` can store information about the time, position, and direction if the object has been thrown. The start time refers to a synchronized millisecond time value to allow synchronous animation on multiple stations. An end time does not need to be stored, as it can be computed from the start time, vector, and a diminution factor, which is currently implemented as the same constant value for all elements.

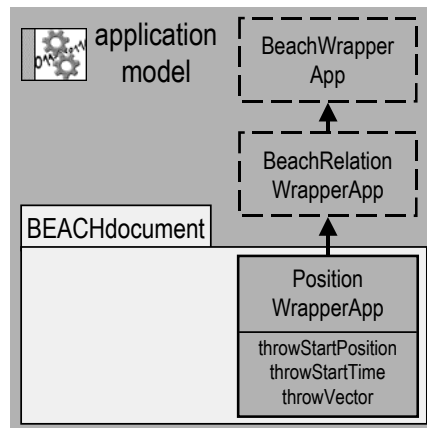


Figure 7-7. Wrapper Document Application Models. The application model for the position wrapper handles the throwing of document element across large display areas.

In order to implement time-based throwing, the COAST framework had to be extended. First, a synchronized millisecond timer was needed. Second, support for time-based invalidation had to be added. In COAST, the redraw of views is handled upon invalidation only. Normally, slots send invalidation notifications to all observers as soon as their value is changed. In order, to be invalidated after some amount of time, slots can register at the timer to be notified after a given duration. This way, the `position` slot of class `PositionWrapperApp` can request to be invalidated every time it has moved by one pixel. This mechanism is related to the animation constraints described in (Myers *et al.*, 1996), but provides support at a lower abstraction level. While the animation constraint is an abstraction that can be used to define the animation of the throwing, the time-based invalidation can be used to implement synchronized animation constraints.

### 7.3. Generic User Interface Model: Roomware User Interface

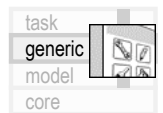
As the BEACH Generic Collaboration framework is designed to be used as an application framework, beside re-usable software components it also provides the complete design and implementation of an “application without application-specific functionality”. The module `BEACHroomwareInterface` has four major tasks.

- It defines a concrete *user interface application* (see section 6.7), used to visually render applications on the display of the local station.
- The *document browser* provides functionality to navigate among workspaces.
- Two user interface elements are *segments* and *overlays*, which are provided as a concrete implementation of the abstract concept of a visual interaction area (see section 6.7) that is appropriate for roomware components.
- In order to be able to make the functionality that is provided by modules and services available in the user interface, the concept of a *toolbar* is used that can dynamically add and remove commands that can be invoked.

#### 7.3.1 Display User Interface Application

The BEACH framework uses one large window on every station that normally covers the complete area of the display. The module `BEACHroomwareInterface` therefore defines a user interface application. It is used as the basis for the generic interaction model to open a window on the local station’s display and creates views for display and display area of the environment model (see section 7.4).

Class `DisplayUIApp` is a concrete subclass of `UIApplication` (fig. 7-8). Every station creates an instance of class `DisplayUIApp` that is attached to the display object representing its display. In addition, class `DisplayArea` that is part of the environment model is extended into the user inter-



face model to be able to refer to a set of root visual interaction areas that should be shown on the display area. This extension means on the conceptual level that the user interface incorporates a concept defined in the environment model in order to integrate the user interface with the environment (req. U-3, UH-3).

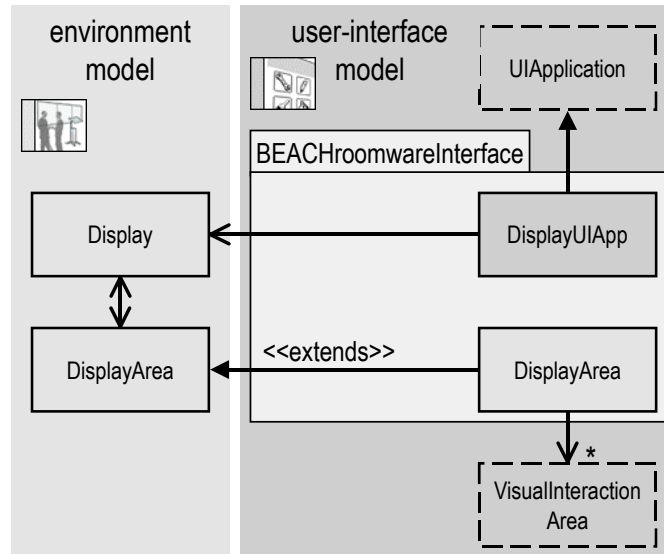


Figure 7-8. The display user interface class `DisplayUIApp` is instantiated by every station to refer to its local display. This is used as the basis for opening a window and views.

### 7.3.2 Document Browser

The document browser shows the contents of a workspace and allows navigating to different workspaces. Class `DocumentBrowser` has an associated application model (`document`) that is used to edit a document (fig. 7-9).

When navigating, the document browser stores visited documents in an ordered collection (`history`). It keeps a reference to the visited application model in the history—instead of storing the data model. This allows preserving the editing and collaboration state when switching between workspaces. This way, it is also possible to switch between different application models for the same document. This gains a high flexibility in dynamically adjusting the collaboration mode (req. C-2, section 2.5).

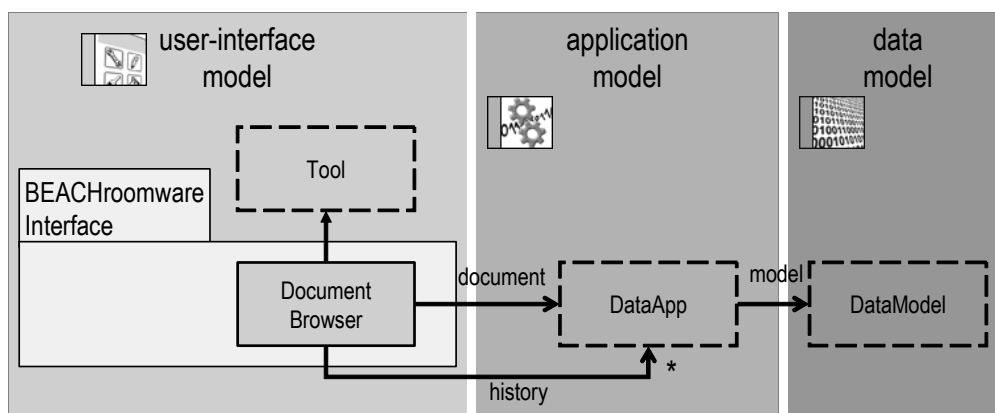


Figure 7-9. The document browser is a tool allowing navigation between different workspaces. Its history stores the visited application models with the associated data model (instead of data models only) in order to be able to restore editing state.

When, for example, users share the same document browser, they work in a *tightly-coupled mode* (sections 2.5 and 4.4), also navigating together. This can be used for shared browsing. If the same application model, but different browsers are used, users can get awareness of other users' activities, but can switch to a different workspace when desired. If only the data model is shared, it is possible to use a different application and/or user interface model. For instance, one user can monitor the overall workspace using an outline or overview application, while others can work on details within the workspace.

### 7.3.3 Overlay and Segment

The main elements of the user-interface of BEACH are segments and overlays (Prante, 1999). The complete visual interaction area of a roomware component can be divided into non-overlapping *segments*, which define the space available for an application model, e.g., a document browser. Segments have an advantage over overlapping windows if enough display space is available (Mynatt, 1999a; Kandogan and Shneiderman, 1997; Moran *et al.*, 1997 (regions)). In addition, *overlays* can be positioned freely and are used in a similar way to the windows of most popular operating systems. They also provide space for a tool, such as a document browser. They would normally be used for toolbars and other smaller tools that have to be at hand all the time (fig. 7-10).

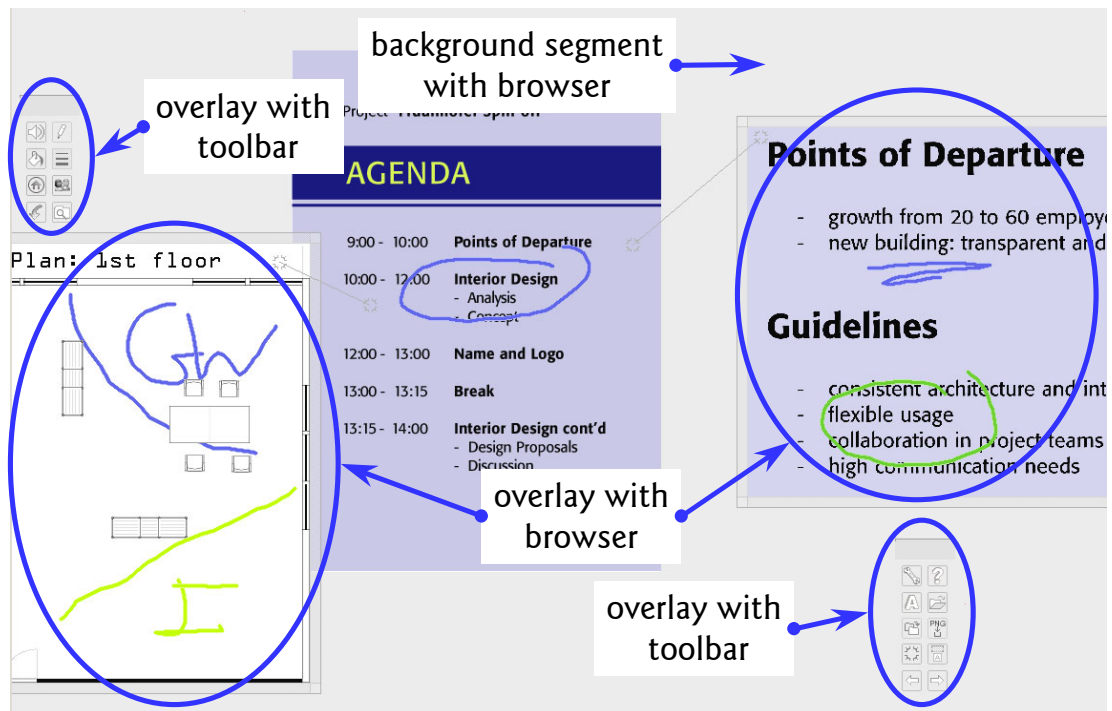


Figure 7-10. Overlays can be used to provide space for tools, such as document browsers or toolbars.

Figure 7-11 shows the concrete classes for segments and overlays (classes `VisualInteractionOverlay` and `VisualInteractionSegment`) that are defined in module `BEACHroomwareInterface`.

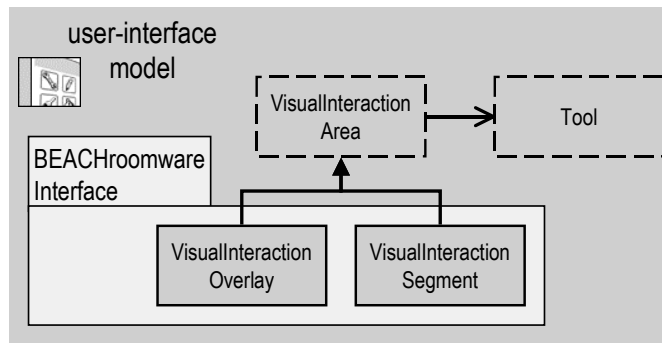


Figure 7-11. Overlays and segments are the two concrete visual interaction areas defined by the BEACH Generic Collaboration framework.

Example 7-1:  
Workspace browser

Figure 7-12 illustrates the relationships between environment, user interface, application, and data models. To display a workspace (a data model), an application model for this workspace is created and shown in a document browser. The browser is placed in the segment of the display area belonging to a roomware component (not included in the figure). All these classes extend the base classes defined in the basic-models layer. If the workspace contains sub-elements, the workspace application model automatically creates appropriate sub-application model objects for all sub-elements.

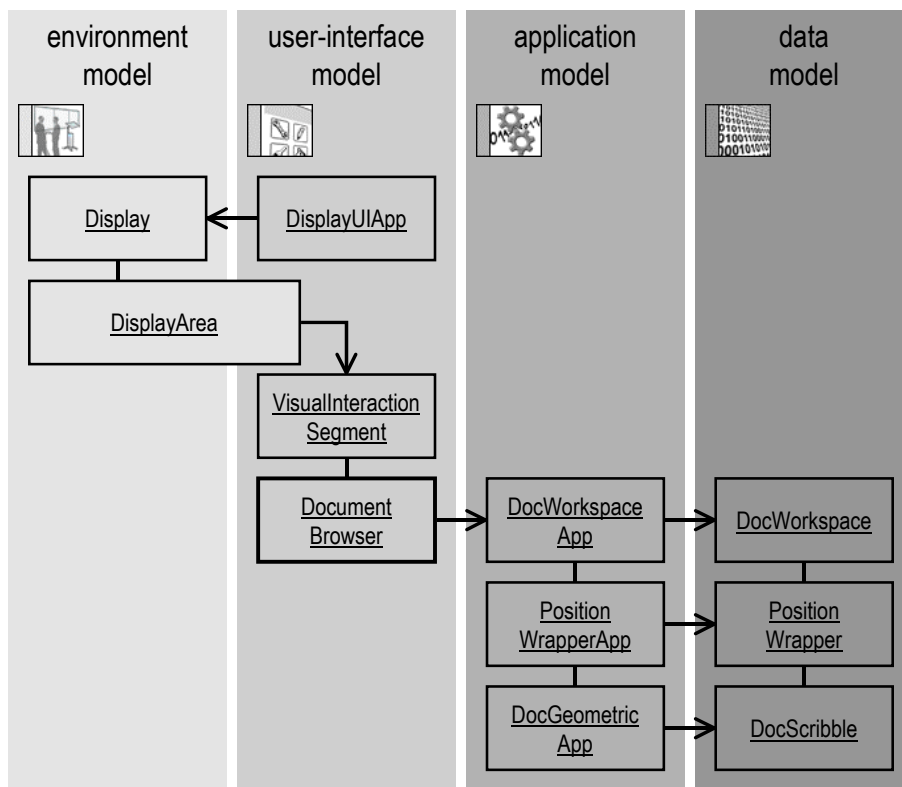


Figure 7-12. Object diagram, showing instances of all models (except the interaction model). The display belongs to a display area, which shows a segment containing a document browser pointing at a workspace. This models a part of the scene shown in the screenshot in figure 7-10.

### 7.3.4 Root Toolbar

Module BEACHroomwareInterface also defines a default root toolbar for applications. It is used as the place for all commands that should be made available to the user via toolbars (section

6.7). As a toolbar is a tool, it can be placed in any visual interaction element. Normally it would be placed inside an overlay, as this allows positioning it freely on a display area (see section 2.3.2).

Figure 7-13 shows an example of a toolbar within an overlay. A “sub-toolbar” command enables hierarchical structures of toolbars. A special sub-toolbar, which is part of the root toolbar, provides a hook for modules to plug in additional commands or toolbars. This hook was described in section 6.1.2 and illustrated in figure 6-3.

Example 7-2:  
Modules toolbar

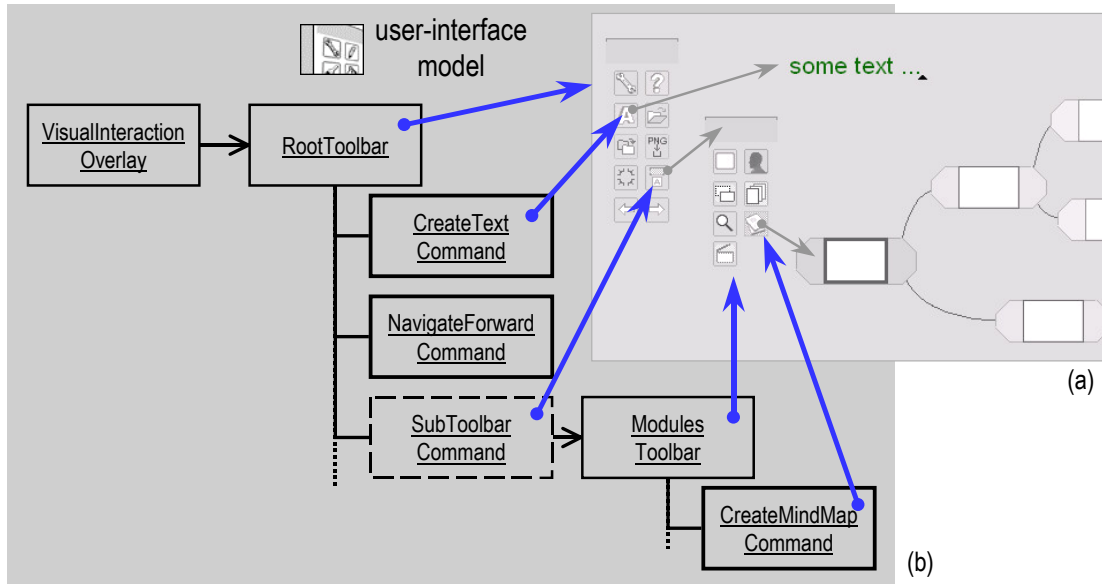


Figure 7-13. Example showing a toolbar within an overlay. (a) A screenshot showing the root and modules toolbars, and a text object and a mind-map that have been created. (b) The objects used to model this part of the user interface.

The toolbar hook is also used in section 9.1 to plug the functionality defined by the modules into the toolbar.

## 7.4. Generic Interaction Model: Visual Output for Roomware Components

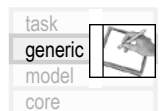
To enable interaction with roomware components, the interaction model defines both view and controller classes for all other models. As every view object has one attached controller object, the figures in this section show view objects only. BEACHroomwareInterface defines two main abstractions:

- It defines the *visual display application*, used to visually render applications on the display of the local station.
- The *display layouter* contributes the algorithm to calculate which part of a display area must be rendered on a given display and which transformation has to be used.

### 7.4.1 Visual Display Application

In order to open a user interface on the display, BEACH's roomware interface module defines concrete subclasses of the user interface and interaction applications (see sections 6.7 and 6.8).

The corresponding part of class DisplayUIApp on the interaction model side is class DisplayVApp. It is used to open a window on the display of the local station. Inside the window, a view of class DisplayView is placed for this display, which in turn opens a sub-view of class DisplayAreaView for its associated display area (fig. 7-14). The dotted lines in the figure denote indirect associa-



tions between the classes. The detailed description of the constructed structure is discussed in section 6.8.3, especially figures 6-20 and 6-21, to illustrate view transformations.

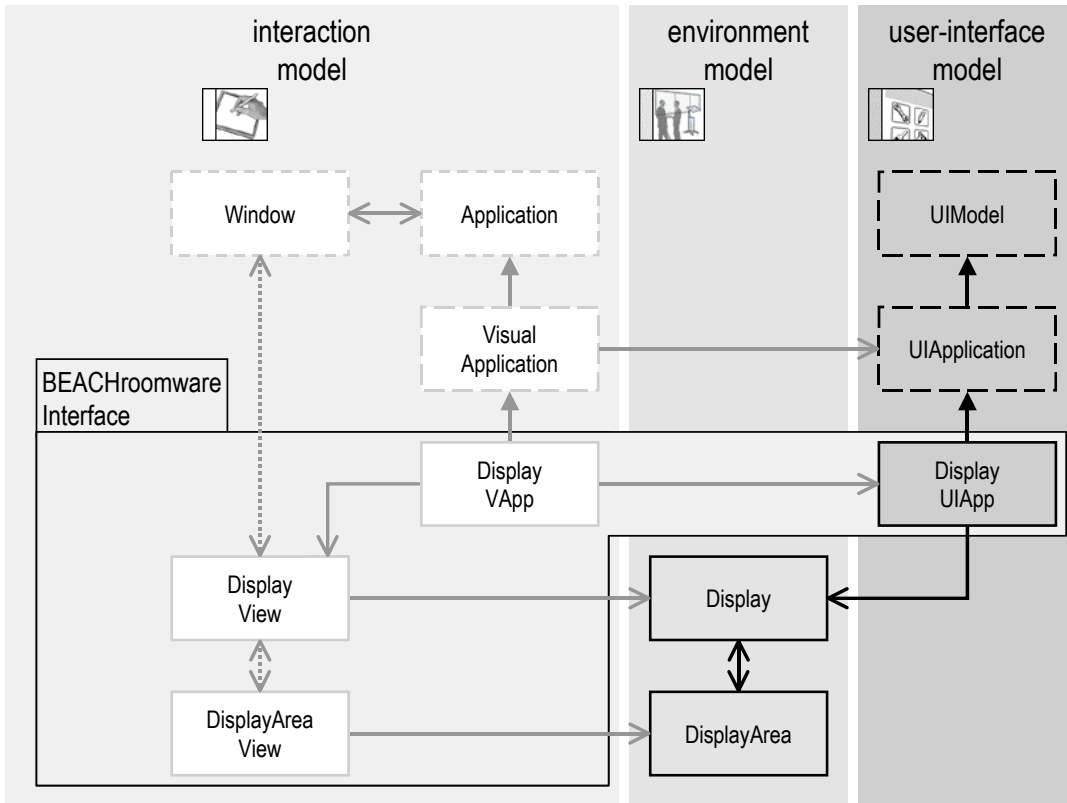
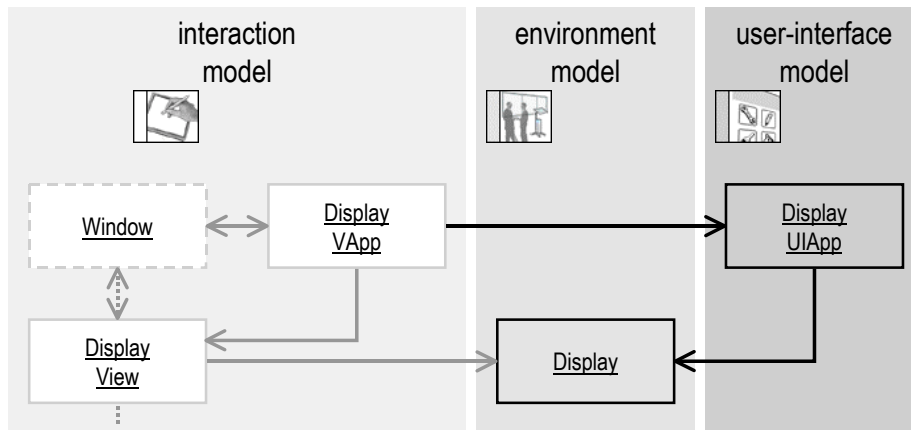


Figure 7-14. The view subsystem of module BEACHroomwareInterface defines the basis to open a window that shows the local display's part of the display area.

When the station is started, it creates one instance of a device service for every attached device (see section 6.6). The display device service creates the display user interface application for its associated display (if there is not already an instance for this display). When the display device service is started, an instance of class DisplayVApp is created (fig. 7-15) that is associated with the display user interface application. This instance opens a full-screen window that shows a view of the display of the associated display user interface application.





Example 7-3:  
Display application  
and view

Figure 7-15. Object diagram for display interaction and user interface applications (instantiation of figure 7-14). The display *interaction* application opens a view for the display specified by its associated display *user interface* application.

The display view, in turn, opens a view for the display area that is currently attached to this display as a sub-view (fig. 7-16). The display area view opens sub-views for all of its segments and overlays, which (in turn) open sub-views for their associated tools.

Due to the dependency-mechanism provided at the core level, the views will automatically be re-computed as soon as the associations among any model-objects change. This happens, e.g., if the display is moved to a different display area, or a different workspace is attached to the document browser. This re-computation is triggered independently of the client that actually changed the shared model.

Example 7-4: View dependencies

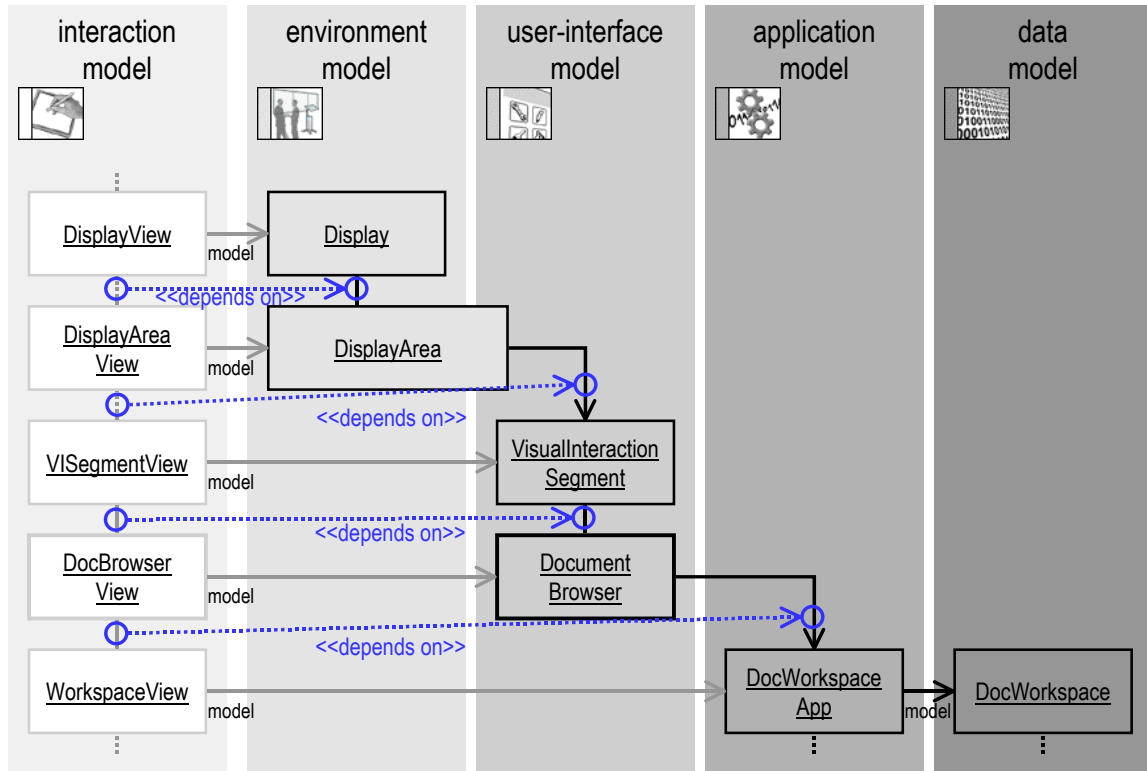


Figure 7-16. Example view hierarchy (continued from figure 7-15): display to workspace. Interaction models are added to a part of figure 7-12. The dependencies between associations in the view hierarchy and corresponding models are highlighted. Dotted lines denote omitted wrappers (for simplification).

#### 7.4.2 Multiple-Computer Devices

If one roomware component consists of multiple stations (req. U-2), their displays are combined into a single display area (as in the example of the DynaWall above). Every display's view object should then show the part of the display area belonging to its local display in a way that yields a homogeneous display area. However, there are cases where not every display has the same orientation or resolution with respect to the other displays of a display area (an example is given in section 8.2 figure 8-14). In these cases, the view transformations (see section 6.8.3) can be used to adapt the display view without having to modify any view class. Every display has the attribute *position* defining the position relative to the other displays. The attributes *scale* and *rotation* specify the magnification and orientation of the display relative to the display area.

An instance of class `DisplayLayouter` is responsible for assigning the right coordinates to each display (fig. 7-17). It computes the values for the transformation wrappers, which are inserted in the view hierarchy by analyzing the properties of all displays that are combined in the local display's display area. The used wrappers are explained in section 6.8.3.

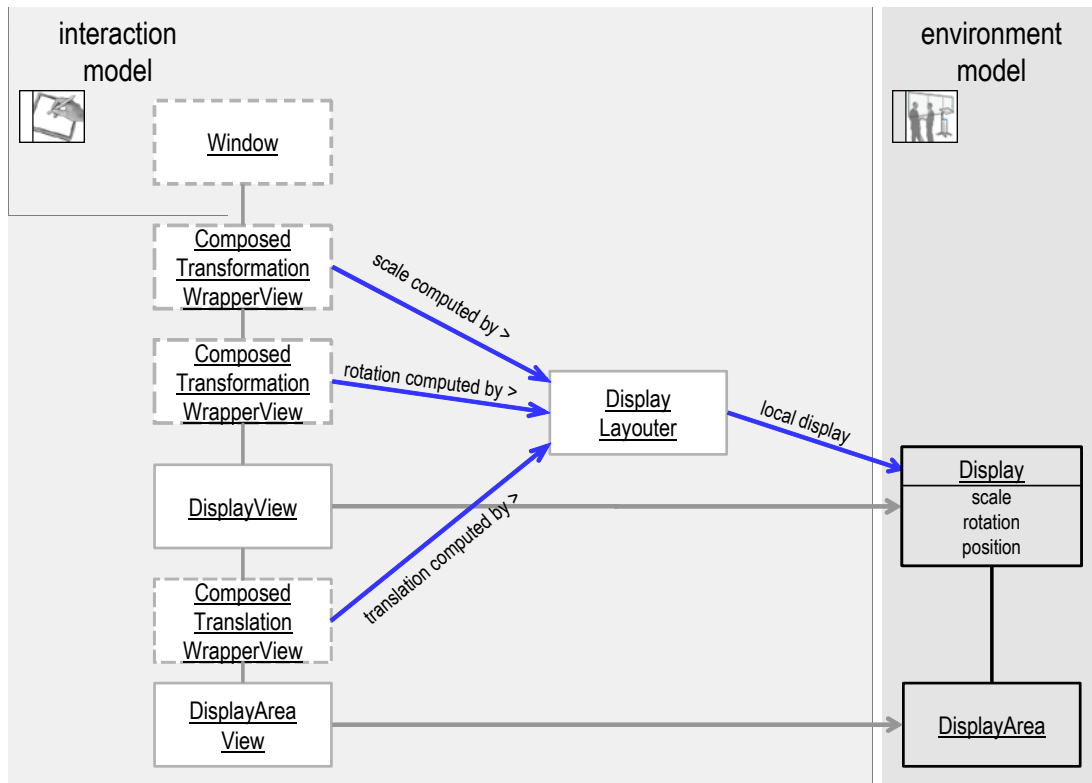
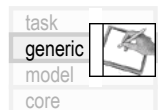


Figure 7-17: Assigning the position within a display area to the local display view. In addition to figure 7-16, the three transformation wrappers shown in this diagram adapt their sub-views' local coordinate systems. (Unlike figure 6-20, the display layouter is shown here instead of the transformation objects.)



## 7.5. Generic Interaction Model: User Input to Roomware Components

Besides adapted visualization, roomware components also need specific support for user input (req. H-1). This section explains:

- *event dispatching strategies* for the devices implemented at the generic level,
- recognition of *pen gestures*, and
- support for *multi-user devices*.

### 7.5.1 Event Dispatch Strategies for Default Devices

As mentioned in section 6.2.1, different devices produce different kinds of events that might need to be handled differently. Module `BEACHdocument` therefore defines a dispatch strategy that uses the focus holder (section 7.2.2) associated with the device that originated an event as the controller for this event (fig. 7-18). In addition, the events generated by mouse or keyboard are defined. Mouse events use the `dispatchToPoint` dispatch strategy (section 6.8.1), while keyboard event use `dispatchToFocusHolder`.

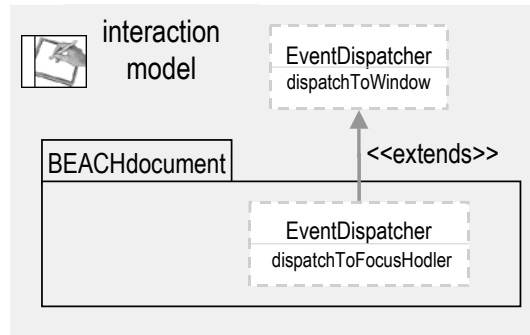


Figure 7-18. Class EventDispatcher is extended with a dispatch strategy for focus-oriented input devices, such as a keyboard.

### 7.5.2 Gesture Recognition

Gestures are often used to enable natural interaction with pens (req. H-1)—especially for roomware components having no mouse or keyboard as input devices. To be able to invoke commands using gestures, the raw input events have to be assembled and interpreted. The interpretation of the user input is difficult, as it can be ambiguous leading to recognition errors if the wrong interpretation is chosen (Mankoff *et al.*, 2000b). This reduces user acceptance and effectiveness. In modeless user interfaces (see section 7.2.2), correct interpretation is especially important, as the shape is used to distinguish between commands and scribbles.

To generate the gesture events needed to handle pen input, each stroke that is drawn is sent to a gesture recognizer to check whether it is similar to one of the set of supported gesture shapes (see also section 3.2.2). Using BEACH for our work and for presentations, we found that the writing speed is crucial for fluid interaction. This led to an incremental recognition algorithm, similar to the one proposed by Rubine (1991). In addition, an incremental algorithm allows continuous feedback to the user of whether a gesture is currently recognized, similar to the idea published in (Arvo and Novins, 2000). Immediate feedback is very helpful, as gesture recognition is inherently error-prone. Thus, a very fast recognition algorithm helps improve the usability of the system by giving feedback about an action before the action is executed.

↓ Section outline

This section explains the incremental recognition algorithm, the design of the gesture recognizer, and how the recognition process works.

#### Incremental Recognition

Example 7-5:  
Immediate feedback

To illustrate the immediate feedback, the example of a user drawing a box shape is used (fig. 7-19). For simplicity, the gesture recognizer used in this example only supports three different shapes: line shape, “L” shape, and box shape.<sup>34</sup> The user starts drawing a line (a), which is recognized as a known shape. To give feedback about the recognized shape, the stroke turns red. When the user reaches the first angle (b), the color changes again to the current scribble color. As an “L” shape has a minimum length of both line segments (c), the “L” shape is recognized until the user comes to the next turn (d). Finally, when the end-point comes close to the start-point again, a box shape is identified (e).

<sup>34</sup> The actual recognizer in BEACH uses a shape set of currently 15 different shapes: box, circle, coil, double-box, double-circle, double-X, encircle, line, L, spike, U, X, and the special shapes: tap, scribble, and abort. The special shapes are explained below.

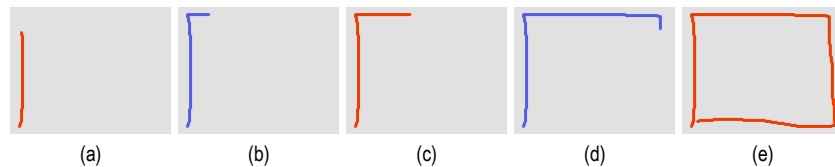


Figure 7-19: Intermediate steps while a user is drawing a box shape. If a shape is recognized the color changes to red (a, c, e).

### Design of the Recognizer

Figure 7-20 shows the objects involved in the recognition process. Class `Recognizer` is responsible for transforming a sequence of pen (or mouse) positions into a set of matching shapes. It uses a set of `ShapeFeatures` that are computed from the sequence of points to determine a set of matching instances of class `Shape`. While it is active, the `StrokeTracker`, used for drawing strokes with a pen, receives events directly from the event dispatcher (see section 6.2.1).

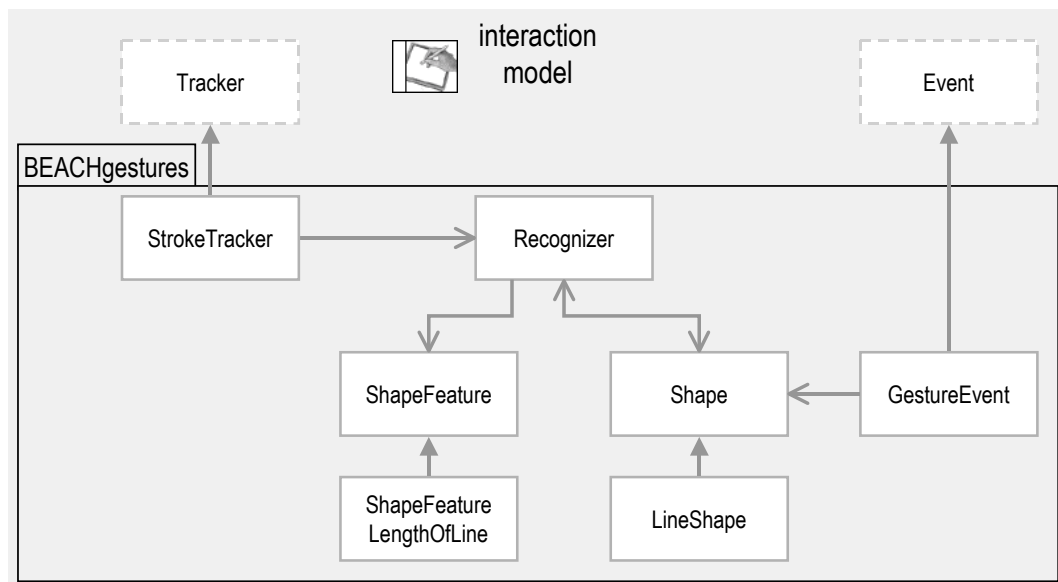


Figure 7-20. Design of the gesture recognition module

### Recognition Process

The recognition process is illustrated in figure 7-21. When the stroke tracker is started (upon a pen down event), it creates a new instance of the recognizer. While the stroke tracker is active, besides drawing and recording the stroke, it sends all points immediately to its recognizer (1). When the recognizer receives a new position, it first updates all of its features (2). The features used in this example (see fig. 7-22) are *key-points* (i.e. the corners of a stroke), the *length* of the stroke, and the *bounding box* (i.e. the minimal box containing the stroke). Based on the values of the features, the recognizer checks all shapes possibly matching, whether they match or not (3). For example, a line shape needs to have two key-points, and the distance from the first to the last point should be similar to the length of the line feature.

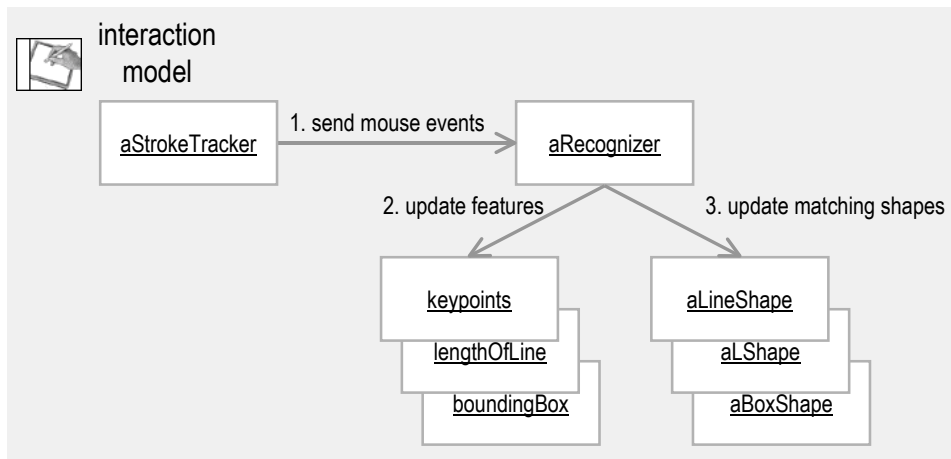


Figure 7-21: The recognizer receives mouse events from the tracker and incrementally updates the features and the list of matching shapes.

After all shapes are updated, the tracker checks whether the currently most likely shape has changed. In this case, it asks the new shape for its associated stroke color and redraws the stroke. Finally, when the stroke is finished (pen up event), the tracker creates a *gesture event* associated with the final shape and passes it to the event dispatcher for further processing.

#### Incremental Update

While the first implementation recomputed all features every time a new point was added to the stroke, we determined that this was too slow to be acceptable. Therefore, the recognizer was changed to update the features incrementally: The recognizer now sends every new point to every feature, which will merge the new value with the previous state. Each feature can decide whether to update or re-compute.

Example 7-6:  
Feature re-computation

- The *length of the stroke* is computed by summing the distances between all points. It is much faster only to add the new distance.
- The *bounding box* needs to check whether the new point is outside the previous box and enlarge if necessary.
- To compute the *key-points*, it is necessary to look for turns in the stroke. As the first and last points of the stroke are always key-points, the incremental update is not as easy as in the previous examples. Depending on whether the last point is a corner or not, the last key-point is replaced (see fig. 7-22 b–c).

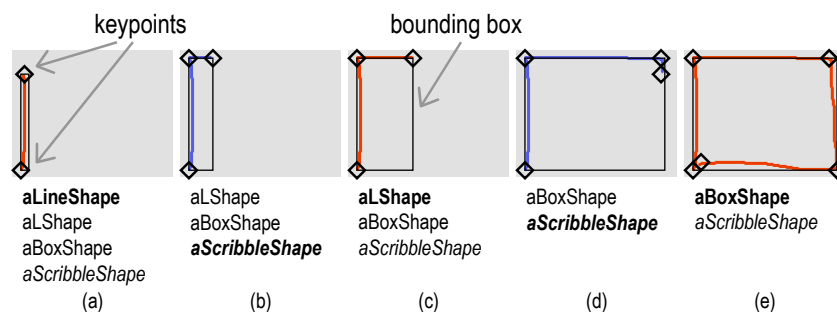


Figure 7-22: The features are updated incrementally while the stroke is drawn, if shapes cannot match anymore, they are removed from the recognizer.

With the updated values, the recognizer asks all shapes for their current similarity to the stroke. If a shape returns zero, e.g., because the stroke has too many key-points, the recognizer removes it from the list of possibly matching shapes. This is illustrated in figure 7-22 at the

transitions a–b and c–d. The scribble shape is the default shape and will never be removed from that list.

#### Special Stroke Shapes: Scribble, Tap, and Abort Shape

The BEACH framework defines three special shapes that are handled specially.

- The *scribble shape* is the default shape that matches any stroke that does not match any other shape.
- The *tap shape* is the only shape that consists of only one key point. It is the equivalent of a mouse click in pen-based interaction models.
- The *abort shape* was introduced to make use of the immediate user feedback. As the name suggests, the abort shape is never handled by any controller. Thus, it enables the user to cancel a shape that was not recognized. An abort shape can start with any other shape. At some point, the user can encircle this point several times. This allows turning any shape into an abort shape.

#### Dispatching Gesture Events

When the stroke is finished (pen up event), the tracker creates a *gesture event* associated with the final shape and passes it to the event dispatcher.<sup>35</sup> In contrast to mouse events that refer to a specific point (see section 6.8.1), a gesture event is associated with a stroke — which could cross the bounds of multiple view objects. Therefore, BEACHgestures provides a dispatching strategy (see section 6.2.1) for gesture events (fig. 7-23). Depending on the shape, it asks the controllers of all views that intersect with the shape or that contain the shape whether they want to handle this shape. The controllers are asked in the reverse order in which their views are displayed. The event dispatcher sends a `handleShape:` message to the first controller signaling interest to handle this gesture event.

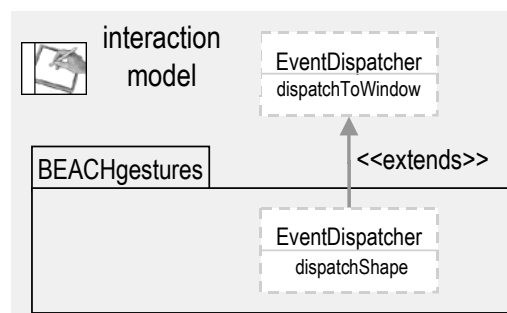


Figure 7-23. Dispatching strategy for gesture events

#### 7.5.3 Support for Multi-user Devices

For multiple-user devices (req. C-3), it is necessary to provide an interface to hardware that is capable of handling multiple users at the same time using the same device.<sup>36</sup> Multiple device

<sup>35</sup> The abort shape is dispatched just like any other shape. This enables applications to provide user feedback if a shape is aborted. However, a controller must not invoke any command when handling an abort shape.

<sup>36</sup> Currently, the available hardware is quite limited concerning concurrent direct user input at the same visual interaction device. Wacom's Intuos graphics tablet can trace two interaction devices (like pens or mice) at the same time (Wacom Technology Co., 2003), but it provides no output facilities. DiamondTouch (Dietz and Leigh, 2001) is a research prototype of a table allowing concurrent input of multiple users sitting around the table. SMART technology (SMART Technologies Inc., 2002) is also currently working on a new version of the SMART board, being able to detect up to four people

drivers can send events, tagged with an identification of the originator, to the BEACH framework.

The device information attached to the events can be used at several levels. On the one hand, it is needed to track concurrent sequences of input events as mentioned above. On the other hand, it could also be used to map the device to the user that is actually using it (either manually or using available context information). This enables truly user-aware multi-user systems, as this information can be, e.g., associated with created artifacts as additional context meta-data to ease later retrieval, or it can be used to automatically select the user's preferences. This is especially helpful, if personalized settings for gesture recognition can be used.

Module `BEACHmultiuser` extends the event dispatching defined at the core level (see section 6.2.1). It defines class `MultiDeviceEventDispatcher` that is capable of tracing several concurrent sequences of input events and directs them to the appropriate tracker (fig. 7-24). In order to identify the device that originated an event, class `Event` is extended with an association to the device that created the event.

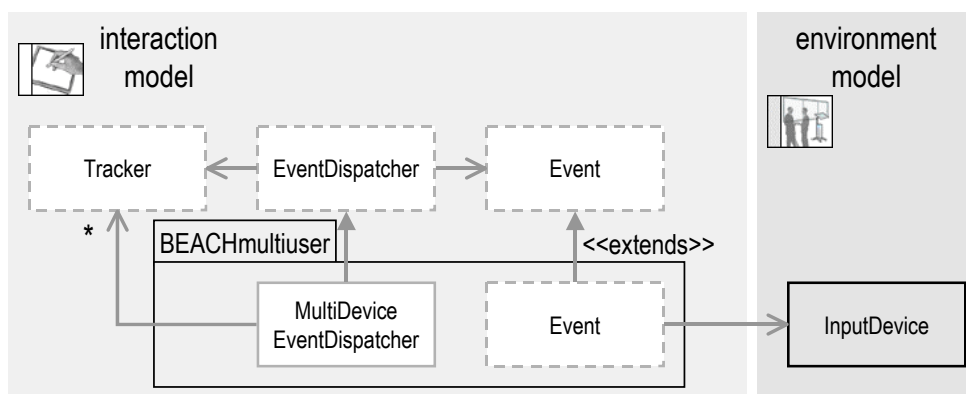


Figure 7-24. The multi-device event-dispatcher can manage several concurrent trackers. As the tracker receives events from the dispatcher, it can be reused directly and does not need to be extended.

## 7.6. Example Device Configurations

To illustrate how the BEACH framework can be used for different devices and configurations, this section gives three examples. The examples are based on the application scenarios defined in section 2.2, showing how different collaboration situations can be modeled with the BEACH framework. However, they can serve as samples for quite generic collaboration situations.

The examples show objects defined by the generic layer only (except class `Station`), as this layer defines the concrete classes used to implement generic support for roomware components.

- First, the `DynaWall` is an example for a multiple-computer device (req. U-2).
- Second, the collaboration between a large public and a smaller private device (req. UC-1) is shown in the case of a `DynaWall` in conjunction with a `CommChair`.
- Third, the `InteracTable`, an interactive table, is used to demonstrate a device that can be used by multiple users at the same time, but with different viewing preferences (req. UH-1, C-3).

(Martin *et al.*, 2002). As a workaround, multiple mice or SMART boards can be connected to one computer. Recently, Rekimoto (2002) has presented a research prototype of an interactive surface that is capable of detecting multiple hand positions using capacitive sensing.



These examples focus on the four shared models, leaving the interaction model aside to reduce complexity. This causes no loss of relevant information, as the interaction model does not have its own state; its state can be completely computed from the state of the other models. Another configuration showing the relationship of the shared models to the interaction model using the example of *ConnecTables* is presented in the next chapter (section 8.2) and in (Tandler *et al.*, 2001).

### 7.6.1 Combining Multiple Computers to One Interaction Device

As mentioned before, the *DynaWall* (fig. 7-2) consists of three computers (multiple-computer device, req. U-2), each with an attached SMART Board (SMART Technologies Inc., 2002). Therefore, the environment model defines the roomware component “*DynaWall*” (*DisplayArea* in figure 7-2) to consist of three stations (*Station1* to *Station3*) with their displays combined to one large display area.

Using the view transformations (see section 6.8.3) a view is opened for each display showing a part of the complete display area. (See also the example given in section 7.4.)

While the three SMART Boards in our lab are mounted to one wall (fig. 7-2 on page 131), the software allows changing this setting dynamically (req. UH-3, U-4). This is useful when the boards are mobile and equipped with sensors such that the presence of other boards can be sensed automatically (similar to the *ConnecTable*, see section 8.2 and (Tandler *et al.*, 2001)).

---

Example 7-7:  
Display area; view  
transformations

### 7.6.2 Tight Collaboration Using Two Different Devices

When a *CommChair* connects to the *DynaWall* specified in the previous example, it is interesting to see, on the one hand, how the tight collaboration between *CommChair* and *DynaWall* is implemented (req. UC-1). On the other hand, it shows how the *CommChair*'s display area is separated into two segments for public and private workspaces.

The display area, which consists of only one display in the case of a *CommChair* (*CommChair* in fig. 7-25), is split into two segments (*CCPublicSegment* and *CCPrivateSegment*). While one segment is used to show a document browser (*CCDocBrowser*) for the private workspace (*Workspace2*), the other connects to the document browser shown at the *DynaWall* (*DWDocBrowser*). This enables very tight collaboration, as using a shared browser results in coupled navigation. Since the same application model is always used, all editing state is also shared between the *DynaWall* and the *CommChair*, which allows providing awareness information (Schuckmann *et al.*, 1999).

---

Example 7-8: Shared  
document browser

As the size of the segments at the *DynaWall* and the *CommChair* differs, the interaction model (not shown in figure 7-25) has to provide an appropriate mechanism for displaying the public workspace at the *CommChair*. Established techniques are scrolling and/or zooming.

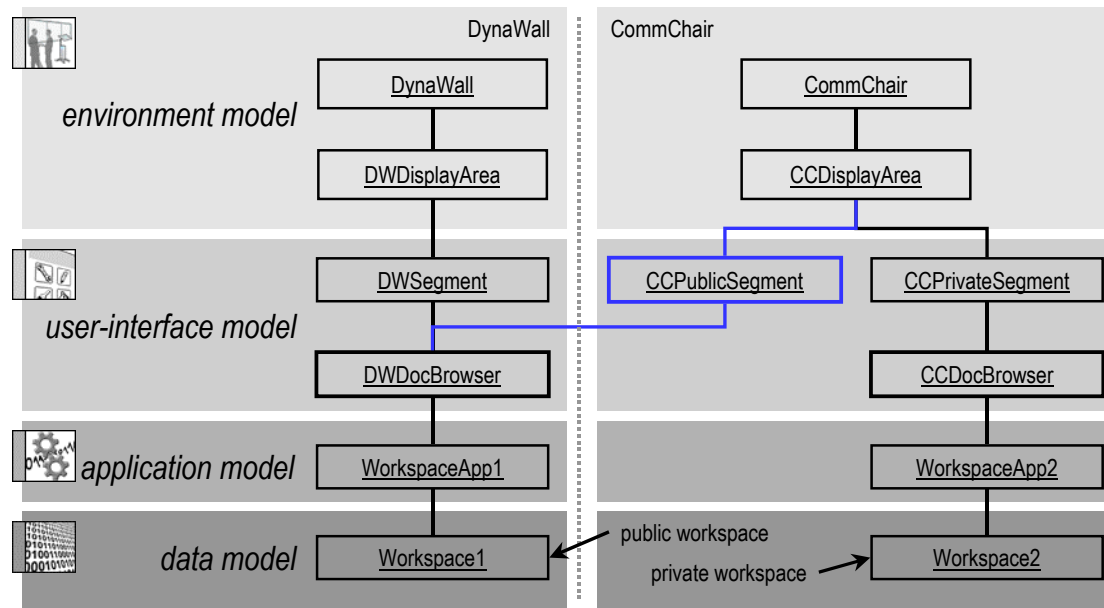


Figure 7-25. The CommChair splits its display area into two segments. One connects to the document browser shown at the DynaWall; the other is used to show a private workspace.

### 7.6.3 Multiple Users Collaborating at One Device

Example 7-9:  
Multiple views on  
one document

While the CommChair in combination with a DynaWall enables collaboration of users with different devices, the InteracTable aims to support several users at the same device (req. C-3). As mentioned, it is necessary to provide adapted orientation of visualizations for multiple users around a horizontal display (req. UH-1).

To realize this, we used an approach where each user can open an overlay that can be moved freely around the display area, similar to a window (e.g. `ITOverlay1` and `ITOverlay2` in figure 7-26). Each overlay gets a separate document browser and workspace application model, but remains connected to the same workspace. In this case, the interaction model will open two views showing the same workspace. This allows each user to rotate his or her workspace to the preferred direction, as the rotation is specified by the workspace application model.

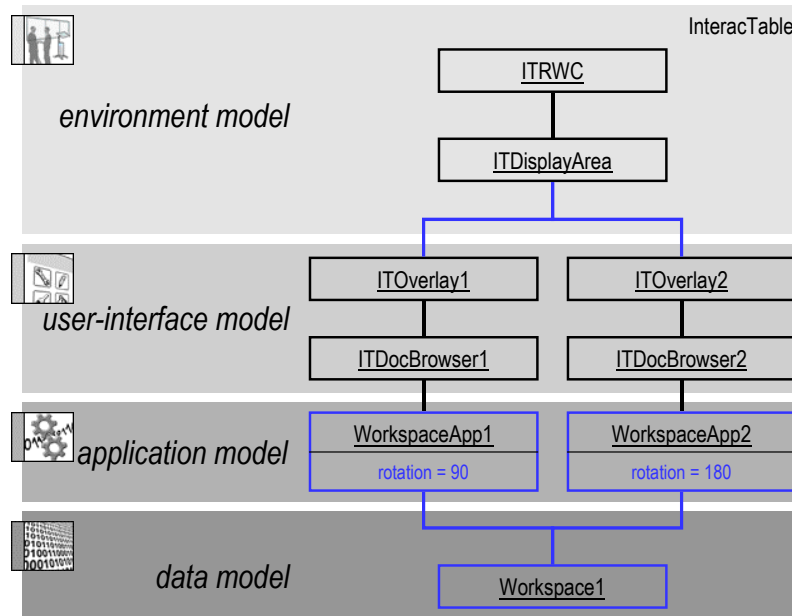


Figure 7-26. Two users working at an InteracTable with the same workspace. By using separate browsers with separate application models, each user can look at the workspace with the preferred orientation. (The figure is simplified omitting the wrappers.)

### 7.7. Discussion: Properties of the BEACH Generic Collaboration Framework

This chapter presents the BEACH Generic Collaboration framework providing generic elements to support **synchronous collaboration** with roomware components. The focus is on **informal meetings** and collaboration situations. It implements the *generic layer* of the BEACH architecture.

The *environment model* adds a simple **model of roomware** components. The **display area** is an abstraction to combine several displays to form a homogeneous interaction surface. This makes the visual presentation independent of the physical setup of display devices and allows dynamic reconfiguration of the display setup. The **display** illustrates how new interaction devices can be defined. Using the display's device service, which is provided by the model layer, the display user interface and the interaction application can be created and opened on the (physical) display.

The *data model* provides typical **document element types**, such as links and anchors, for a **spatial hypertext** model. Spatial hypertext is appropriate for informal meeting situations. The relationship between an element and its container is modeled as a **position wrapper** to decouple element and container, thus raising reusability. The position wrapper is an example of a relation wrapper.

The *application model* aims at supporting **modeless interaction**. To enable **throwing of document elements** across large displays, the application model of the position wrapper holds the information needed to determine the current position while thrown. This is an example of how editing state can be added by the application model. By extending the default application model of the position wrapper, it is possible to throw any document element within any container as long as a position wrapper is used to place the element inside the container. To implement the animated throwing, **synchronized time-based invalidation** had to be implemented.

The *user interface model* defines the **display user interface application**, which is the default application, opening a full-screen window to represent the display of a station. As this is a very

fundamental task, it is interesting to note that this is not implemented as core or model level functionality. By using the device hook, which is defined at the model level, the display and functionality for visual interaction and a visually-oriented user interface can be added at the generic level. This demonstrates that the BEACH Model framework can be extended to support new interaction devices and forms. Another example of using the device hook is presented in section 8.3.

As generic user interface elements to structure the available interaction space, **overlays and segments** are defined as concrete specializations of the abstract visual interaction area. The user interface model extends the display area to refer to the visual interaction areas that are currently displayed. This is an example of how the **user interface is integrated with the physical environment** (req. U-3). As the display area reflects the physical configuration (being defined by the environment model), modifications of the display setup represented by the display area will cause changes in the user interface.

The root **toolbar** provides a **hook for modules** to plug in module-specific functionality. This way, new functionality that is defined by modules can be made available in the user interface.

A **document browser** is used to navigate between workspaces. Its history remembers the visited application models (in contrast to storing data models), in order to be able to restore the complete editing state when navigating back. Different **collaboration modes** can be distinguished depending on whether users share browser, application model or data model.

The *interaction model* provides the **display interaction application**, which is the corresponding part of the display user interface application in the interaction model. It actually opens a window containing a view hierarchy, showing the part of the display area currently visible on the local display. The **display layouter** is responsible for calculating the correct bounds and transformation for each display within the display area. It can be used as a hook for extending the supported display configurations, which might be necessary for future roomware components.

The views' **dependency mechanism** is explained by an example view hierarchy. It shows how the presentation can be connected to shared models, while being able to adapt the presentation to the local context. This is necessary to generate the different presentations for all displays that are combined by the same display area. Using constraints also ensures that changes made to the display setup are immediately reflected in the visual presentation.

To provide an appropriate input for roomware components, the interaction model implements an **incremental gesture recognition** algorithm, which allows giving **immediate feedback** about recognized shapes to the user while drawing a pen stroke. The abort shape makes use of the immediate feedback by enabling users to **cancel misinterpreted gestures** before they are executed. Pen input is an example of how another interaction form can be supported using the BEACH Model framework. A new *tracker* is defined that assembles pen events into pen strokes. The event dispatcher is extended by a new **event dispatching strategy for gesture events**, and the controllers are extended to handle gesture events.

**Multi-user support** is needed for roomware components supporting concurrent input at a single device. The BEACH framework defines a new *event dispatcher* that is capable of handling concurrent input sequences coming from different devices. However, the defined trackers need not be adapted, as the event dispatcher is responsible for sending events to the trackers and the tracker does not have to be aware of other active trackers.

The last part of this thesis presents applications of the BEACH model, architecture, and framework. Chapter 8 presents examples where BEACH has been extended to support new forms of interaction. Chapter 9 explains some tools that have been developed on top of the BEACH framework.

---

↓ Next part

## Part III. Applications for Roomware Environments: Validating the Usability of the BEACH Model and Framework

---

The last part of this thesis shows applications of the BEACH conceptual model and software framework. By analyzing these applications, it can be shown that the BEACH conceptual model helps create a clearly structured, reusable and extensible software design. The applications validate the usability of the model and framework and prove that they ease application development. In addition, they illustrate how their features can be applied. Chapter 8 presents examples where BEACH has been extended to support new forms of interaction. Chapter 9 explains some tools that have been developed on top of the BEACH framework. These two chapters illustrate that new interaction forms and tools can benefit from the reusable design and implementation. Up to now, more than 15 software developers have used the BEACH model and framework to create 12 tools and extensions. This found that the BEACH model helps identify reusable parts of a software system to be developed. For one example, it is shown that the implementation effort in terms of lines of code is reduced to less than one third. Chapter 10 finally presents conclusions, also discussing open questions and directions for future work.

---



## 8. Extending BEACH for New Forms of Interaction

---

This chapter presents three examples of devices for which new interaction styles have been implemented on top of the BEACH framework. The styles have been implemented as BEACH modules. The examples are: the Passage mechanism, the ConnecTables, and audio feedback. The Passage mechanism is an example of an interaction style using *physical objects*. The ConnecTables show how to dynamically *re-configure the environment model* of roomware components. The audio module provides an *audio-based interaction style*. The implementation of the Passage mechanism reveals that the BEACH model helps *identify reusable parts*. Two modules originally developed for Passage provide a major part of the functionality needed for the ConnecTables. The implementation of Passage using the BEACH framework also shows that the *implementation effort* in terms of lines of code compared to a previous implementation based on state-of-the-art technology is reduced by a factor of more than three. Finally, the chapter discusses the experiences of extending the BEACH framework to support new interaction forms. It is found that the BEACH framework flexibly supports combination of interaction modalities and distribution of interaction and interaction devices.

---

The purpose of the BEACH framework presented in the previous chapters is to ease the development of applications for a roomware environment, while offering a platform that allows incorporating new devices that require new forms of interaction.

This chapter presents three examples of devices for which extensions have been implemented on top of the BEACH framework. The extensions have been implemented as BEACH modules that belong to the task layer, but also introducing new interaction models at the model layer, and default implementations at the generic layer. The examples are:

- The Passage mechanism to transport digital information using *physical objects*.
- The ConnecTable as an example how to *dynamically re-configure the environment model* of roomware components triggered by physical interaction.
- Audio feedback as a *different form of interaction*.



These interaction forms have been cooperatively developed by the members of the AMBIENTE team with major participation of the author of this dissertation. The software design has been created by the author alone. The implementation was done by the author and other team members.

To illustrate how applications can be constructed with the help of the BEACH framework, the next chapter presents several applications that have been built on top of BEACH.

### 8.1. Passage Mechanism: Interaction with Physical Objects

The first BEACH extension that has been implemented is the passage mechanism. An earlier implementation was published in (Konomi *et al.*, 1999). This section first gives an overview of the functionality of the passage mechanism. It is followed by a description of the interaction model that is used to access sensors and a description of the supported sensors. Then, the actual passage system is presented.

#### 8.1.1 Functionality of the Passage Mechanism

*Passage* describes a mechanism for establishing relations between physical objects and virtual information structures, i.e., bridging the border between the real world and the digital, virtual world. So-called *passengers* (passage-objects) enable people to have quick and direct access to information and to use the objects as “physical bookmarks”. It provides an intuitive way for the “transportation” of information between roomware components, e.g., between offices or to and from meeting rooms.

A passenger does not have to be a special physical object. Any uniquely detectable physical object may become a passenger. Since information is not stored on the passenger itself but only linked to it, people can turn any object into a passenger: a watch, a ring, a pen, glasses, or other arbitrary objects. The only restriction passengers have is that they can be identified by the “bridge” and that they are unique. The “bridge” is a device attached to a roomware component that is capable of detecting passengers. Figure 8-1 shows a key-chain as an example of a passenger. The passenger is placed on a dark area of the InteracTable representing the real part of the “bridge” device that is embedded in the margin of the InteracTable. When a passenger is placed on a bridge, its virtual counterpart is made accessible. With simple gestures, the digital information can be assigned to or retrieved from the passenger via the virtual part of the bridge.

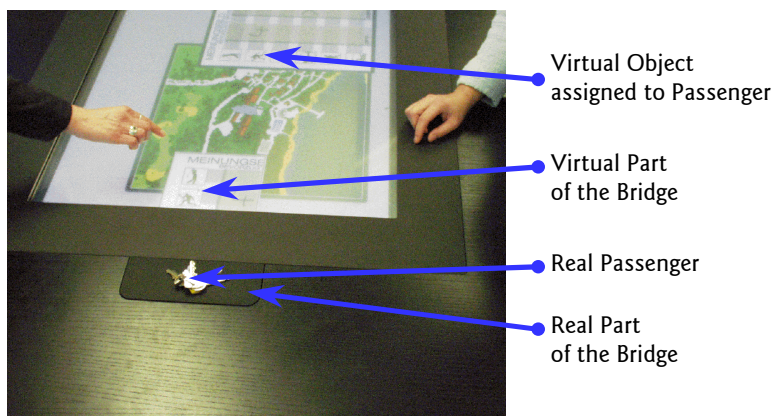


Figure 8-1. Passage: a key-chain as a passenger object on the bridge of the InteracTable. The interface area in the front of the display represents the virtual part of the bridge.

We developed two methods for the detection and identification of passengers. The first method allows the use of arbitrary objects without any preparation or tagging. Here, we use a



very basic property of all physical objects—the weight—as the identifier. Therefore, each bridge contains an electronic scale for measuring the weight of the passengers. The second method uses a contact-free identification device using radio-frequency-based transponder technology, which is built into every bridge. Small electronic identification tags (RFID) that do not need batteries are attached to or embedded in physical objects so that the passenger can be identified by a unique 32-bit ID. The identification via the weight of an object provides greater flexibility and aims at short-term assignments. In contrast, the electronic tag method provides higher reliability and is useful for long-term assignments—at the expense of requiring some preparation of the objects.

#### 8.1.2 Passage and Sensor Architecture Overview

The handling of sensors is not a task unique to the passage system. According to the BEACH model, the system was therefore divided into four modules at three levels to allow easy reuse of sensor-related functionality. Figure 8-2 gives an overview of the resulting system architecture. Module `BEACHsensormodel` defines an environment and interaction model for sensors. Consequently, this is placed at the *model level*, as this defines the basic abstractions for interaction mediated by sensors, analogous to what the view model is defining for visual interaction. The sensor model is used by the two modules implementing the interface for the scale and the tag reader, respectively. These modules belong to the *generic level*, as the interface to the sensors is independent of any task or applications. These two are the parts that are good candidates for reuse.

The `BEACHpassage` module, then, provides the *task-specific* part of the Passage system. The task, in this case, is to transport digital information using physical objects. It needs three main abstractions that are unique to this task. First, the *bridge* is the new part of the user interface that allows assigning information to objects. Second, the *passenger* is the digital representation of a physical object that has been detected by a sensor. Third, the *bridge observer* is an interaction model that allows detecting passengers that are placed on the bridge and provides the interaction for assigning information to it. The `BEACHpassage` module uses the sensor model for interfacing with information. This way, it is possible to use arbitrary sensor implementations for the passage system.

## 8. Extending BEACH for New Forms of Interaction

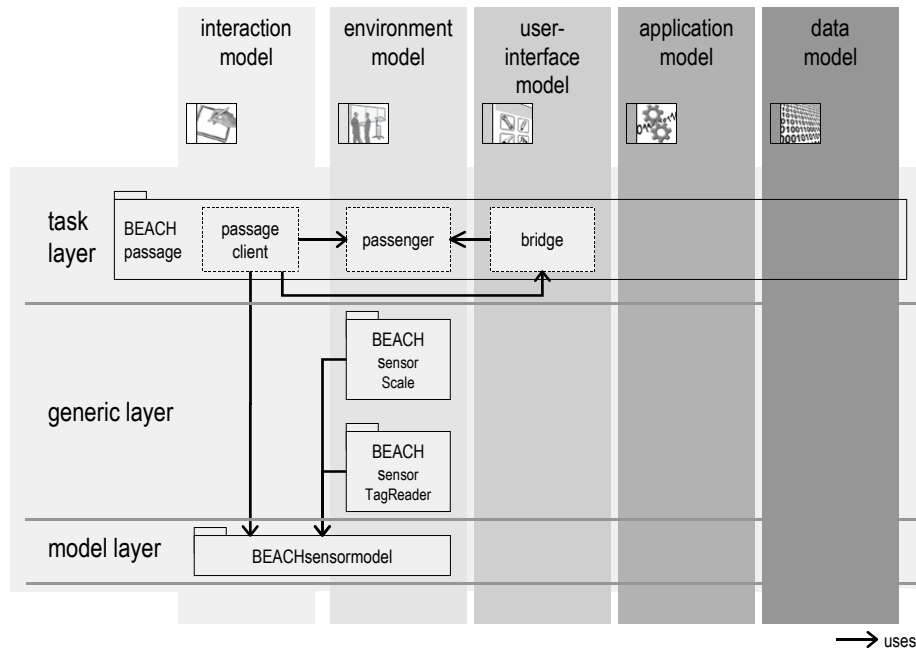


Figure 8-2. Passage architecture overview. A new interaction model is defined to allow physical objects act as part of the user interface. Sensor-related functionality is placed in separate modules to allow reuse of these components.

### 8.1.3 Interaction Model for Physical Context-Information

Similar to the interaction using displays and pointing devices that is supported by the `BEACHviewmodel` module, the interaction with sensors requires an appropriate interaction model as well. Module `BEACHsensormodel` defines base-classes for both the environment and interaction model (fig. 8-3, classes added for the Passage implementation are marked by a diagonal background pattern). Abstractions for the sensors and the actuators represent the interaction devices as part of the environment. Sensor observers and actuator controllers are responsible for the interaction.

Class `Sensor` extends class `InputDevice`. If a sensor detects a physical object, the software representation of this sensor will set its value to an object representing the physical object. For example, a tag-reader will use the identification of the sensed tag. A `SensorObserver` watches the sensed value and triggers actions whenever the sensed value changes. As the sensor is designed to be a shared model, it is possible for sensor observers to access values sensed by sensors that are connected to remote computers. To handle the communication with the sensor hardware, each sensor (like any other device using the device hook, see section 6.6) is notified on system startup and shutdown. Then it can start a process to monitor the sensor.

Classes `Actuator` and `ActuatorController` are not used by the Passage system. They are included for completeness of the sensor interaction model. Actuators are physical objects that can be controlled by software (Greenberg and Fitchett, 2001; Greenberg and Boyle, 2002). However, currently there is no module actually providing a driver for actuators for the BEACH framework.

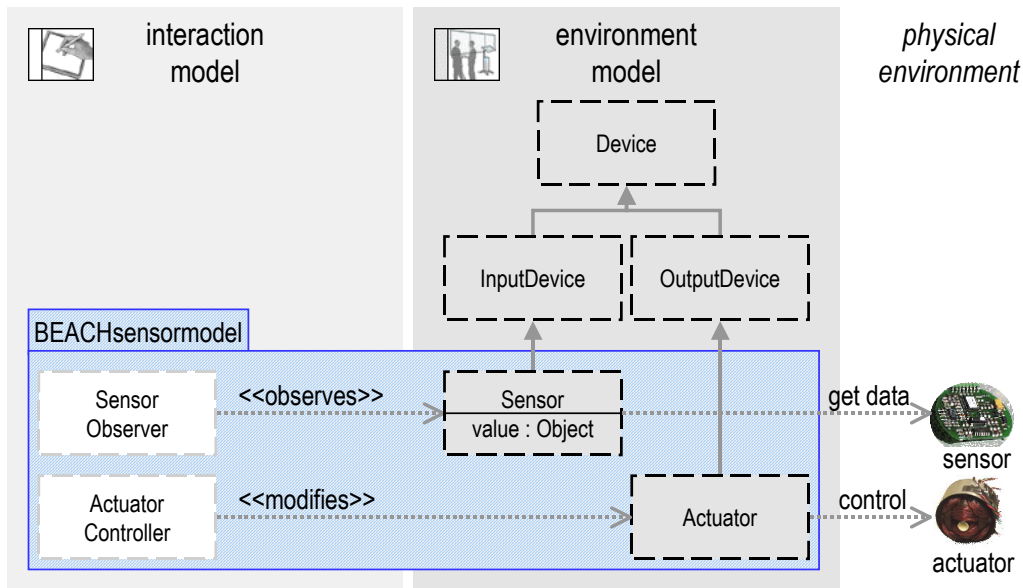


Figure 8-3. The sensor interaction model supports physical objects and context information as part of the user interface. Classes added for the Passage implementation are marked by a diagonal background pattern.

#### 8.1.4 Sensor Management

Two concrete sensor classes have been implemented for the Passage system: class *ScaleSensor* and *TagSensor*.

Class *ScaleSensor* (fig. 8-4) undertakes the communication with an electronic scale that can be connected to a PC via the serial port. The scale sensor starts a process that repeatedly queries the currently detected weight in a defined interval. If the sensed weight changes, the value attribute of the scale sensor is set to the new weight (as a number). This makes the sensed value available to remote clients also, as the value is shared.

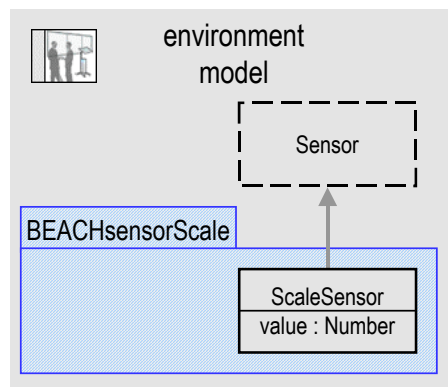


Figure 8-4. The scale sensor handles the communication with an attached digital scale. It represents the currently sensed weight as part of its object state.

Class *TagSensor* (fig. 8-5) defines the interface to the RFID tag reader used in the passage system. In contrast to the scale sensor, the tag reader is able to push values to the software when a tag is sensed. Therefore, the process started by the tag sensor need not poll for new values, but waits for new data from the tag reader. In addition, the tag reader module defines class *Tag*, as a representation of the RFID tags that can be attached to physical objects.

## 8. Extending BEACH for New Forms of Interaction

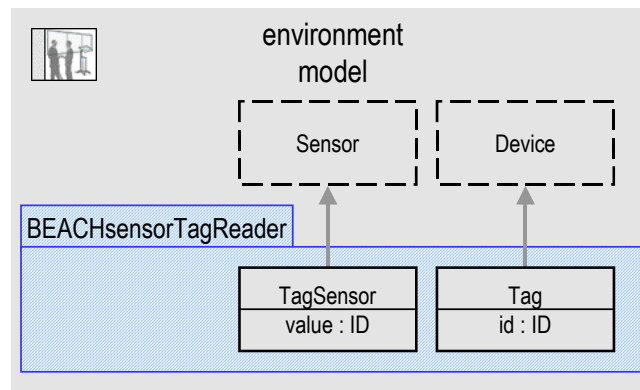


Figure 8-5. The tag sensor provides an interface to an RFID tag reader. The currently sensed tag ID is represented as part of its object state. The tag is a model of the tags that can be sensed.

### 8.1.5 Passage Module

The sensor model defined above is used by the passage module to realize the passage mechanism. Class `BridgeClient` is a sensor observer, which is notified whenever the sensed value changes (fig. 8-6). It retrieves the sensed value from its associated sensor and looks in the passage registry (class `PassageRegistry`) for a known passenger object (class `Passenger`) with the sensed ID. If none is found, a new passenger object with the sensed ID is created and added to the registry. Passengers must be able to reference the currently assigned application model. To enable this, the user interface model extends the passenger.

Finally, the bridge observer informs its associated bridge user interface (class `BridgeUI`) about the newly sensed passenger. As sensors, passengers, and bridges are shared objects, they can be easily distributed among different machines. The stations with connected sensors that run a bridge observer and the stations that open a bridge view can be configured arbitrarily. One station could run the device service querying the sensor for data, another could run the bridge observer watch the sensed value and inform the bridge, while still another shows the bridge view.

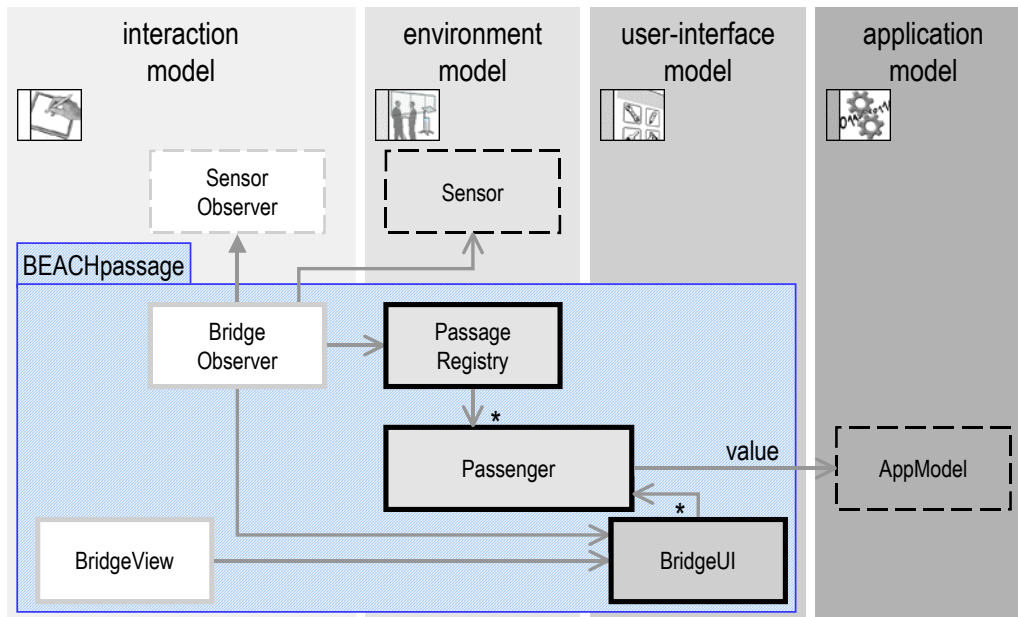


Figure 8-6. Design of the Passage module. The bridge observer observes an attached sensor. The passage registry manages all passenger objects that are in use. The bridge shows information attached to a detected passenger object.

When the bridge is informed about the passenger, the virtual part of the bridge (class `BridgeView`) is shown on the display. Class `BridgeView` opens a sub-view for the application model that is currently associated with the passenger (if any). In addition, it provides the possibility for the user to retrieve the document assigned to the passenger or to assign a different document to the passenger. Due to the dependency mechanism, the view is automatically updated if the sensed passenger or the assigned document changes (fig. 8-7).

As an application model can be assigned to a passenger (in contrast to a data model), it is possible to transfer a document including the current editing state via the passage mechanism. A user can therefore move to a different device and continue working.

## 8. Extending BEACH for New Forms of Interaction

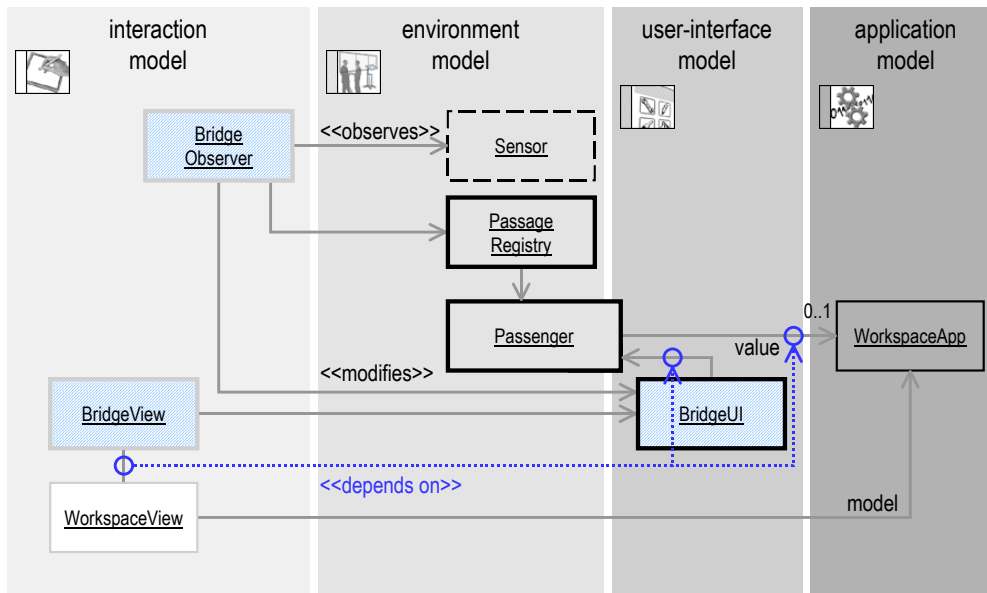


Figure 8-7. Passage interaction diagram. The bridge observer informs the bridge if a different passenger object is detected. The view of the workspace currently associated with the detected passenger is updated automatically whenever a different passenger is detected or a different workspace is attached to the passenger.

### 8.1.6 Analysis of the Passage Implementation

The BEACH model helped identify reusable parts that belong to the model and generic layer. Later, when implementing support for the ConnecTables (see next section) it was possible to directly reuse much of the code developed for Passage.

The implementation of the bridge is an example that illustrates the importance of the separation of input, output, and user interface issues. The interaction with the bridge combines two *different modalities*: a visual representation and manipulation of physical objects. In fact, this separation is the key to flexibly combining interaction modalities (what is called *flexi-modal* by Myers *et al.* (2002)).

As class `BridgeUI` is also shared, this separation enables *flexible distribution*. Multiple bridge observers (that act as controllers for the bridge) can be used if several sensors are integrated in order to broaden the range of detectable physical objects. Due to the shared user interface model of the bridge, the sensor can be connected to arbitrary computers. Any computer with a display that is in close physical proximity to the physical part of the bridge can be used to render the view of the bridge. Here, again, the view transformations (see section 6.8.3) can help orient the output generated by the view. The InteracTable shown in figure 8-1 has one bridge integrated near the left side of the display. To display the virtual part of the bridge oriented towards the left side, a rotation transformation is used. A scale transformation can scale the view of the workspace to fit inside view of the bridge, which has a fixed size.

### 8.1.7 Comparison of Implementations

The first implementation of the Passage system, which is described by Konomi *et al.* (1999), was not implemented using the BEACH model and framework. It was implemented from scratch by an experienced (Post Doc) computer scientist with no knowledge of the BEACH model or framework. It had an interface to the BEACH software prototype that existed then. The second implementation presented in this section was a complete re-implementation of the system. It was done by a graduate computer scientist trained using the BEACH model and framework. It is interesting to compare the two approaches.

Overall, the first implementation of Passage needed over 11.500 lines of code (including code to manage the sensors). Due to the reuse possibilities of the BEACH framework, this was reduced in the second implementation to less than 3000 lines of code<sup>37</sup> (again, including sensor management). However, these numbers can give a vague idea of the implementation effort only. As the first implementation used plain C and Perl, the latter Smalltalk, the development language might have an impact on the length of written code. Although the development time for the first implementation was not exactly tracked, it was developed during a period of several months. The first version of the second implementation was done in less than a week and required minor modifications only later on.

The first implementation used a Microsoft Access database and an Apache web server for the Passage registry. As the second version was completely built upon the BEACH framework, the Passage registry could be implemented as a single shared object. This reduced the code necessary to query the registry for sensed objects dramatically. For communication with sensor managers (i.e. the processes communicating with the sensor hardware), interested clients had to create a direct socket connection to the sensor manager process. As a consequence, all sensor managers had to be started before the clients, and clients lost the connection if a sensor manager died. In the reimplemention, the shared-object space was used as an indirect communication medium. In this way, sensor managers and clients can be started and stopped in arbitrary order. In addition, there is no latecomer problem, which has to be considered when shared events are used to notify about state changes only. As the current state is stored, any client can always access the most recent information.

## 8.2. ConnecTables: Dynamic Reconfiguration

The ConnecTable is a new component that was developed as part of the second roomware generation (Streitz *et al.*, 2001; Streitz *et al.*, 2002). It is designed for individual work as well as for cooperation in small groups. By moving two ConnecTables together, they can be arranged to form a larger display area (see figure 8-8). This is a natural way of establishing cooperation, as recommended in (Roseman and Greenberg, 1996b).

Integrated sensors measure the presence of another ConnecTable and initiate automatic coupling of the displays once they are close enough. When two ConnecTables' displays form a larger tabletop, the software provides a homogeneous interaction area encompassing both displays. The temporarily created common workspace contains all objects from the two previously separated workspaces keeping their positions and sizes constant (fig. 8-12). As an indicator for the users, the background color of the common workspace is changed.



Figure 8-8. (a) Three working-modes of the ConnecTable: connect, stand-up, sitting. (b) Two connected ConnecTables.

↓ Section outline

This section first presents the physical realization of the ConnecTables, followed by the design

<sup>37</sup> The lines of code do not count empty lines. For Smalltalk, the file out of the modules was counted.

of the software support. Finally, the dynamic view of the connection process is given. A more detailed discussion of the design issues of the ConnecTable can be found in (Tandler *et al.*, 2001).

### 8.2.1 Physical Realization of the ConnecTables

The pen-interactive display forming the tabletop allows a user to interact with information objects by pen. Currently, a Wacom PL-400 graphic tablet (Wacom Technology Co., 2003) is used. The ConnecTable's computer unit is based on an embedded PC board with a 266 MHz Pentium mobile CPU. The independent power supply for at least 3 hours is based on custom-built Nickel Metal Hydride rechargeable batteries integrated into the upper part of the chassis.

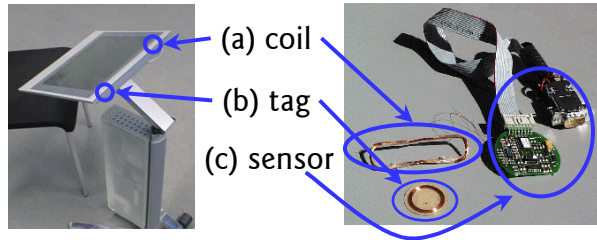


Figure 8-9. Sensor technology integrated in the ConnecTable. Coil and tag are placed left and right at the top of the display to detect other tables.

The sensing technology integrated into the ConnecTable's tabletop to detect others tables uses the same RFID tags as the passage system (see previous section). The right part of figure 8-9 shows the utilized components and the left part shows their location in a ConnecTable. A coil (see fig. 8-9a) establishes an energy field by emitting electromagnetic waves on a certain frequency. When a passive transponder tag (fig. 8-9b) enters the energy field emitted by the coil, it uses the induced energy to send back its unique 32-bit identification to the coil using another frequency. Signals that are received by the coil are forwarded to the sensor (fig. 8-9c) where they are processed. The identification mechanism starts at a distance of approximately 3 cm and takes 1.5 sec. The sensor connected to the coil communicates with the embedded PC using a serial interface.

### 8.2.2 Software Design to Support ConnecTables

To provide the functionality of connecting ConnecTables, the modules BEACHconnection has been implemented. It reuses the sensor interaction model and the tag reader interface that were implemented for the Passage system (see sections 8.1.3 and 8.1.4). Figure 8-10 shows the classes defined by BEACHconnection. The ConnectionModule starts a ConnectionClient and a ConnectionMonitor for the local station of each ConnecTable. This is possible as the module hook (see section 6.1.2) enables modules to add services implementing new functionality. The station is extended to allow a Connection between two stations. The CompositeRoomwareComponent is used to combine ConnecTables.



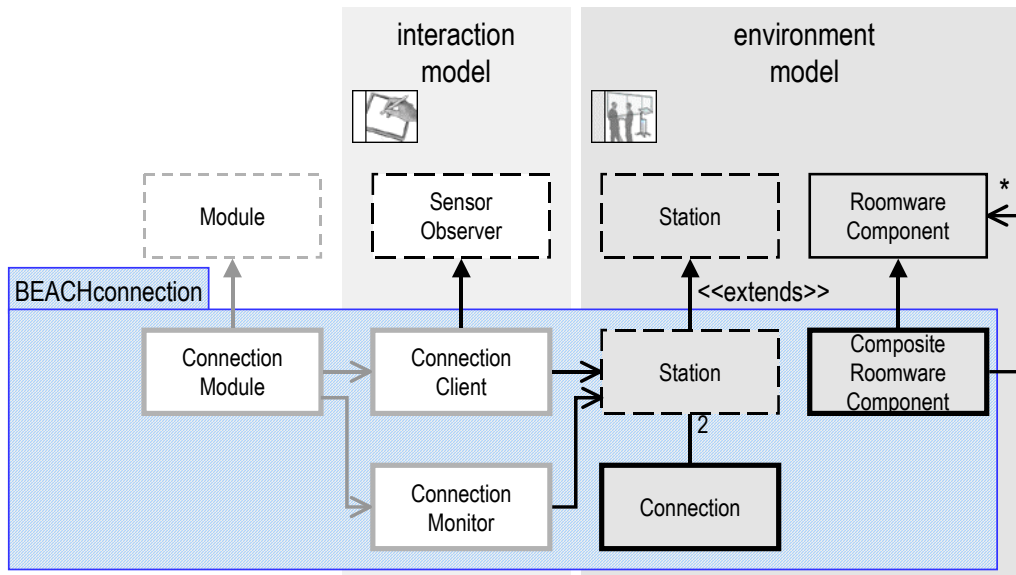


Figure 8-10. Design of the connection module. Connection clients watch for other ConnectTables detected by the sensors. The connection monitor combines two adjacent ConnectTables to form a logically combined roomware component.

If a new value is sensed by the sensor attached to a `ConnectionClient`, the `ConnectionClient` checks if the ID belongs to a tag attached with another station (fig. 8-11). In this case, the environment model is updated representing the currently connected stations. Introducing a new abstraction (the connection) that is tailored for the task (connecting tables) in the environment model at the task level enriches this model with semantically new information compared to the raw information detected by the sensors. The role of the connection client can thus be seen as an interpreter of context information in the sense of the conceptual framework as defined by Dey (2000).

The connections modeled in the environment model are unidirectional, representing “station A has detected station B”. This is important for two reasons. First, the connection objects can be created and modified simultaneously by the BEACH clients running at both involved stations; as they operate on different objects *no conflicting accesses* can occur. Second, as hardware sensors tend to have dropouts and failures, this provides *redundant information* about the status of the connection between two tables.

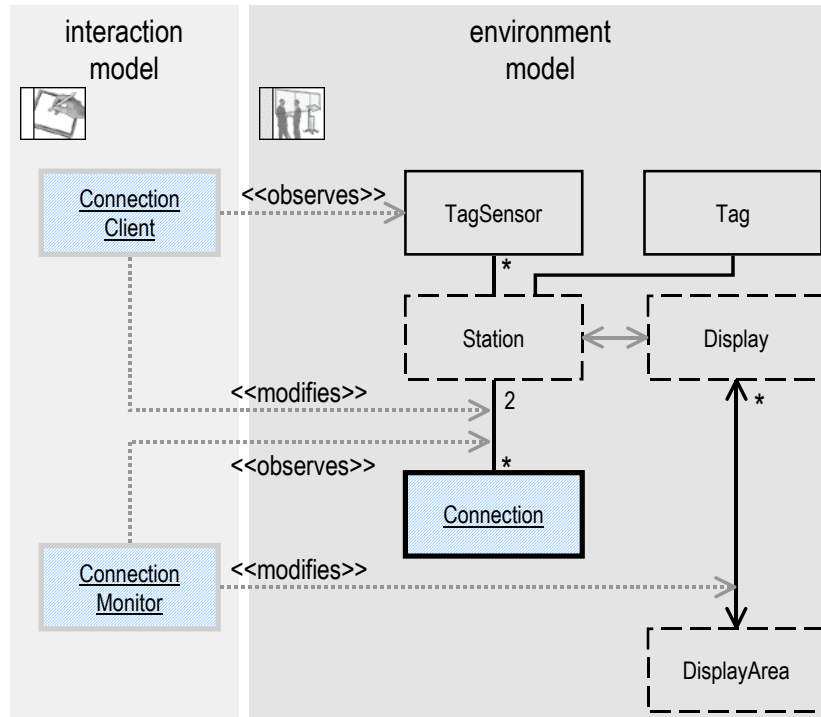


Figure 8-11. Sensors attached to a station. The ConnectionMonitor handles the connection if a sensor recognizes another ConnecTable. Connection client and monitor communicate indirectly by modifying the same station object.

Every change in the present connections between stations is detected by the `ConnectionMonitor`. The `ConnectionMonitor` then triggers the connection (or disconnection) process of the `ConnecTables`. Triggering the connection upon changes in the modeled connections among stations, rather than upon changes in the values sensed, offers additional flexibility.

When two `ConnecTables` are connecting, concurrency has to be taken into account. As every `ConnecTable` has one sensor attached, this normally results in redundant information about the connection, as explained above. To avoid both tables trying to change the same attributes of shared objects at the same time, only one computer should change the environment model of roomware components. Otherwise, the conflicting accesses as soon as they are recorded by the BEACH server would result in rollbacks and canceled transactions. Therefore, only the `ConnectionMonitor` running on the station with the smaller station ID executes the connection. However, having a `ConnectionClient` running on both stations providing redundant information about the sensed values raises the tolerance against hardware dropouts.

### 8.2.3 Connecting two ConnecTables

The client triggered by the connection monitor to execute the connection has the task of reflecting the new configuration of the roomware components in the environment model. This includes the adjustment of the displays and merging all workspaces to one common workspace.

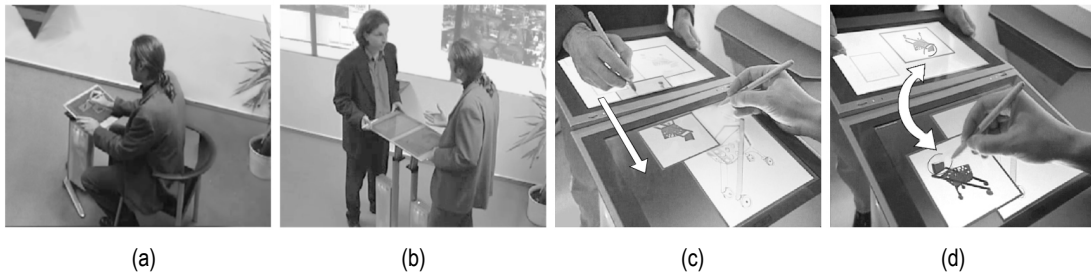


Figure 8-12. To switch between individual work (a) and tightly-coupled work, two ConnectTables are placed next to each other (b). This results in a homogeneous display area enabling the exchange of information objects (c). Moreover, users can have their own but shared views of the same information object for tight collaboration (d).

### Combining Roomware Components

In order to combine the displays of two ConnectTables to a homogeneous surface when they are connected, the model described so far was extended by a new abstraction for the dynamic combination of several roomware components. A **CompositeRoomwareComponent** consists of multiple other roomware components (fig. 8-10), but has its own display area (see fig. 8-13 below). If other roomware components are connected to a composite roomware component, their displays belong to the common display area—which adjusts the local view of all connected roomware components automatically.

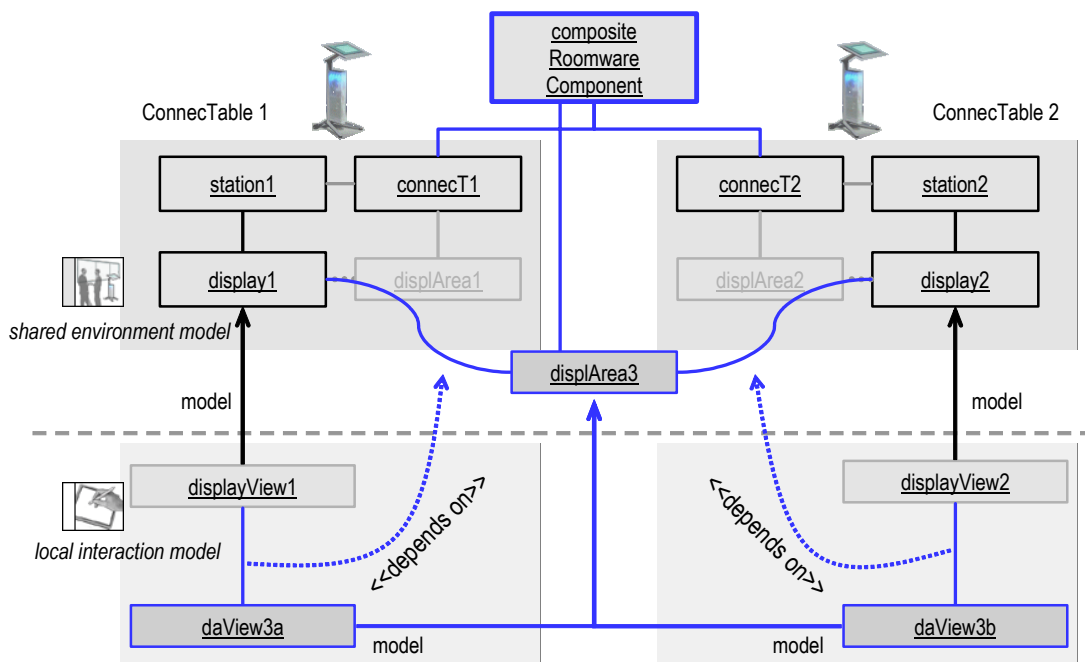


Figure 8-13. Two connected ConnectTables. Both displays are temporarily assigned to the common display area (`displArea3`) of the composite roomware component.

### Connecting the Roomware Components

To connect two ConnectTables, a new composite roomware component with a new display area is created. The two ConnectTables are added as components to it. This also moves their displays from the ConnectTables' original display areas to the new common one (fig. 8-13). As a result, the view objects will be updated to show the correct part of the common workspace: Per definition, the station changing the model will be the lower display using a zero rotation

and an offset of (0, 1). The other ConnecTable is assigned to the upper display. As both ConnecTables have the sensors and tags built in on the same side of the display, this display is rotated by 180 degrees relative to the display area (fig. 8-14). Therefore, its rotation is 180 and its offset is set to (0, 0). The displays' transformations (see section 6.8.3) are used to ensure that the presentation is visualized with the correct orientation.

While the software part of the implementation described so far can handle an arbitrary number of tables, this part is tailored to connections between exactly two ConnecTables. This is no problem, as the current hardware realization is only equipped with sensors that can detect a second table only. If advanced detection hardware becomes available (e.g. Hinckley, 2003, this part of the software has to be extended with functionality that computes appropriate offsets and rotations.

### Connecting the Workspaces

Every display area shows originally its own workspace. When two ConnecTables are connected, the contents of their workspaces must be moved to the new common workspace of the composite roomware component's display area (fig. 8-14). All objects shown in the upper display's workspace (*workspace1*) must be rotated, as the new workspace's orientation is upside-down compared to the original workspace's orientation. The objects in the lower display's workspace (*workspace2*) must be moved down by the size of the upper display. For calculating the positions of the objects in the common workspace, it was possible to reuse the transformations (see section 6.8.3), which were originally developed to transform the output of views.

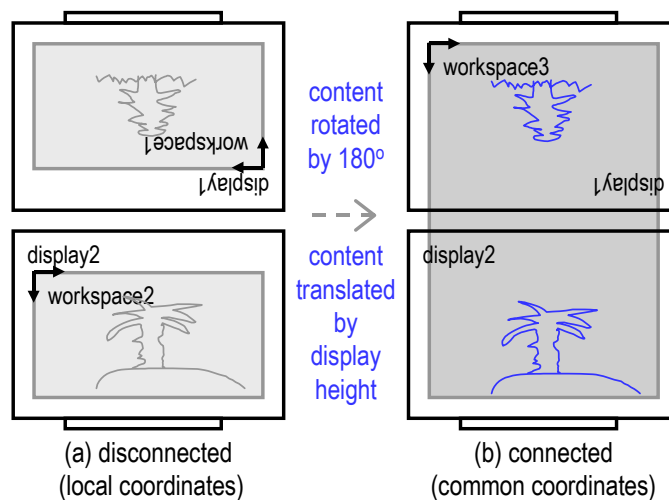


Figure 8-14. Combining the contents of two individual workspaces to a larger common one. To keep the contents at the same physical position they have to be rearranged with respect to the coordinate system of the common workspace.

Finally, the background color of the workspace is darkened to give *feedback to the user* that the ConnecTables are connected. For the design of the ConnecTables' user interface, we decided that the simplest design is also the best. When connecting, as little as possible information displayed at the tables should change, just as physical objects placed on a table would also not change if the table were moved. While this idea proved to be natural for users, we noticed that it is necessary to provide feedback about a successful connect action, especially if it is not obvious whether or not the sensors have detected the other table. Changing the background color was chosen as an ambient, non-disturbing visualization.

### Disconnecting

When the sensors recognize that the ConnecTables are separated again, the reverse operations are performed. The objects in the common workspace are moved back to the private workspaces of the ConnecTables depending on their position inside the workspace. Objects overlapping the border between the displays are moved to the display's workspace where most of them are visible.

Then, the ConnecTables are removed from the composite roomware component and the displays are attached to the previous display areas. Consequently, the views are updated to show these display areas and workspaces.

#### 8.2.4 Analysis of the ConnecTables Implementation

The implementation of the ConnecTable module could draw on abstractions that are provided by the BEACH framework. The *display area* enabled dynamically adding and removing available displays. The *transformations* made it possible for displays with different physical orientations to be combined to form a homogeneous display area that is correctly rendered on all involved displays without having to adapt any view. In addition, the ConnecTable module could directly reuse the *sensor interaction model* and the *sensor management* that had been developed for the Passage system (see previous section). This was possible because the BEACH model suggested identifying reusable parts at the model and generic levels for the implementation of Passage.

For the ConnecTables, it was crucial that the environment, user interface, application, and data models be shared. The *shared environment model* allowed sensors integrated in both ConnecTables to access information about the tags attached to the remote station, and to modify the model by adding connections between stations. The shared user interface, application, and data models are necessary to be able to flexibly and easily exchange user interface elements, editing state, and information (see fig. 8-12).

In general, the implementation effort was about two weeks by a (at that time, 3rd year) university student with two years experience of using the BEACH framework and model. The most time-consuming part was actually dealing with the concurrency issues, as in the case of the ConnecTables the same, complex action is triggered by two sensors simultaneously. In contrast, actions invoked by users are very unlikely to conflict<sup>38</sup> and are thus well handled by the conflict detection algorithms of COAST. As described, the solution was to introduce the connection as an intermediate, task-level abstraction in the environment model. In addition, the implementation of the sensor management had to be improved, as the sensors do not always produce reliable results. Dropouts had to be handled by software to avoid unintended disconnecting and reconnecting while the tables were placed next to each other.

An obvious extension to the implementation described here is support for connecting an arbitrary number of ConnecTables. This requires on the hardware side an appropriate sensor hardware setup capable of detecting ConnecTables placed at the sides and at the bottom of each table. On the software side, the *Connection*, *Tag*, and *TagSensor* need to be extended to reflect the additional geometry information that can be captured by the sensors. They must be aware of their position relative to the station's display. The *ConnectionMonitor* must be extended to construct the equivalent layout of all displays within their display area and to compute the corresponding transformation for the workspaces' contents. All other aspects of the implementation can remain unchanged. This is possible as the shared environment, data, and user interface models allow straightforward access to the other ConnecTables' states, and because the environment model already provides the display area as an abstractions for the combination of physical displays.

---

<sup>38</sup> In our experience, two users simply do not do the same action at exactly the same time.

### 8.3. Integrating Audio Feedback: Non-Speech Augmentation

In contrast to working with physical objects, manipulating digital information objects produces no acoustic feedback about the ongoing activities. Therefore, we experimented with adding acoustic feedback for interaction with roomware components (Müller-Tomfelde and Steiner, 2001; Müller-Tomfelde, 2003).

Acoustic feedback can be helpful in a number of situations. First, working with a gesture-based interface like the one provided by BEACH, acoustic feedback about recognized gestures and triggered commands gives additional cues for the user about the interaction. This can easily be implemented in a similar way as the immediate visual feedback described in section 7.5.2, as it focuses on a single user, involving local components only.

Other situations for acoustic feedback arise when several people are collaborating. Both for multiple users at one interaction device (req. C-3) and for users working at different roomware components (req. UC-1) acoustic feedback can help avoid unexpected interference.

For example, if two users are working loosely coupled at a DynaWall, acoustic feedback can provide cues of the other user's activity. This is especially helpful since large visual interaction areas (such as the one provided by the DynaWall) cannot be overviewed when working close to the surface. Therefore, acoustic feedback can be used to announce the arrival of new information objects if an information object is thrown towards another user.

Furthermore, if a user sitting in a CommChair is modifying material shown at a DynaWall, people working at the wall may not notice remote actions—even if the “remote” user is working in close physical proximity.

These cases are similar from the perspective of the software, as in both cases several computers are involved, due to the fact that the DynaWall is a multi-computer device (req. U-2). As the BEACH framework implements data, application, environment, and user interface models as shared objects, it is easily possible to use the information that is present in any of these models to give feedback on any involved machine.

↓ Section outline

This section presents the support for audio feedback. First, the hardware setup is described. Subsequently, an overview of the software architecture is given, before the audio interaction model and the MIDI output support is explained. Finally, the example of audio feedback for thrown objects is explained to illustrate how the audio feedback works.

#### 8.3.1 Audio Hardware Setup

In order to give acoustic feedback at the DynaWall, a loudspeaker was mounted invisibly behind every segment of the wall. The speakers are connected to a PC equipped with dedicated audio hardware, denoted “audio PC” in figure 8-15. The audio PC was added as a fourth station of the DynaWall, which has no visual display, but special audio hardware and audio output devices attached.

In our first setup, the audio PC was only connected via Ethernet with the BEACH server. However, we experienced that the network delay and the transmission of messages by the server was not fast enough and had an unpredictable delay, resulting in unsynchronized acoustic and visual feedback. Therefore, we added direct MIDI connections between each station and the audio PC. This allows computing visual and acoustic feedback at every machine for its local context, enabling sufficient synchronization of visual and acoustic output. The audio PC has the role of generating, mixing, and filtering the audio signal from the MIDI data.

It is important to understand that the MIDI connection is used to synchronize visual and audio output at the DynaWall; the audio feedback is given independent of the client actually initiating a change in a shared model. Thus, this setup can also give feedback for remote interactions.

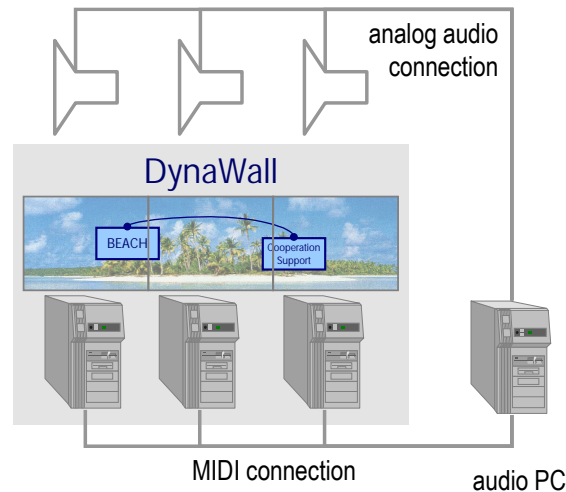


Figure 8-15. Audio hardware setup. The audio PC is connected to the DynaWall via MIDI to ensure a real-time transmission of audio information.

### 8.3.2 Audio Software Architecture Overview

To enable the generation of audio output, three software modules have been implemented. According to the BEACH conceptual model, the functionality was divided into three layers and it covers two concerns, the interaction and environment models. Similar to the visual interaction model defined in chapter 5.2, an audio interaction model (BEACHaudiomodel) is defined (see fig. 8-16). It serves as an abstraction, mediating between the module providing acoustic feedback for interaction with documents (BEACHdocumentAudio) and the interface to the used audio hardware, in this case a MIDI device (BEACHaudio.MIDI). These three modules are presented in the next sections.

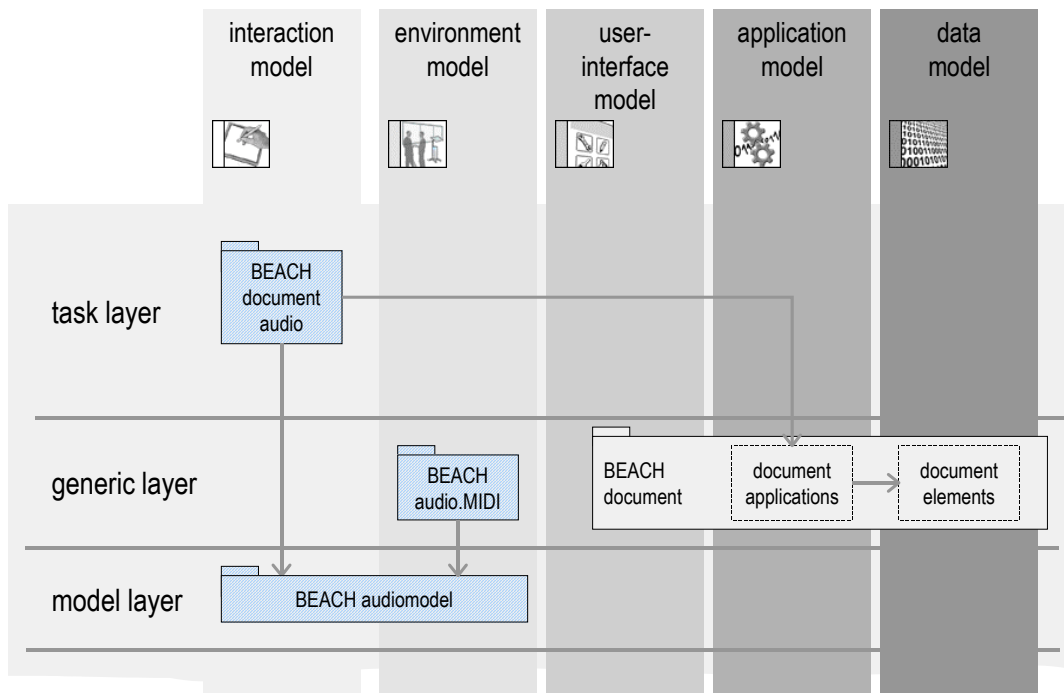


Figure 8-16. The audio software architecture is structured according to the BEACH conceptual model. The audio model defines an abstraction for audio output. MIDI support is implemented as a concrete output device. Module document audio provides acoustic awareness about activity in the document application model.

### 8.3.3 Audio Interaction Model

The audio interaction model (`BEACHaudiomodel`) defines an abstraction for output of acoustic feedback, similar to the visual interaction model defined in chapter 5.2.

Analogous to the “view” classes (see sections 6.2.2 and 6.8), it defines “audio presenters” (class `BeachAudioPresenter` in fig. 8-17) that handle the acoustic rendering of document state and—in contrast to view classes—also of state changes.<sup>39</sup> This is important, as acoustic feedback can be very helpful to inform a user about activity, implying state change. It is very difficult to design an acoustic representation of a document’s state without being annoying over time. The audio presenters use the dependency mechanism (see section 6.1.3) to detect state changes automatically by subclassing `BeachComputedModel`.

Audio presenters also define a hierarchy representing the structure of the document (fig. 8-17). To be able to couple the audio model with an application, the class `AudioApplication` is defined. Similar to class `VisualApplication` (see section 7.4.1) that is concerned with the view hierarchy, class `AudioApplication` manages the audio presenter hierarchy in an analogous way. As class `AudioOutputDevices` is an output device, an audio output device can be dynamically attached to a station. Its device service, which is provided by the device hook (see section 6.6), can be used to open and close the audio application.

To generate output, an audio presenter can allocate an `AudioChannel`, which provides the functionality to describe audio signals in a device independent form. This is equivalent to the

<sup>39</sup> While not implemented in the BEACH framework, it is possible to visualize state changes as well. Gutwin and Greenberg (1998) have shown that animated state changes help to provide workspace awareness.



GraphicsContext used to render visual output in the view interaction model. Supported atomic audio output elements are “loops” and “sequences”. Loops are pieces of audio data being repeated until stopped explicitly. They are used for continuing feedback, e.g. while an element is moved. Audio sequences are played once; they notify about changes only.

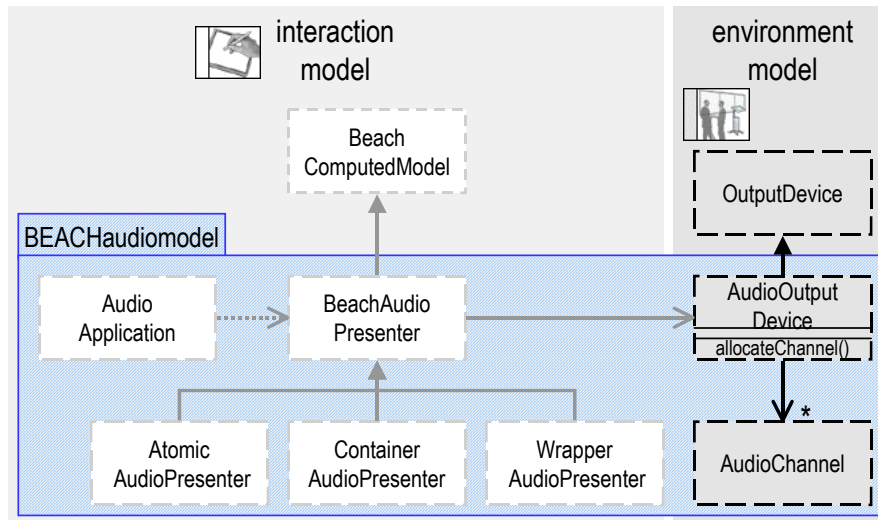


Figure 8-17. Interaction model for audio output. The interaction application is extended to handle a hierarchy of audio presenter objects.

#### 8.3.4 MIDI output

As the DynaWall’s stations are connected to the audio PC via MIDI, a concrete audio-output device is provided by module `BEACHaudio.MIDI` that generates MIDI events and sends them to the MIDI interface (fig. 8-18). As MIDI supports only a very limited number of channels, class `MIDIChannel` defines virtual MIDI channels, which are mapped to currently available channels by the `MIDIOutputDevice`.

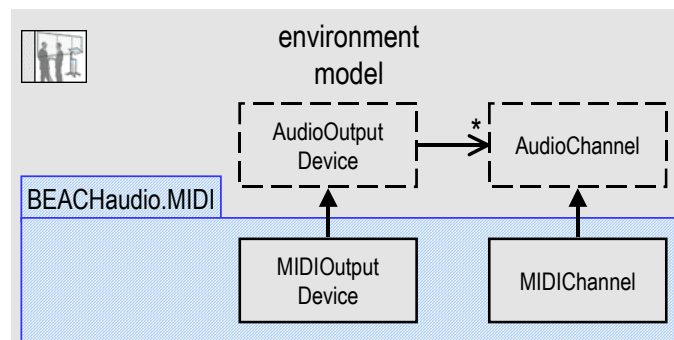


Figure 8-18. MIDI output is a concrete implementation of the abstract audio-output device and channel.

#### 8.3.5 Audio Feedback for Throwing

Document content is hard to represent acoustically if no speech should be used, which would interfere with collaboration among people. Thus, the `BEACHdocumentAudio` module concentrates on geometric properties of document elements. This decision implies that this implementation of acoustic feedback cannot be used as a replacement for views; it only provides additional feedback in a multi-modal way.

## 8. Extending BEACH for New Forms of Interaction

In addition, we decided to use acoustic feedback to notify about changes and ongoing activities only. The example that is discussed here is rearrangement of document elements by moving or throwing<sup>40</sup>.

To enable acoustic feedback, class `PositionWrapperAudioPresenter` (fig. 8-19) is defined to handle acoustic rendering of the position of document elements. It monitors changes of the position caused by movement commands, and checks whether a document element is currently being thrown. In this case, not only the position and size but also the current speed is used to generate the output signal. Details of the generation of audio output are not in the focus of this thesis and can be found in (Steiner, 1999; Müller-Tomfelde, 2003).

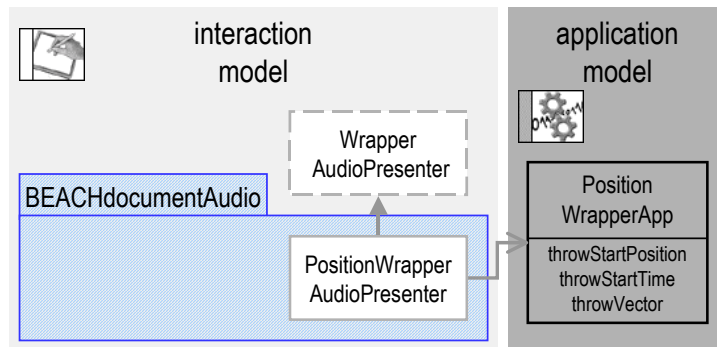


Figure 8-19. The audio presenter for position wrappers monitors movements happening in the position-wrapper application model.

Example 8-1: Adding acoustic interaction

Figure 8-20 illustrates how the audio interaction model is added. The audio interaction application is attached to the same user interface display application that is also used for the visual output, as it augments the visual presentation. The audio interaction model then creates a hierarchy that resembles the view hierarchy. While in this example all container audio presenters serve only the purpose of holding the sub-presenters, the audio position wrappers give acoustic awareness about the position of elements.

This is also an example that shows how two different interaction models are used for the same user interface model, stressing the importance of the separations of interaction and user interface concerns.

<sup>40</sup> The implementation of the throwing interaction style is explained in section 7.2.2, page 134.

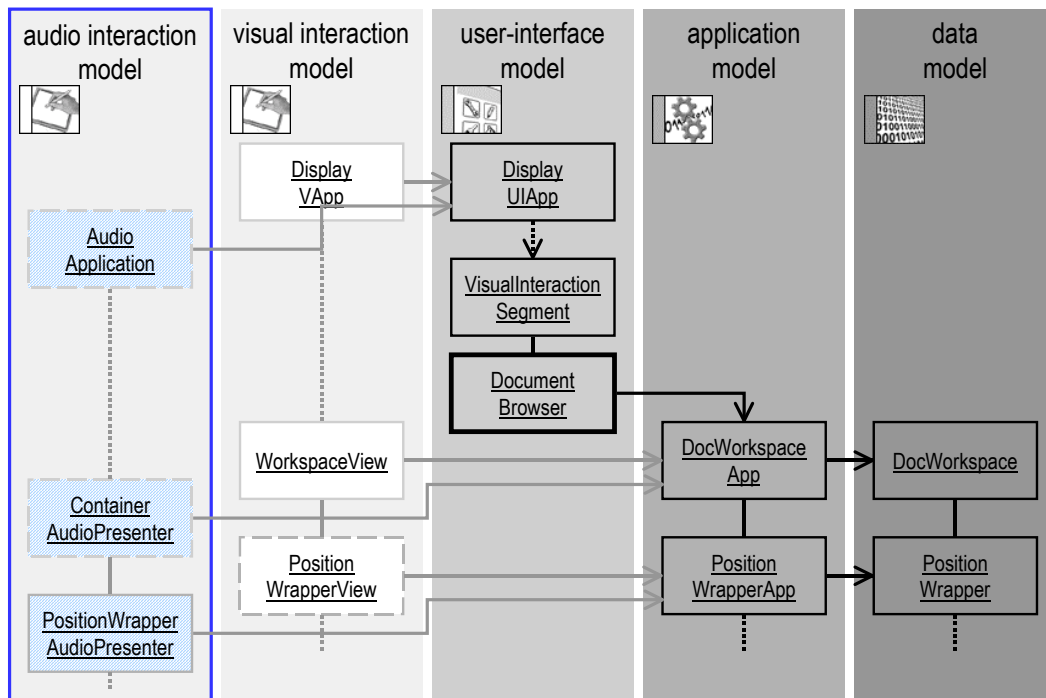


Figure 8-20. An interaction model for acoustic output can be added without interfering with the existing interaction modalities. The audio interaction application uses the same user interface application to be able to augment the visual presentation.

### 8.3.6 Analysis of the Audio Feedback Implementation

While the sensor interaction model developed for the Passage system shows how another input modality can be added, the audio interaction model is an example of another output modality that augments the visual presentation. This serves as an example of two interaction models *sharing the same user interface*, application, and data models.

The *device hook* provided at the model level (see section 6.6), was used to be able to add an audio output device to the station and use its device service to start the audio interaction application.

Additionally, the audio interaction model generates—similar to the view interaction model—a presentation for a rather complex application model with a hierarchical structure, even if the current acoustic presentation is limited. This is possible by using the abstractions of *composites and wrappers* that are also employed in the view interaction model.

An issue for future investigation is the possibility to audio transformations in an analogous way as for views (see section 6.8.3). *Audio transformations* could be inserted into the audio presenter hierarchy to modify the acoustic presentation of sub-hierarchies, e.g. to adapt the volume, frequencies, or the speed.

The integration of the feedback for throwing was done by a 3rd year university student with no prior experience of using the BEACH framework as part of his three month internship. However, only a small part of the work was dealing with BEACH related issues. The major part of the work was creating the hardware setup, dealing with device drives of audio hardware, and designing pleasant and non-annoying acoustic representations.

## 8.4. Discussion: Experiences Extending the BEACH Framework

This chapter presented examples showing how to implement new interaction forms on top of the BEACH framework. The **sensor model** provides an interface to include properties of physi-

cal objects that can be recognized by sensors as input to applications, and actuator to manipulate physical objects. The sensor model is used by the *Passage* system and by the *ConnecTables*. The **audio model** enables giving acoustic feedback. Implementing these interaction models, we experienced that new interaction forms can be realized in a **straightforward and convenient** way. The chosen samples prove that the BEACH model and framework supports **different interaction forms** (req. H-1).

The audio interaction model revealed a benefit of using a software architecture that is structured according to the BEACH model. Automatic generation of user interfaces and appropriate interaction models is not the focus of the BEACH framework. Consequently, a new interaction model has to be defined for all components that have to support this new interaction form. Fortunately, as the framework clearly separates concerns and abstraction levels, each component profits from all **interaction forms defined at lower levels of abstraction**. If, for example, a new interaction model for speech input is defined, then, at the generic level, common actions, such as “move”, or “delete” can be invoked by speech commands. Now, even if no specific support is added at the task level, all components that use the generic commands can be controlled by speech as well. However, if cross-modal generation of user interface is necessary, approaches as presented in section 3.3 can be easily integrated.

The audio model shows also how to **combine multiple modalities**, here acoustic and visual feedback, in an independent manner. Synchronization of modalities is performed via the underlying shared models. Therefore, it is necessary that the **interaction be separated from the user interface model**, as otherwise it is not possible to add interaction modalities for the same user interface. However, to provide more sophisticated forms of multi-modal interaction, especially for multi-modal user input, additional components might be necessary as part of the low-level interaction model that are managing multiple and flexible change of interaction modalities (Oviatt *et al.*, 2000; Myers *et al.*, 2002).

The *ConnecTables* are an example of how to handle **dynamic changes in the environment model**. Whenever sensors detect that two *ConnecTables* are placed next to each other, they update the environment model accordingly.

This can be used to provide **environmental awareness**. As the interaction model implemented as part of the *BEACHconnection* module observes the state of the environment model, it is possible to trigger functionality upon state changes.

### 8.4.1 Comparison of the Interaction Models

While the three sample interaction models (including the view model described in section 6.8) define different abstractions that are unique to the supported interaction style, they follow a similar structure.

- All interaction models **operate on shared models**; they can access user interface, application, data, and environment model. Using shared model allows **flexible distribution** of user input and presentation of information.
- The interaction models use the **device hook** provided at model level (see section 6.6) to add the necessary interaction elements. The service associated with each device can be used to establish communication between the device and the components handling the corresponding actions. This is the essential feature for **ensuring extensibility** (req. S-2) for future interaction forms.
- The concepts of interaction and user interface **applications** are used as the root for these two concerns. The applications are created by the corresponding device service.
- The interaction models normally have two parts: one dealing with **presentation**, the other with **user input**. The visual interaction model defines views and controllers, the sensor model actuator and sensors. However, this separation might not be necessary or feasible in all cases. The implementation of the audio model currently supports acoustic presentations only.

- Presentation of shared objects is modeled as being **dependent on shared models**. This has the benefit of being independent of the client causing a state change.
- To be independent of the underlying hardware the output part introduces an **abstract interface to generate the presentation**. The views use a graphics context, the audio model the audio channel.
- **User input triggers modification** of the shared model state. The way user input that is received by the interaction model depends on the interaction style and the available device drivers.

Common possibilities for mapping input to functionality are **sending events** (as in the view model) or **observing device state** (as in the sensor model). It seems that events are appropriate if the device generates *explicit* user input, such as mouse or keyboard, and the input is directly connected to a complex user interface or application model. Complex, here, means that it is modeled as multiple interrelated objects that are dynamically changing candidates for the target action. When events are used to invoke functionality, different interaction forms require **different event dispatching strategies**.

When functionality is triggered upon changes of shared state, this gives freedom to **distribute input devices** and input handlers. Here, state changes *implicitly* define input events. The way to deal with this kind of input can be regarded as rule-based. For each action, a rule describes the state in which it has to be triggered. In contrast to the “classical” input events that are normally dispatched to a single controller only, it is quite possible that several rules exist that specify the same state in which they have to be triggered. For these rules, **constraint solving** is more appropriate than event dispatching, as the main problem is to figure out when the specified event (i.e. the state change) has occurred, not determining the appropriate action. To ease this detection, the ConnecTables module introduced the “connection” as a tailored abstraction. It abstracts from the interpretation of the available context information and allows the specification of the rule at the appropriate level of abstraction (task level in this case).

#### 8.4.2 Properties of the BEACH Architecture

Using the BEACH framework revealed that its architecture has properties that make reuse and extensibility more convenient.

As the functionality belonging to the different levels of abstractions of the BEACH model is implemented by the BEACH framework as separate layers, **lower level components can be reused** easily. For instance, the sensor model implemented for the Passage system defines functionality at the model level. This was reused unmodified in the implementation of the ConnecTables. This example shows that the layers allow making a **horizontal cut** through the architecture (fig. 8-21a). All components placed at a higher level can thus be modified, or new components can be added without the need to adapt lower-level components.

Analogous to the horizontal cut, which is enabled by the software layers, it is also possible to make a **vertical cut** in the architecture (fig. 8-21b). This means that software components implementing separate concerns can also be replaced without the need to change the components they depend on. This chapter presented examples showing that **different interaction models** can be implemented without the need to modify other models. If a **new user interface** is created, this might require changes in the interaction model only. If the software interface of the application model has to be changed, this might affect both user interface and interaction model.

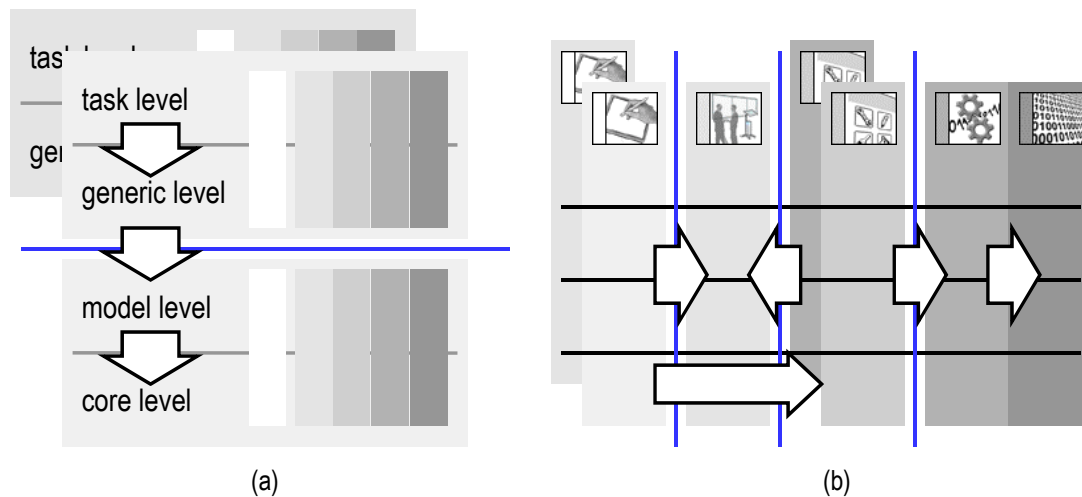


Figure 8-21. The BEACH architecture enables horizontal (a) and vertical (b) cuts through the architecture. This is essential to ensure reusability and extensibility.

#### 8.4.3 Developing Distributed Systems with the BEACH Framework

The second dimension of the BEACH conceptual model (figure 4-2) is concerned with coupling and sharing. This chapter presented several examples of how the **shared-object space** that forms the basis of the BEACH framework reduces significantly the development effort, while providing means of flexible distribution. The examples also show the benefits of introducing the dimensions of separation of concerns in combination with sharing in the underlying BEACH conceptual model. This enables developers to decide for each concern about the appropriate degree of coupling for the task at hand.

All three examples demonstrate that **sharing the data model** is essential for collaborative ubiquitous computing applications, by enabling an intuitive way of accessing and manipulating shared information. The **shared application model** enriches the collaboration possibilities, as it allows using the same editing state, which could be transported by Passage, shared by the ConnecTables, or used for remote acoustic feedback of remote interactions. This allows flexible changes of devices, locations, and participants while being able to continue work seamlessly in spite of context changes. The ConnecTables also rely on a **shared user interface model** to be able to exchange user interface elements, such as toolbars or overlays, while being connected. The **shared environment model** is used in the implementation of the ConnecTable to find other tables to connect to. The example of the *sensor model* demonstrates that using the shared environment model also eases the **distribution of sensors and sensor observers**.

In spite of the **interaction model** not being shared, the examples presented in this chapter illustrate how the **dependency mechanism** is used to generate a **coupled distributed presentation**. As the presentation is local, but coupled to a shared model via constraints, the presentation can be adapted to the local context. This is used by the ConnecTables to transform the common coordinate system to the local one that is rotated or translated accordingly to show the correct part of the common display area with the correct orientation. The *audio interaction model* illustrates how a remote device can use an arbitrary interaction model to allow flexible provision of **awareness of remote actions** and state changes.

↓ Next chapter

While this chapter discussed the possibilities of integrating new forms of interaction with the BEACH conceptual model and framework, the next chapter shows how applications can be created based on the framework.

## 9. Tools for Collaborative Roomware Environments

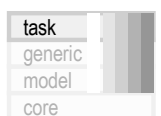
---

This chapter presents examples of how applications are implemented using the BEACH conceptual model and framework. Two examples are discussed in detail. The *MagNets* tool is used for collaboratively generating and organizing ideas. It shows how new document element types can be implemented that define a new interaction behavior. *PalmBeach* extends the support for creativity sessions to portable devices. Ideas can be noted on a Palm and beamed to a public display. PalmBeach shows how it is possible to include a different hardware platform, which requires new user interface concepts and a different implementation for a conceptually shared model.

---

With the help of the BEACH model and framework, 12 tools and extensions have been developed up to now (including the three extensions described in the previous chapter). These tools and extensions are implemented by about 25,000 lines of Smalltalk source code<sup>41</sup> in total, which gives an average of about 2000 lines per application. To date, more than 15 software developers have used BEACH<sup>42</sup>, both students from the Darmstadt University of Technology and researchers at IPSI, with varying experiences in software development ranging from novices to experts. Most of them had no prior knowledge of the Smalltalk programming language; none of them had prior knowledge of BEACH. Several developers continued using the BEACH model and framework for several years.

We experienced that programmers with little training—but with the help of BEACH—were able to implement software that supports synchronous cooperation and different forms of interaction, up to a complexity they would be able to handle as equivalent single-user desktop software. In addition, this software required no special care about synchronous actions or special



---

<sup>41</sup> The total number of lines in the Smalltalk file-out including resources.

<sup>42</sup> BEACH here (and in the following) refers to conceptual model, software architecture, and application framework.

interactions. After one year of experiences with BEACH, students had internalized the concepts of the BEACH model and were able to create software designs that reflect the specific properties of roomware environments.

The following tools have been developed for BEACH. *BeachMaps* is a tool of the BEACH “Creativity Suite” (Prante *et al.*, 2002) that provides cooperative mind-maps (Buzan, 2002). It can be used to structure collaboratively the ideas gathered during a brainstorming session. *BEACHshow* is a simple tool to give presentations in a roomware environment, enabling the audience to directly participate by remotely annotating slides. The *RoomViewer* gives environmental awareness by visualizing nearby roomware components. The *Remote Access Tool* allows access to remote roomware components. It is an example that shows that it is useful to have a local interaction model that can be adapted to its context. *BEACHsearch* provides a query-by-example user interface to retrieve documents. It can also use context information that is attached to documents as search criteria, such as the roomware component at which it was created or users that have been present at this meeting. This information is taken from the environment model. *BEACHcontext* integrates the BEACH framework with the iROS Event Heap (see section 3.3.2), in order to establish a simple interface to sensor and context data. *BEACHweb* provides a BEACH client that works as a web server. Users that have no access to a BEACH client, but a web browser can view BEACH documents. The web browser represents the current state of the workspace rendered as Scalable Vector Graphics (SVG) (World Wide Web Consortium (W3C), 2003). Update is not automatic due to limitations of the HTTP protocol, but users can update the presentation manually by reloading the web page. This is an example of how to generate an interaction model for a web browser.

↓ Chapter outline

This chapter, now, presents two examples in detail showing how applications can be implemented using the BEACH conceptual model and software framework. They are taken from the BEACH “Creativity Suite”. The first tool, *MagNets*, shows how new document element types can be implemented that define a new interaction behavior. The second, *PalmBeach*, is an example of how it is possible to include a different hardware platform, which requires new user interface concepts and a different implementation for a conceptually shared model. The examples have been selected to illustrate different aspects of the BEACH model and framework. The examples presented in this chapter were implemented by students and colleagues with the help of the BEACH model and framework. The author contributed to the software design only.

### 9.1. MagNets: Extending Document Model and Interaction Behavior

“MagNets” is the first application that was developed with the BEACH model and framework. It supports collaborative generation and structuring of ideas in creativity sessions. MagNets (“Magnetic cards to form idea-Networks”) provide enhanced interaction mechanisms in analogy to the magnetism-metaphor in order to support successive bottom-up structuring of cards.

↓ Section outline

In this section, first, the functionality of MagNets is explained, followed by the design of the software application, highlighting important aspects of the BEACH framework.

#### 9.1.1 MagNets Functionality

When several people generate ideas during brainstorming sessions, efficiency is increased if each participant can write down the ideas immediately to prevent ideas being lost. Besides, new ideas should immediately be visible to other participant to trigger further associations and ideas. In a roomware environment, this can be easily realized. Every participant can write down ideas using a personal device (such as a CommChair). All ideas are placed in a shared workspace, which is also visible at a public device, such as the DynaWall.

To write down ideas, MagNets offers small (digital) cards, which can be collected in a shared workspace. It distinguishes two types of “magnetic” cards: *element cards* and *title cards*. Element cards are repelling upon creation to avoid overlap. When an element card is dragged on top of another, the first one is pushed away to a defined distance. This is to fulfill the basic require-



ment of brainstorming that during the stage of idea creation all ideas have to be treated equally.

In contrast to element cards, title cards are attracting upon creation, thus supporting progressive clustering of cards during idea-structuring phases. They serve to name clusters of ideas that are related to each other. When an element card is dragged over a title card, it sticks to it and becomes attracting itself. It is freed again by quickly tearing it away from the magnetic cluster.

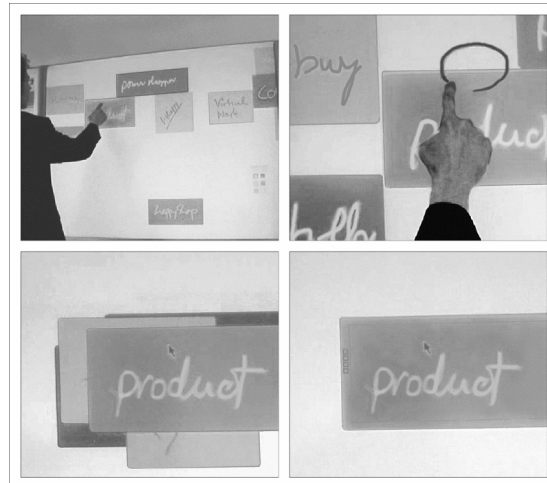


Figure 9-1. The animated collapsing of a magnetic card cluster.

MagNets also provide flexible visualization forms for card clusters; they can be collapsed and expanded (see fig. 9-1). This is useful to save space on the whiteboard and to redirect the focus of attention to other ideas.

An extensive description of the functionality of MagNets is published in (Prante *et al.*, 2002).

### 9.1.2 MagNets Software Design

The example of the MagNets modules illustrates how it is possible to add new document elements that have a new interaction behavior. To implement the described functionality, the MagNets module defines the “card” as a new document element (class `Card` in figure 9-2). Clusters of magnetic cards are represented by instances of class `MagneticGroup`. The `MagNetsModule` class can be used as an example of how to add new toolbars to the generic root toolbar. In addition, a new application model for position wrappers is needed to realize the new interaction mechanism.

#### Cards as New Document Elements

A “card” is a container object containing only scribbles, text, and images. To implement a new document element, the data, application, and interaction model is defined for cards. If a card is a title card, attribute `isTitle` is set to `true`. Title cards can be used to create magnetic clusters of cards. In this case, an instance of class `MagneticGroup` is created and all members of the cluster are added as members of this group.

The application models of cards and magnetic groups take care of redirecting all commands that influence the whole cluster. For example, move and throw commands always involve the whole group. In addition, collapse and expand commands are handled by the application model to switch between the two different visualization modes. Handling collapsing of clusters in the application model has the advantage of allowing loosely coupled interaction with the same cluster of cards. Users working with a different application model can independently col-

lapse one group, e.g., if one user wants to continue elaborating the elements while the other user has already switched the focus to other topics.

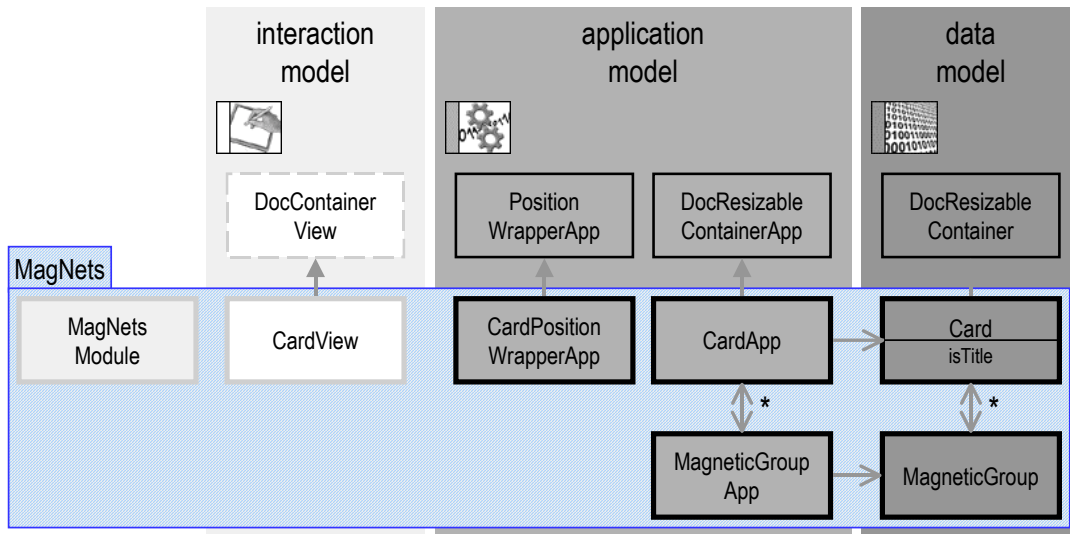


Figure 9-2. Classes defined by the MagNets module

#### Implementing New Behavior of Cards: Magnetic Attracting and Repelling

Attracting and repelling between cards does not only concern the membership of cards in a magnetic cluster. Moreover, as it is also reflected in the position of cards, a new application model has been implemented for position wrappers that is used instead of the default position wrapper application model by the card's application model.

When move or throw commands are sent to an instance of `CardPositionWrapperApp`, it checks for intersecting and close-by cards at the final position. If other cards are detected that are attracting (i.e. that are title cards or that already belong to a magnetic cluster), the card is added to the magnetic group. If a repelling card is found, a new position is computed for the card being moved that ensures a minimum distance to all repelling cards.

#### Making New Document Elements available to Users

To enable users to actually create cards as part of BEACH documents, the “create card” command must be made available in the generic user interface of BEACH. To allow new modules to add new document elements, the generic root toolbar (see section 7.3.4) uses a hook to include commands defined by other modules. (This hook is explained in section 6.1.2.) Therefore, the `MagNetsModule` metaclass overwrites the `toolbars` method and returns a sub-toolbar-command that opens the “MagNets” toolbar. The “MagNets” toolbar, in turn, provides a set of “create card” commands for a collection of both title and element cards in different colors.

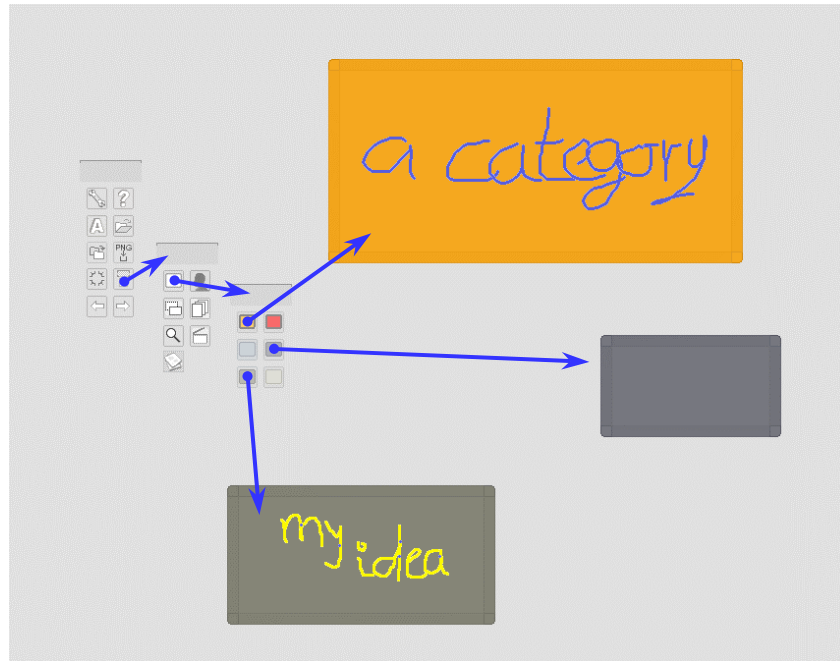


Figure 9-3. The MagNets toolbar is plugged into the modules toolbar. It can be used to create MagNets cards during creativity sessions. *Element cards* are used to write down ideas. *Title cards* are magnetic attracting and can be used to cluster cards into categories.

### 9.1.3 Analysis of the MagNets Implementation

The implementation of the MagNets illustrates three aspects. While to add new interaction forms the device hook can be used (see chapter 8), the BEACH framework provides the *toolbar hook* to include new functionality in the user interface (see section 7.3.4). The new document elements are implemented as subclasses of *abstract document classes*. In this respect, the BEACH framework can be classified as a typical white-box framework. To implement the card's magnetic behavior, a specialization of the *application model* of the position wrapper was used.

As the position wrapper is selected depending on the container in which a card is placed, this design enables implementing different behaviors depending on the container. This way, it was possible to reuse the cards when placed in different structures. In (Prante *et al.*, 2002) a third tool of the BEACH creativity suite, called BeachMaps, is described that covers the idea-structuring phase after all ideas have been collected. It uses a graphical hierarchical structure called mind-map (Buzan, 2002). Ideas written on MagNets cards can be placed into a mindmap. BeachMaps supports automatic layout according to the hierarchy. Here, the magnetic behavior is not needed; instead normal position wrappers can be used.

To provide awareness about the re-positioning of the cards, the movement of the card to its final position is animated. When implementing the animation for the “throwing” interaction technique (see section 7.2.2), we experienced that in a distributed environment the animation works best if there is only one transaction used to set the parameters of the animation. In addition, time-based invalidation of the presentation and a synchronized timer is used to guarantee consistent presentations that are generated locally.

In principle, the animation of the magnetic behavior uses the same approach. However, we faced additional problems due to concurrency. In contrast to ordinary document elements, the final position of magnetic cards depends on other cards, as cards are not allowed to overlap. When other cards are placed concurrently by different clients, their final position can interfere, as the local client cannot consider this when calculating the position. This often results in rollbacks, as both clients access all slots specifying the positions of the cards within this

workspace. Whenever any of these positions is changed by another client in the meantime, the transaction is canceled by the server. As the animation of the movement was not the focus of our research, we did not develop a sophisticated solution. When giving demonstrations of MagNets we simply try to avoid this situation. This example illustrates that concurrency cannot always be handled automatically. However, the shared-object space ensures that each action results in a consistent state, even if this is not the one preferred by all users.

### 9.2. PalmBeach: Implementing Models on a Different Platform

While MagNets supports creativity sessions, “PalmBeach” is focused on collecting ideas in between scheduled sessions using small, mobile devices. Currently, PalmBeach can be used with handheld devices running PalmOS, e.g. the Palm (Palm, Inc., 2003).<sup>43</sup> Ideas that are generated in between meetings can be further processed in a subsequent meeting.

As the Palm is very limited in terms of memory, processing capability and display size compared to roomware components, it was neither possible nor advisable to directly port MagNets and the BEACH framework to this platform. Instead, we chose to develop an independent implementation for the Palm based on the *same conceptual document model*, but tailored to the needs and capabilities of this device. In addition, an interface to exchange data between PalmBeach and MagNets was implemented.

Therefore, PalmBeach can be used as an example to show how the same conceptual model can be implemented differently on different platforms.

↓ Section outline

This section first describes the functionality of PalmBeach and the user interface designed for the Palm. Then, the software design of PalmBeach is presented, discussing the differences from the design of MagNets. Finally, it is explained how data is exchanged between the Palm and the DynaWall.

#### 9.2.1 PalmBeach Functionality and User Interface

##### Tailored User Interface for PDAs

The user interface of MagNets is designed for large interactive surfaces; thus, it does not fit for the small display of a PDA. For PalmBeach, therefore, a new user interface had to be developed (Prante *et al.*, 2002; Magerkurth and Prante, 2001a). It also uses MagNets’ card-metaphor for idea collection. The user can create content on the cards as well as model spatial and link relations between the cards. In addition to the MagNets implementation, each card has a title and an icon to allow for a semantic structuring.

To cope with the limited screen space of a PDA and to the demands of a card-based approach, the user interface separates the content of a card from its relations to the other cards, providing two different views. The *Detail View* (right side in figure 9-4) is used for modifying the content of a single card. The *Relation View* (left side in figure 9-4) shows all of the defined cards to visualize their relations.

---

<sup>43</sup> However, as PalmBeach is implemented on top of an abstraction layer encapsulating the handheld’s operating system called pNF (Magerkurth and Prante, 2001b) there is already a basis to port it to different platforms.

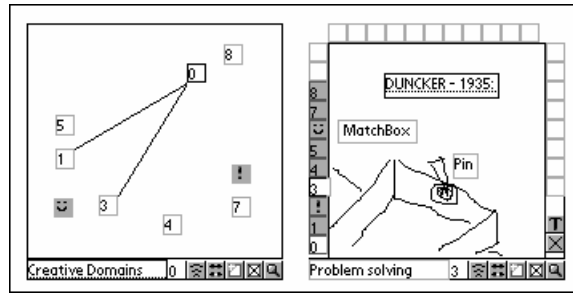


Figure 9-4. Screenshots of PalmBeach's Relation View (left) and Detail View (right).

In the Detail View, the user can draw scribbles, handwritten text, or use text-boxes to create annotations. The Detail View also hosts a framing set of tab controls, the *Navigation Stack*. Each of the tab controls represents a previously created card. Its appearance indicates how the corresponding card relates to the selected card, e.g., if there is a link between them. A single tap on any of the tab controls switches to its corresponding card.

The *Navigation Stack* was introduced to reflect the early stages of idea generation. This is analogous to brainstorming, where ideas are simply collected and not immediately brought in relation to each other to prevent early tunneling in the search space. When the initial flow of ideas ceases, the preliminary organization of the *Navigation Stack* may then be augmented using the Relation View.

In the Relation View, the iconic representations of the cards can be dragged around the screen to model their spatial relationships as well as using hyperlinks complementing the spatial structure.

#### Exchange of Ideas

As PalmBeach is focused on collecting ideas in between scheduled sessions, it is essential that the created ideas can be shared among participants in the next meeting. Therefore, a simply way to exchange ideas between devices is needed. PalmBeach uses the infrared interface built into Palm PDAs to transmit information to other devices. To be able to transfer information generated on a PDA to MagNets running on the DynaWall, an infrared receiver has been integrated in the DynaWall. This way, PalmBeach users can beam their ideas to the DynaWall (see figure 9-5), where they are converted to MagNets element cards.



Figure 9-5. A PDA running PalmBeach pointing at an electronic whiteboard to transfer a card

#### 9.2.2 Differences between the Software Design of MagNets and PalmBeach

PalmBeach has been implemented by Carsten Magerkurth (Magerkurth and Prante, 2001a). For the implementation, it was not possible to reuse any code that had been written for MagNets. First, a completely different user interface was designed that fits the needs of a small PDA. Consequently, a new *interaction and user interface model* had to be designed and implemented. As it is very unlikely that a different interaction style would be used for the user in-

interface provided by the Palm, the interaction and user interface model have been combined (fig. 9-6). This also helps reduce the size of the code.

Second, the *data and application model* could not be reused. MagNets is implemented on top of COAST. However, currently COAST has not been ported to the Palm OS platform. Porting a synchronous groupware framework to a PDA platform is known to be a challenging task due to limited memory and processing capabilities (e.g. no threads) on the one hand, but a different (i.e. mobile) usage on the other (Roth, 2002). To keep data and application model small, a single class was used to represent both the data and the application model facet (fig. 9-6). In addition, no wrappers were used to model the relationship between a workspace and its contents. Still, it is an implementation of the same conceptual model.

Last, only a very simple *environment model* is needed. Class *Beamer* (not shown in fig. 9-6) needs to be aware of the presence of other devices it could transmit data to. This is detected by waiting for responses for messages sent via the infrared port of the Palm.

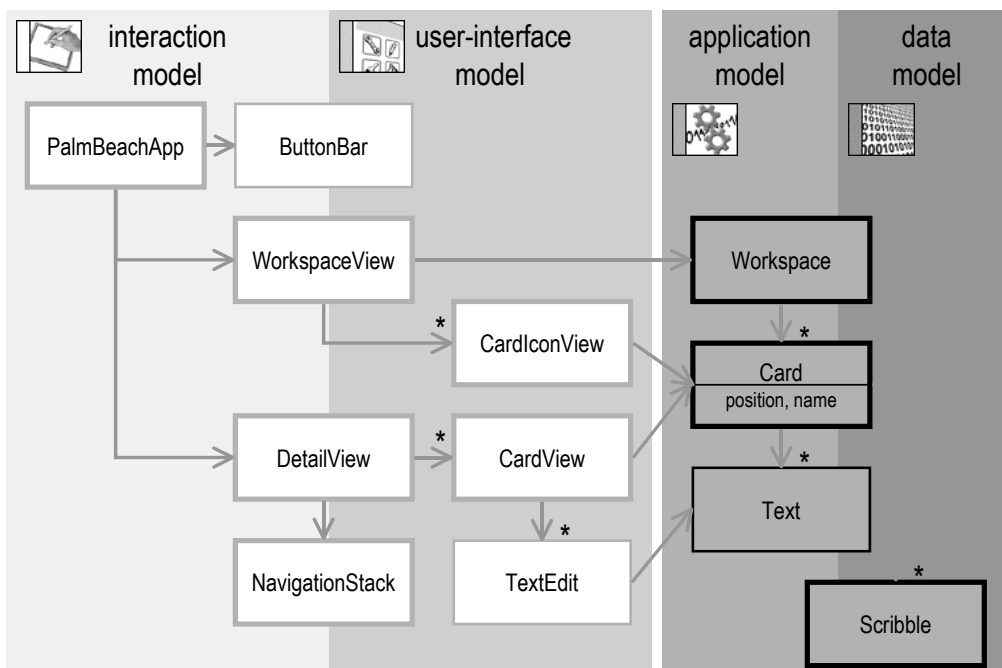


Figure 9-6. Software design of PalmBeach. Due to the constraints of the PDA, data and application models, and user-interface and interaction models are combined.

The software design of the *combined application and data model* is very simple (fig. 9-6). A workspace can contain multiple cards. Each card has a position within the workspace, a name, and a collection of text and scribble objects.

The *combined interaction and user interface model* has class *PalmBeachApp* as root. *PalmBeachApp* combines aspects of the user interface and interaction application, as defined in sections 6.7 and 6.8. As explained above, *PalmBeach* switches between a workspace and detail view (fig. 9-6). The button bar is always visible. The workspace view uses a different view class for cards, only showing an icon for the card (see left screenshot in figure 9-4). For performance reasons, no view objects are created for scribbles. They are rather directly drawn by the class *CardView*.

### 9.2.3 Data Exchange between PalmBeach and MagNets

At the time of implementing *PalmBeach*, no wireless network connection was available for Palm. Therefore, the built-in infrared port was used to transmit cards to other Palm and roomware components. To be able to transmit cards, an infrared receiver was attached to the center station of the *DynaWall* (fig. 9-7).

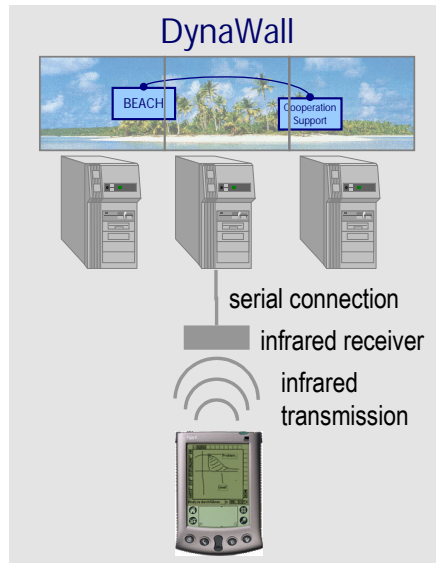


Figure 9-7. Hardware setup for data exchange between Palm and DynaWall

At the station, to which the infrared receiver is connected, the `PalmBeachService` is started using the capability of modules to define services (see section 6.6). When started, it opens the connection to the receiver and waits for card or a workspace with cards to be transmitted. The transmission uses a compact binary marshalling format.

When data is received, it is unmarshalled and corresponding MagNets card objects are created (or updated if a corresponding object already exists) to have the appropriate representation of the cards for the different platform. To make new card objects accessible to the user, the “bridge” implemented for the passage system (see section 8.1) is used. The `PalmBeachService` (defined in module `PalmBeachConnect`, see fig. 9-8) registers itself as a new kind of sensor, the `PalmSensor`. When cards are received, the `PalmBeachService` first creates a passenger object associated with the received data; then the `PalmSensor` is informed about a newly sensed object (the Palm).

To transmit data back to a Palm would be possible using the same interaction (by assigning data to the bridge opened for a detected Palm). However, the currently installed hardware enables transmission from the Palm to the DynaWall only.

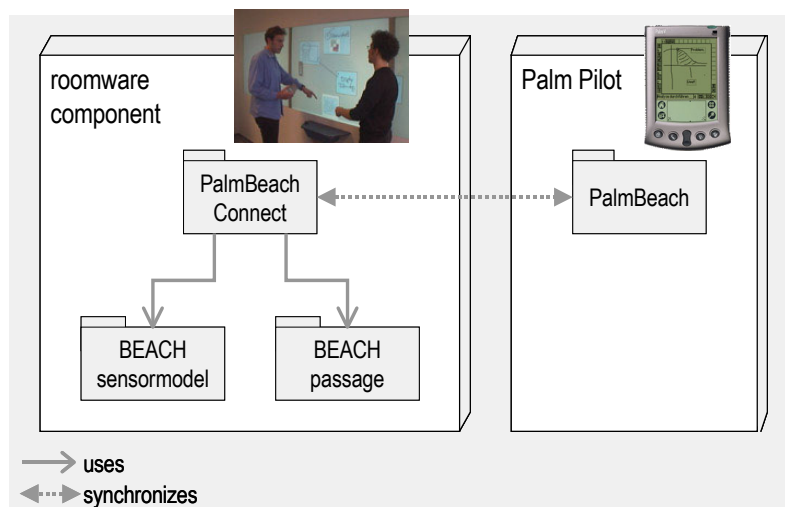


Figure 9-8. Software architecture to handle the data transmission between PalmBeach and roomware components

### 9.2.4 Analysis of the PalmBeach Implementation

Although not much code could be reused, PalmBeach proves that the BEACH model can be applied for the construction of software that runs on other hardware platforms with different requirements. PalmBeach shows that it is possible to combine the implementation of several concerns due to size restrictions, without violating the guidelines given in the BEACH model.

As the Palm we had available when implementing PalmBeach had no permanent network connection, PalmBeach does not support synchronous collaboration. This is an example of a lower degree of sharing, illustrating that there possible steps on the second dimension of the BEACH model (coupling and sharing) in between local and fully shared models.

### 9.3. Discussion: Experiences Building Roomware Applications

This chapter presented two sample tools that have been developed on top of the BEACH framework to explain different usage and extension possibilities.

The *MagNets* tool is an example of how new functionality can be easily integrated into the framework. First, it provides a **new type of document element**, the magnetic cards. They offer a **new form of interaction**, the magnetic attracting and repelling. As the relationship between a document container and its elements is modeled as an explicit **relation wrapper** object, an adapted algorithm for positioning elements can be realized. In this way, the new behavior can be reused for other document elements if desired. The new functionality is linked into the user interface using the **toolbar hook** that is offered by the BEACH framework.

*PalmBeach* shows how the **same conceptual model can be implemented differently** on different platforms and for different devices. PalmBeach and *MagNets* *conceptually* share the same data model. As PalmBeach runs on a PDA, which has restrictions in memory and processing capabilities, the implementation cannot be reused. When data is transferred between the PDA and a roomware component, it is converted between the implementations. In addition, a standard Palm has no permanent network connection. To transfer data, the user has to explicitly use beaming instead of having a synchronously coupled shared-object space. This is an example for a **different implementation of conceptually shared objects**.

PalmBeach has a **different user interface for the conceptually same data model** than *MagNets*. To conform to the platform restrictions, the PalmBeach implementation **combines several concerns into one class**. Interaction and user interface models are implemented as two facets of one class. In this case, this is no major restriction for reuse, as it is unlikely that a different interaction style has to be implemented for a PDA.

The implementation of the described tools was done by university students with some experience in object-oriented programming. In general, their first impression of the BEACH framework is its complexity and the lack of up-to-date documentation. However, without understanding the whole of the framework, it was possible for them to develop tools for roomware applications, without considering distribution issues and with little thought about supported interaction forms.

However, as with all distributed software systems there are cases when concurrency remains an issue. Although it is technically possible to **guarantee consistency**, surprising rollbacks can be very annoying for users. Due to the dependency mechanism, the output generated by the interaction model will always be consistent with the shared model. If rollbacks are received, the interaction model should nevertheless not only restore the correct state, but also provide **awareness about the actions causing conflicts**. We found some **guidelines** that should be followed when developing shared systems with the BEACH framework.

- In order to **prevent conflicts**, software developers have to take care to create a design that aims at minimizing possible conflicting interactions. One approach is a fine-grained model to detect conflicts, as provided by the COAST framework. Our experience shows that



finding a mature design in terms of concurrency sometimes needs several iterations, as for connecting the ConnecTables.

*Another example that seems simple at first glance is the ordering of components within a workspace. The components within a workspace must have a defined order used for presentation that is consistent among all clients. Our first (and obvious) approach was to use a `components` slot that stores an ordered collection. However, if users at two clients try to insert components concurrently (e.g. because they are both writing in this workspace), it happened quite often that one transaction was rolled-back. The reason was that the semantics of an ordered collection do not allow concurrent insertion of elements, as the client inserts an element at a specific position within the collection and this position was already occupied by the first transaction. We therefore improved our design to use a slot of type set. It is possible to insert elements into a set concurrently, as the set defines no order. Now, we had to define the order in a different way. We extended the position wrapper that is used inside workspaces to store a timestamp. The timestamp is set to the current value of the synchronized clock whenever an element is added to the workspace (or raised to the front). Each client, then, uses the timestamp to calculate the order of elements. Now, when two elements are added synchronously, their order is determined by the exact timestamp (in milliseconds) of the action.*

Example 9-1:  
Synchronous  
insertion into a  
workspace

This example illustrates the way developers can solve concurrency issues by changing the design. Using a shared-object space, however, both solutions ensure consistency and have the same functionality. While the first one was faster to design and implement, the latter can cope with concurrent actions. Using this approach, developers can decide how much effort to put into concurrency issues. When using BEACH, we experienced that the critical issues concerning concurrency are noticed very quickly when testing a prototype. We therefore encourage developers to use the simplest approach at first and improve the design as necessary for the intended task in subsequent iterations.

- Another pit-fall caused by the automatic detection of dependencies between interaction model and shared models that depends on the chosen design is the unintentional creation of **avoidable dependencies**. As too many avoidable updates cause decreased performance the designer should also consider which actions trigger which re-computations. Again, a fine-grained model to detect dependencies helps minimizing unnecessary computations while keeping the guaranteed consistency of the interaction model with the observed subjects.
- We found that it is essential to establish clear rules about the responsibility of using transactions. As nested transactions cause run-time errors or dead-locks, we came up with the rule that only the **interaction model is responsibility for starting and ending transactions** (in contrast to allowing all basic models to begin transactions). *Views* use transaction to record dependencies during display updates,<sup>44</sup> *controllers* to modify shared state. This turned out to be very natural, as this way, user actions are completely executed or completely canceled. In addition, this clear rule helps ensuring reusability, as it cannot happen that some method that is called within a transaction calls another method that tries to start its own transaction. This rule helps new developers, as it turned out that the problems of nested transactions are hard to understand when having little experience writing cooperative software.

To summarize, it has been shown that the separation of concerns defined by the BEACH model enables **extensibility** of data, application, and user interface models. Also, the examples illustrate situations in which different approaches for coupling and different **degrees of sharing** are useful.

↓ Next chapter

<sup>44</sup> When updating the presentation, transactions have to be used to access shared state. As during presentation update the shared state is not *modified*, a special read-only variant of transactions is available, called *display transaction*, which allows optimized transaction processing (see section 6.1.3).

## 9. Tools for Collaborative Roomware Environments

The next chapter is the final chapter of this thesis. It wraps up the main ideas and contributions of the thesis. The contributions of the thesis are evaluated to see if they fulfill all requirements. Then, alternative application areas are discussed. Finally, the questions that have remained unanswered are outlined, to indicate possible directions for future work.

## 10. Conclusions & Future Work

---

The final chapter of the thesis starts with a summary of the main ideas presented, highlighting the key contributions made. Returning to the requirements identified at the beginning of the thesis, it is discussed in detail how the contributions meet these requirements. Finally, open questions and directions for future research are discussed.

---

Approaching the end of this thesis, this chapter firstly discusses the achievements of the dissertation. In order to show that all requirements have been fulfilled the next section compares the solution developed in this thesis against the requirements identified in chapter 2. Finally, open questions that have been raised by this thesis are listed to give pointers for future work.

### 10.1. Achievements of the Dissertation

This section summarizes the main contributions of this thesis; an extensive discussion can be found in the last section of each chapter. As mentioned in the introduction, this dissertation contributes at four different levels to the state of the art: at the (1) conceptual, the (2) architectural, the (3) design, and the (4) application level (see section 1.4). These levels are used to structure this section.

#### 10.1.1 The BEACH Conceptual Model: Contributions at the Model Level

At the conceptual level, a **generic conceptual model for roomware applications**, the BEACH *conceptual model*, has been developed (chapter 4). It unifies approaches that have been developed in the relevant research areas (section 1.1): human-computer interaction, computer-supported cooperative work, and ubiquitous computing.

Summary of the BEACH conceptual model

The BEACH model is structured along **three design dimensions**: separation of concerns, coupling and sharing, and level of abstraction (section 4.1).

The first dimension, *separation of concerns*, separates **five basic concerns** represented as models: data, application, user interface, environment, and interaction models (section 4.3). The

*data model* specifies the kind of data the users can create and interact with. To work with data, an *application model* provides the necessary functionality. These two models are independent of the currently used device. Instead, available devices and other relevant context information are described by the *environment model*. The *user-interface and interaction models* define how the applications can be presented to the user, taking into account the environment model.

Second, *coupling and sharing* specifies the **degree of coupling**, and which parts of the models are shared (section 4.4). Focusing on synchronous collaboration, crucial aspects of groupware systems are *access to shared data* and *coupling of applications*, making it necessary to share data and application models among devices. In addition, information about the physical environment such as the presence of nearby users or other available devices has to be exchanged. Elements of the user interface can be distributed among complementary devices. Depending on how much state is shared, the *degree of coupling* can be controlled. If the entire user interface and editing state is shared, a tightly-coupled collaboration mode is realized; if only the same data model is shared, users work loosely coupled. By sharing models, software can provide awareness about activities, environmental changes, and user interface state.

The third dimension of the conceptual model is the *level of abstraction* (section 4.5). It is a widely used software engineering technique to separate different levels of abstraction in order to reduce the complexity at each level. The BEACH model defines **four levels of abstraction**. At the *task level*, application-dependent tailored support for tasks is given. The *generic level* provides generic functionality, which is application-independent, but domain-specific. The *model level* implements the basic separation of concerns in an application-, domain-, and platform-independent manner. Finally, the *core level* provides a platform-dependent specialized infrastructure.

#### Contributions and Discussion of the BEACH conceptual model

The BEACH conceptual model was applied to structure *related systems* in section 4.6. Demonstrating that the BEACH model can provide a structure for a range of existing systems has validated that it covers the essential aspects of ubiquitous computing applications.

To provide **guidance for developing software** for roomware environments, a conceptual application model helps define the high-level structure of software systems (chapter 4). A **unified model** is important, since future development will cross the borders of individual research areas. This relieves the application developer of the problem of trying to merge incompatible concepts that have been created for different purposes. The BEACH conceptual model is a generic model (section 4.7). Its **wide applicability** is gained by defining structural elements and common concerns at a rather high level of abstraction, such that it can be applied to many different applications and in many different contexts. It leaves **much freedom** for application developers and architects to choose approaches and architectural styles that are appropriate for the problem at hand. On the other hand, the guidance provided is necessarily at the same high level of abstraction. This implies that **no help can be given for anything specific** that “falls into the cells of the grid” opened by the model. However, this is not a problem caused by the BEACH model; rather, it is inherent in conceptual modeling.

In addition to the overall contributions, the model itself contributes to the state of the art in each **design dimension**.

Firstly, a clear *separation of concerns* gives the possibility to **adapt different aspects of a software system independently** (sections 3.1, 4.3). This is important if, e.g., two devices that require different user interfaces are used to support collaboration. Therefore, the BEACH model introduces a clear **separation of user interface and interaction** orthogonal to the level of abstraction (section 4.3.3), which has not yet been proposed in related models. This separation is important to enable tightly-coupled collaboration using different interaction styles (req. U-1, UH-1, UH-2). The structure that is imposed on software systems by the basic concerns is vertically organized. As the concerns have a defined dependency that can be sequentialized, it is

possible to make a **vertical cut through the architecture** of systems built according to the BEACH model (section 8.4). This is a key factor for improving maintainability, extensibility, and reusability of software systems in general.

The second dimension, *coupling and sharing*, provides new contributions in combination with separated concerns. Based on the separation of concerns, an **extended definition of “coupling”** can be given: Coupling can now be refined as *sharing the same interaction, user interface, application (editing), or data state* among several users or devices (section 4.4). Coupling can thus be implemented such that the same interaction, user interface, or application models are access by all participants.

Looking at the separate concerns from the sharing dimension, the **notion of “awareness” can be extended** in a similar way. Each concern leads to a different aspect of awareness if the corresponding model is shared (section 4.7). Thus, besides the *workspace awareness* that results from a shared application model, awareness can be given for the data, user interface, environment, or interaction, as soon as each model is shared.

For the third dimension, the *level of abstraction*, the three levels commonly used for software frameworks are extended by the BEACH model (section 4.5). **One additional level**, the *model level*, is introduced to define an application-, domain-, and platform-independent interface for higher-level components. The levels of abstraction define a vertical structure for software systems, in contrast to the horizontal structure defined by the concerns. As each level builds on the lower levels only, these levels allow a **horizontal cut through the architecture** of software systems built according to the BEACH model (section 8.4).

In summary, the BEACH conceptual model provides two **key contributions** to the state of the art (fig. 10-1). First, it proposes the strict *separation of user interface and interaction* concerns orthogonal to the level of abstraction that is not found in current HCI models. This is a crucial extension of HCI models that is required in the context of ubiquitous computing. Second, it introduces a new view on the concept of *sharing*. By applying the CSCW concept of sharing in the context of ubiquitous computing, sharing user interface, interaction, and environment state becomes relevant. Thereby, the concept of sharing as known from CSCW can be extended to function as a guiding principle for UbiComp application design. This novel design approach helps ensuring the extensibility and flexibility that is required in ubiquitous computing.

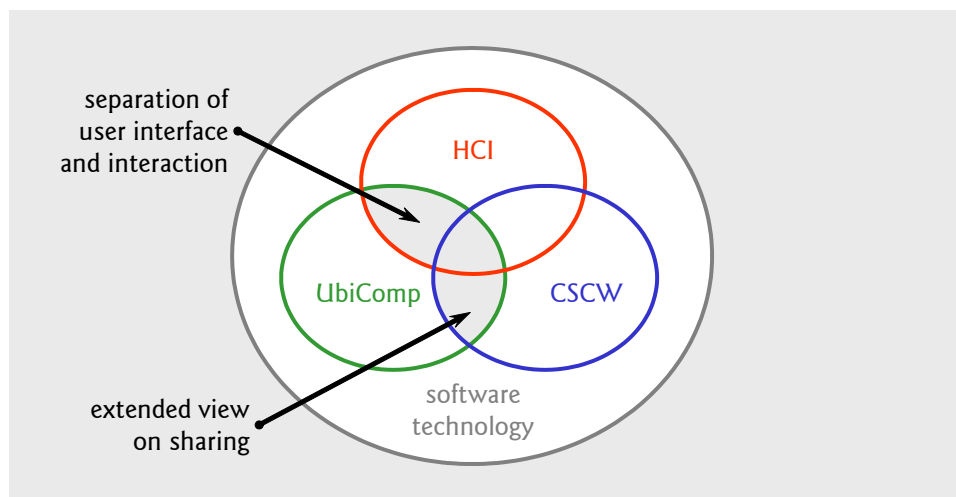


Figure 10-1. Key contributions of the BEACH conceptual model are: the separation of user interface and interaction concerns (orthogonal to the level of abstraction), and an extended view on the concept of sharing.

### 10.1.2 The BEACH Software Architecture: Contributions at the Architectural Level

At the architectural level, the **architecture for the software infrastructure for roomware applications** has been developed as part of this thesis—the BEACH software architecture (chapter 5). It applies the BEACH conceptual model in the context of roomware environments in order to provide a basic structure, but refines it to meet the concrete needs of applications supporting synchronous collaboration in *roomware* environments. The BEACH architecture, therefore, serves as example of how the BEACH model can be applied in practice. Moreover, the BEACH architecture proves that the BEACH model provides **guidance for structuring software systems** for roomware environments. It has been demonstrated that using this structure leads to an architecture that is **flexibly extensible** and that **eases the reuse** of software components. By defining **key abstractions** at several abstraction levels, the complexity of software development is reduced. The underlying realization is hidden, thus facilitating thought at the appropriate abstraction level.

In order to design the architecture, the requirements for roomware applications (chapter 2) had to be examined to identify concrete **implications for the construction of the architecture** (chapter 5). These are decisions for those parts of the conceptual model which leave freedom for the developer to choose an appropriate design. The most prominent decisions are:

- At the *model level*, a **visual interaction model** was chosen as the primary interaction style.
- At the *generic level*, the **environment model defined a representation of roomware components**. In addition, explicit support for **pen-based and multi-user interaction** was given.
- At the *core level*, a server-based, **replicated distribution architecture** was used. It provides a shared *state* in contrast to sharing *events*. **Dependencies** guarantee a consistent presentation rendered by the interaction model. **Shared objects** instead of simple models such as tuples allow for representing complex information. This is needed to be able to share all basic models: the data, application, user interface, environment, and interaction models.

### 10.1.3 The BEACH Application Framework: Contributions at the Design Level

At the design level, the BEACH architecture has been implemented as a layered **object-oriented application framework** (chapters 6 and 7). It serves as **proof-of-concept** to show how the conceptual model and architecture can be successfully applied.

Developing a framework requires experience concerning the **common parts** of typical applications in the given domain. Additionally, knowledge of **variable aspects** also plays an important role, since the framework must provide possibilities to add specialized components and behavior. They have been considered in the design of the BEACH framework. The design of the framework therefore contributes abstracted knowledge about common and variable aspects that is valuable for developers of ubiquitous computing applications as guidelines for the software design.

The BEACH framework provides **reusable components** for roomware applications to support **synchronous and informal collaboration**. The *data model* provides a **spatial hypertext** model. Spatial hypertext is appropriate for informal meeting situations. The *application model* aims to support **modeless collaboration**. **Throwing of objects** is one example of an interaction and editing style designed for the needs of roomware components. The *user interface model* defines generic **user interface elements** tailored to the properties of roomware components. It can dynamically adapt to changes in the environment. The *environment model* adds a **model of roomware** components that can be flexibly configured to reflect different settings.

The *interaction model* provides new forms of interaction with roomware components. To provide an appropriate input for roomware components, the interaction model implements an **incremental gesture recognition** algorithm, which supports provision of **immediate feedback** to the user about recognized shapes while drawing a pen stroke. The event dispatcher is extended by a new **event dispatching strategy for gesture events**. **Multi-user support** is needed for roomware components supporting concurrent input at a single device. The **display lay-**

**outer** is responsible for calculating the correct bounds and transformation for each display within the display area.

#### 10.1.4 BEACH Roomware Applications: Contributions at the Application Level

The BEACH framework has been used to develop several new interaction techniques and applications as proof-of-concept. Up to now, more than 15 software developers have used the BEACH model and framework to create 12 tools and extensions. This found that the BEACH model helps identifying reusable parts of a software system to be developed.

The *Passage* system and the *ConnecTables* use **actions with physical objects** as user input, which enable natural interaction with computers. The *audio interaction model* provides an ambient, **multi-modal presentation** of information. These interaction forms have been cooperatively developed by the members of the AMBIENTE team with major participation of the author of this dissertation. The software design has been created by the author alone. The implementation was done by the author and other team members, as noted in chapter 8.

In chapter 9, tools for roomware environments have been presented that support various collaborative tasks in roomware environments. *MagNets*, *BeachMaps*, and *PalmBeach* form a suite of **creativity support** tools. They benefit from the natural interaction with roomware components, enhancing the overall process of idea generation in a natural way. This suite has been developed together with Thorsten Prante and Carsten Magerkurth, with help for implementation from Alexander R. Krug.

Besides being a contribution on their own, the developed tools and new forms of interaction provide evidence that the BEACH conceptual model effectively supports developers to meet the requirements of roomware environments. They show that the model helps reducing the implementation effort when accompanied with appropriate software development tools such as the BEACH framework.

## 10.2. Evaluation: Comparison against Requirements

This section returns to the requirements identified in chapter 2 to show how they have been fulfilled by the conceptual model, architecture, and framework proposed in the thesis. It is shown how each requirement is fulfilled at the conceptual level. In addition, examples are given that show a concrete application of several aspects of the BEACH architecture and its implementation as the BEACH framework. An overview is given in table 10-1.

### 10.2.1 HCI Requirements

#### Requirement H-1: Different Forms of Interaction

By **separating the interaction model** from the other concerns in the conceptual model, it is possible to use a different interaction style without modifying other components. This technique has its origin in the domain of human-computer interaction models; however, existing models never clearly separate user interface and interaction concerns. In chapter 8, examples are given where this technique was applied to support interaction with physical objects (the *Passage* mechanism and the *ConnecTables*, see sections 8.1 and 8.2), as well as audio feedback (section 8.3).

#### Requirement H-2: Different User Interface Concepts

Similarly, the **separation of the user interface model** in the conceptual model creates the freedom to change the user interface independently. As an example, a new user interface concept (based on segments and overlays, see section 7.3) has been developed for the roomware components.

### 10.2.2 UbiComp Requirements

#### Requirement U-1: Multiple and Heterogeneous Devices

The basic concept that supports multiple devices is the **sharing of the models**. The handling of heterogeneity is realized by the introduction of the conceptual **model level**. It defines the **abstractions that are common for all devices**, providing opportunities for individual adaptations and extensions. Due to the separation of the five basic concerns, adaptations and extensions can be made to any one of the concerns, without interference with the others. Examples of support for heterogeneous devices are the different roomware components, presented in sections 2.1 and 7.6. PalmBeach shows how a different implementation can be used for the same conceptual model (see section 9.2).

#### Requirement U-2: Multiple-Computer Devices

To support multiple-computer devices such as the DynaWall, it is essential that all computers have access to a **shared-object space**. Data and application models must be shared to access the same data, and a shared environment model is needed in order to describe the current physical setup of the computers. In addition, the user interface model has to be shared, in order to provide a homogeneous user interface crossing the borders of the separate computers. These are two examples that highlight the need for **extending the concept of sharing** to include environment and user interface models.

In section 7.1.1, a concrete environment model for composite roomware components was presented. Distinguishing between the concepts of the physical display and the **logical display area** provides the basis on which a display area can be extended across multiple physical displays.

#### Requirement U-3: Context and Environmental Awareness

On the conceptual side, the **environment model** is introduced to model relevant context and environmental information. The **interaction model for the environment model** is used to update the model using sensors (or to modify the physical environment using actuators).

Examples where environmental information is used are the Passage mechanism and the ConneCTables (see sections 8.1 and 8.2). The presence of physical objects, e.g. a Passenger or another ConneCTable, is reported in the environment model via sensors. Dependencies are used to ensure consistency and to support notification about changes within the environment model, e.g., when the Bridge has to be shown or two ConneCTables should be connected.

#### Requirement U-4: Dynamic Configuration

Dynamic reconfiguration is possible, since the **environment model is shared** and can be modified by any client at any time. It has proven very helpful to use **constraints** to explicitly model dependencies. In the BEACH framework, the interaction model is kept up-to-date by one-way constraints that are automatically generated when executing declarative descriptions of the representation (see sections 5.5.3 and 6.1.3). In this way, the developer is freed from having to think about handling explicit change notifications, which can become very complex in a distributed environment. Constraints can also be used to trigger modifications of shared models, as in the example of the ConneCTables (see req. U-3 above or req. UH-3 below).

### 10.2.3 Combined UbiComp and HCI Requirements

#### Requirement UH-1: Adapted Presentation

To support adapted presentations for specific roomware components and devices, the interaction model can draw on information provided by the **environment model** about the local device and the context in which it is running. The implementation of the visual interaction model in the BEACH framework introduces the **view transformations** in order to adapt the lo-



**cal visual representation** sent from the different views to the graphics context (see section 6.8.3). This allows the representation to be scaled, e.g., to fit in the available space on a CommChair, or to be rotated to provide the preferred orientation for different users at an InteracTable. Depending on the constraints imposed by the available device, it would also be possible to select a different representation.

#### Requirement UH-2: Multiple-Device User Interface and Interaction

User interfaces and interaction styles including multiple devices are enabled by **sharing user interface and application models**. This supports the creation of a synchronized representation of the same data, the same editing state, and related user interface elements on different computers and using different modalities. A very simple example is a user sitting in a CommChair in front of a DynaWall (see sections 2.2 and 7.6.2). This user can interact both directly with information displayed on the CommChair and remotely with information visible at the DynaWall.

#### Requirement UH-3: Physical Interaction

Physical interaction is enabled, since the **interaction model can observe the environment model** and react to changes in the environment. Here, again, the example of the Passage mechanism and the ConnecTables can be used for illustration. As a reaction to changes made by sensor observers to the environment model (see also req. U-3), software functionality is triggered. This is shown on the virtual part of the Bridge in the Passage mechanism (see section 8.1), or in the connection of two adjacent ConnecTables to form a common, homogeneous workspace (see section 8.2).

### 10.2.4 CSCW Requirements

#### Requirement C-1: Multi-Device Collaboration

The key technique for supporting collaboration between multiple devices is a **shared-object space**. In practice, there are many different options for how a shared-object space can be implemented. The choice of an appropriate implementation depends on the requirements of the target platform. For the BEACH framework a replicated approach with incremental updates that synchronize the replicates was chosen.

#### Requirement C-2: Flexible Coupling and Modeled Collaboration Mode

Flexible strategies for coupling can be realized by **separating data, application, and user interface models** and by **sharing** not only the data, but also application and user interface models. In this way, the current collaboration mode is represented as shared objects that can be accessed and modified by any client. *Tightly-coupled collaboration* is achieved when several people use the same application model. *Loose collaboration* can be realized by using different application models for the same data model. An example of tight collaboration is given in section 7.6.2 with a CommChair and a DynaWall. Other examples are discussed in (Schuckmann *et al.*, 1999).

#### Requirement C-3: Multiple-User Devices

To support multiple users concurrently working at one device, the **core level** must be implemented in a way that supports extensions (section 6.2.1). It must be designed in such a manner that **new kinds of events and new dispatching strategies**, such as multiple-user event handling, can then be added. The BEACH Generic Collaboration framework provides a module to handle concurrent input (section 7.5.3).

### 10.2.5 Combined UbiComp and CSCW Requirement

#### Requirement UC-1: Collaboration with Heterogeneous Devices

The option of using different devices for collaboration is gained by introducing **model-based sharing** and providing a **local interaction model**. On different devices, **different user interface and interaction models** can be used while working with the same data and application model (Seitz, 1999). The **environment model** is needed to provide information about available interaction devices and about the underlying platform. For heterogeneous platforms, it is also possible to use several distinct implementations to realize a conceptual shared-object space. PalmBeach, for example, uses a different implementation for the conceptually same data model (see section 9.2).

### 10.2.6 Software Engineering Requirements

#### Requirement S-1: Generic Functionality—Reusability

The **separation of the five basic concerns** is an important factor in the reusability of software components. In addition, **separation of different levels of abstractions** also opens the possibility to reuse components at the desired level of granularity. The purpose of the conceptual **generic level** is the definition of reusable components. Therefore, this level is implemented as the BEACH Generic Collaboration framework. The generic components provided by this framework have been used to implement the new forms of interaction and tools described in chapters 8 and 9.

#### Requirement S-2: Tailorable Functionality—Extensibility

To ensure extensibility, a **clear separation of concerns** and of **levels of abstraction** is also necessary. In this way, different concerns can be extended independently of the other concerns. The **model level** provides the basis for extensions, in that it defines the abstractions that are common for all implementations.

In the context of ubiquitous computing, using **shared models as the design approach** enables extensibility in another manner. Currently, services deployed in ubiquitous computing environments usually provide access to data and application models only. If within such an environment user interface, interaction, and environment models are shared as well, new devices can be added dynamically at any time that for instance support further interaction modalities for existing user interfaces. This is essential to be able to evolve the overall system over time.

Requirement	H-1	H-2	U-1	U-2	U-3	U-4	UH-1	UH-2	UH-3	C-1	C-2	C-3	UC-1	S-1	S-2
Separation of concerns			✓											✓	✓
• Interaction	✓				✓		✓	✓	✓			✓	✓	✓	✓
• Environment					✓	✓	✓		✓				✓	✓	✓
• User interface		✓						✓			✓		✓	✓	✓
• Application								✓			✓		✓	✓	✓
• Data										✓					
Coupling & sharing			✓	✓				✓		✓					✓
• Interaction	✓						✓	✓							
• Environment															
• User interface								✓			✓		✓		
• Application								✓			✓		✓		
• Data											✓		✓		
Level of abstraction														✓	✓
• Task															✓
• Generic														✓	
• Model	✓	✓	✓										✓	✓	✓
• Core	✓									✓		✓		✓	
Dependencies					✓	✓									

Domain	Req.	Requirement Name	Domain	Req.	Requirement Name
HCI	H-1	Different Forms of Interaction	CSCW	C-1	Multi-Device Collaboration
	H-2	Different User Interface Concepts		C-2	Flexible Coupling and Modeled Collaboration Mode
UbiComp	U-1	Multiple and Heterogeneous Devices		C-3	Multiple-User Devices
	U-2	Multiple-Computer Devices	UbiComp & CSCW	UC-1	Collaboration with Heterogeneous Devices
	U-3	Context and Environmental Awareness			
	U-4	Dynamic Configuration			
UbiComp & HCI	UH-1	Adapted Presentation	SE	S-1	Generic Functionality—Reusability
	UH-2	Multiple-Device User Interface and Interaction		S-2	Tailorable Functionality—Extensibility
					✓ = requirement is fully supported
	UH-3	Physical Interaction			(✓) = requirement is partially supported

Table 10-1. Comparison of contributions against requirements. It shows that all requirements are fulfilled and that all aspects of the BEACH model are necessary.

### 10.3. Open Questions & Future Research Topics

The work that is presented in this thesis can be continued in a number of directions. For some issues, work that has been developed by others has to be integrated; for others, research is still necessary. Happily, researchers are currently addressing many related problems.

#### 10.3.1 Beyond Roomware

While the roomware components require different forms of interaction due to their different form-factors, their hardware and software platform is quite homogeneous. In general, ubiquitous computing environments contain a variety of hardware platforms. The example of Palm-Beach (section 9.2) showed that the underlying hardware implies changes in the software architecture. At the conceptual level, the BEACH model can cope with this heterogeneity. The BEACH architecture and framework, however, are designed for roomware only. Here, work is needed that addresses the **issues of heterogeneous platforms and interoperability**. The data that is available and that is worked with in ubiquitous computing environments is accessed and modified by many different applications that usually have been developed independently.

To enable this integration, the data must be structured in a way that fits all applications interested in it. While simple structures might be more **robust against evolution and different requirements** (Tandler, 2003), some data has an inherently complex internal structure that cannot be mapped easily onto simple structures such as tuples (Johanson and Fox, 2004). One approach that has recently been proposed is using **ontologies** as basis for UbiComp application development (Niemela and Vaskivuo, 2004). One aspect that is addressed by the MPACC architectural style and Gaia (section 3.3, Hess *et al.*, 2001b) is the transparent **adaptation and conversion of data formats**. It still remains a research challenge to develop and integrate other architectural styles that address heterogeneous platforms and interoperability.

The dynamic aspects of the setting that the BEACH framework is designed for are changes in collaboration modes and styles. There was no need to cope with **mobility** across different environments and migration. For instance, while running, a BEACH client cannot switch the server to which it is connected. **Dynamic changes of the environment** are investigated by the Aura project (section 3.3). Spontaneous Container Services that combine component frameworks, service discovery, and aspect-oriented programming (Popovici *et al.*, 2003) are a promising approach to meet the required flexibility on the implementation side.

When the used **device is changed**, the used application must migrate, preserving its current state. As the BEACH framework keeps all application state persistent, it is easy to stop an application at one device, and restart it at another, where the current state can be retrieved from the server. However, no support is given for **code migration**, as for example in (Grimm *et al.*, 2002). One approach to integrate code migration in the BEACH framework that is consistent with the sharing philosophy proposed by BEACH is to encapsulate code as shared large binary objects that are installed as executable code when loaded.

When devices change their environment, they have to cope with heterogeneous infrastructure and services. The *application model* needs to support **service discovery** and **mapping of abstract service descriptions** to available services. This is investigated for instance by iROS, Gaia, and Aura (section 3.3), MIT's Metaglug/Hyperglue (Coen *et al.*, 1999; Shrobe *et al.*, 2002), UC Berkeley's Ninja (Czerwinski *et al.*, 1999), and Sun's Jini (Sun Microsystems, 1999; Sun Microsystems, 2001).

The *environment model* that is provided by the BEACH framework is limited to the needs of a roomware environment. In order to be useful beyond roomware, more generic models need to be integrated. Such models are developed in a number of ongoing research projects. As discussed, these related models can be grouped into three categories: physical environment, context, and logical environment models. Models of the **physical environment** are developed, e.g., in EasyLiving or CoolTown (section 3.3) or the Steerable Interfaces project (Pingali *et al.*, 2003). Steerable interfaces, for instance, demand much higher resolution in localization and environment modeling than supported in current pervasive computing architectures. A scalable location model that is designed for mobile users to access location-based services is presented in (Roth, 2003). Support for **context awareness** is described in (Dey, 2000; Moran and Dourish, 2001; Schmidt, 2000). The user's task as an example of **logical context** is investigated in the Aura project (section 3.3). Especially, **capturing user activities and goals** is a challenging research issue.

Another consequence of the dynamic and heterogeneous nature of UbiComp environments is that it cannot be assumed that all services are always running and never fail. Therefore, researchers have started to explicitly address the problem of **failure tolerance** (Cheng *et al.*, 2002; Johanson and Fox, 2004). Our experiences with the BEACH framework show that a **shared, persistent state helps in recovering from crashes**. If all state is stored by the server and all application state is shared, a client retrieves its complete saved state as soon as it is restarted. However, the framework is currently not resistant against crashes of the server. We plan to extend it to be robust against lost connections between client and server (for instance because the server machine has gone down). In this case, the clients wait for the server to be

available again, automatically reconnect, and send all transactions that have been locally cached. However, the server is very stable and we experienced very few crashes. In general, coping with failures has to be addressed in several ways in ubiquitous computing applications. Besides the technical aspects just mentioned that try to handle software failures transparently, in UbiComp the possibility of failures has to be tackled explicitly by all parts of a software system as also failures become ubiquitous. This includes the user interface that needs to provide **awareness of broken components** in a non-obtrusive manner. The software must handle **failures as first-class entities**.

### 10.3.2 Multiple-Device Interaction & Cross-Device User Interfaces

In a ubiquitous computing environment the user interface can be enhanced when it can draw on the properties of the present devices. This includes both mobile PDAs that are carried by users and devices integrated in the environment. As the BEACH framework concentrates on roomware, it provides opportunities for extensions in a number of ways.

For the *user interface model*, a major issue is raised by device heterogeneity. For cross-device and device-independent user interfaces, it is necessary to find good **abstractions that can be mapped onto different modalities** and interaction styles in a natural way (Myers *et al.*, 2000; Olsen *et al.*, 2000b; Olsen *et al.*, 2001). This also involves the area of **dynamic user interface mapping and generation**, which is, for example, investigated by systems such as ICrafter (Ponnekanti *et al.*, 2001) and others (Banavar *et al.*, 2000; Sussman *et al.*, 2001; Nichols *et al.*, 2002). Here, new concepts and techniques are missing that need to be developed. For instance, the mapping of an abstract user interface description onto a set of available devices in a way that best supports users is still a major challenge (Pierce and Mahaney, 2004).

For *interaction models*, the focus on visual interaction has to be widened. It is especially important to investigate support for **multi-modal interaction**. The novel way of integrating different interaction modalities by sharing the user interface state needs to be combined with traditional approaches (Oviatt *et al.*, 1997; Nigay and Coutaz, 1995) and reflected in suitable software development tools. It needs to be explored how the **extensibility for new interaction forms** can be improved if the information common to different modalities is provided by a shared user interface model that acts as the mediator between different modalities.

### 10.3.3 Sharing in Ubiquitous Computing

At the *sharing dimension* of the BEACH model, current groupware frameworks and architectures have been designed with a rather homogeneous set of devices and scenarios in mind. By placing these in the context of ubiquitous computing, new requirements become relevant, as discussed in section 2.5. While BEACH started exploring collaboration issues in the UbiComp context, its focus is on roomware environments. For instance, work has to be continued to be able to support **devices without a permanent network connection**. While replication is one important aspect to allow off-line work, the technology used for BEACH (as provided by the COAST framework) does not work well if replicated documents are modified independently. In this case, it is very likely that all changes made while being off-line will be rejected when synchronizing again. Instead support for **versioning and merging** is needed that can take semantic information into account. One approach that was followed by Xerox PARC's Bayou project is to explore weakly consistent replicated databases for asynchronous collaboration that also supports devices with limited resources (Demers *et al.*, 1994; Edwards *et al.*, 1997).

### 10.3.4 Software Technology

While the previous issues focused on design and implementation in ubiquitous computing, the BEACH model can be applied for **conceptual work** in software technology as well. When applying any model for designing software systems, the knowledge about **architectural styles and design patterns** helps in the reuse of successful approaches. For the BEACH model, only a few architectural styles have been employed. To ease application development, an extensive set of

architectural styles and design patterns would be desirable. For that reason, there is a need to investigate what architectural styles and design patterns are adequate for the different basic models, on different levels of abstraction, and for different degrees of sharing. In fact, if used in this way, the BEACH model can be utilized as a **conceptual framework for structuring research in UbiComp software modeling**.

When the BEACH architecture was designed, the design was influenced by object technology. Currently, new, *advanced modeling techniques* are being developed. One example is **multi-dimensional separation of concerns** (Tarr *et al.*, 1999; Herrmann and Mezini, 2000) or **aspect-oriented programming** (Kiczales *et al.*, 1997a). Once these techniques reach maturity, it will be necessary to check the extent to which they should influence the way in which the BEACH model is mapped to the design of software architectures. Especially aspect-oriented programming seems to open elegant possibilities to separate the coupling and sharing dimension into distinct software units. An example of how software architectures can benefit from these technologies is *spontaneous containers* that can dynamically adapt software components for mobile devices (Popovici *et al.*, 2003).

### 10.3.5 Real-life Applications

A topic that has been avoided in this thesis, but which is highly relevant for real-life applications, is **security and privacy**. Ubiquitous access to personal information always involves the risk of unauthorized and unintended usage (Langheinrich, 2001; Da Campo, 2001; Shoemaker and Inkpen, 2001). For instance, if someone can walk up to any device to get access to personal data, it must be ensured that all data is completely removed as soon as the user leaves the device (Russell and Gossweiler, 2001). These issues must be handled at the core level in a secure manner.

The last point that is mentioned here is a rather general one. For ubiquitous computing to be successful in future, a **standard ubiquitous computing infrastructure** needs to be developed. This includes establishing standard protocols and data formats (Hong and Landay, 2001) and providing a reference implementation and standard development methodologies. It also includes questions about which parts of the functionality should go in the operating system of the devices, which in a middleware operating system for the environment, and which into toolkits and frameworks (Myers *et al.*, 2000, p. 23f). The ongoing research has to establish the foundation for these standardization efforts in order to be effective.

The topic addressed by this thesis belongs to an active area of research, and the long path to standardization still lies far ahead. However, as ubiquitous computing is becoming more widespread every day, the need to cope with heterogeneous devices that require multiple interaction forms, and to collaborate in ubiquitous computing environments is constantly rising. The *conceptual model*, *software architecture*, and *application framework* that have been presented in this thesis free software developers from the burden of handling all details of multiple interaction styles, and relieves them of many critical issues when dealing with synchronous collaboration. By these means, the developer can concentrate on the task of designing software at an appropriately high level of abstraction, and thus can create applications of higher quality and more flexible extensibility. Altogether, developers are enabled to investigate and create novel approaches to today's and tomorrow's UbiComp, CSCW, and HCI problems.

## Acknowledgements

Congratulations! Now, you've made it through the main part of the thesis, assuming that you belong to the large majority of people that read texts always from the beginning to the end. That is indeed an impressive job you've done, considering that you have read about 80.000 words—and this really deserves appreciation!

However, in one or another way, many people have positively influenced the process of doing my research and writing this thesis who deserve even more to be acknowledged. First of all, I want to express my gratitude to my advisors Prof. Dr. Erich Neuhold (Fraunhofer IPSI) and Prof. Dr. Brad Myers (Carnegie Mellon University). They have been extremely helpful in writing this dissertation and provided important feedback and advice. I also like to thank Brad for taking the time in his busy schedule—in addition to providing feedback—to help me with the pitfalls of the English language and to come all the way to Darmstadt for my defense.

The next who deserves special emphasis within this list is my mentor Dr. Dr. Norbert Streitz. His vision of the roomware components and his initiative to start this research in a newly founded research division of IPSI laid the foundation to work on a challenging research topic. I always enjoyed the creative and constructive atmosphere within the AMBIENTE group and the opportunity to work in such an interdisciplinary team. Norbert enabled us to keep much freedom and autonomy, allowing contributing one's individual strengths, and trusting the knowledge and assessment of the domain-specialists in his group. Nonetheless, I am especially grateful that he urged me to finally really start writing, took the time to read several drafts of my thesis, and provided valuable feedback from a non-hard-core-computer-science perspective, thus helping to improve the overall understandability.

Having already mentioned the AMBIENTE group, I want to continue thanking my current and former colleagues at IPSI. Here, I want to explicitly name those who had a major impact on the thesis and my time at IPSI (in alphabetic order and with no claim of completeness): Axel G. (for being one of the best developers I had the pleasure to work with, and for being a perfect room mate—what a pity that you moved out of our room!), Carsten M. (for the implementation of PalmBeach), Carsten R., Christian M.-T. (for implementing the non-BEACH side of the audio-support for the DynaWall), Christian S. (for being the unofficial supplementary advisor for my thesis; constructive and *honest* critics; teaching me the ancient wisdom of how to build synchronous groupware and CSCW models; hours of support for the COAST framework), Dieter B. (for support, motivation, and most of all: for re-activating my Ph.D. process when it seemed to be stalled, enabling me to submit—finally!), Friederike J. (for always taking time to chat and to listen to my worries when necessary), Hans S. (for the first implementation of MagNets), Jörg F. (for very constructive feedback on an early draft of my UbiComp'01 paper (Tandler, 2001b) that had a major impact on the structure and the understandability of the names of the BEACH model), Laura D. (for being very hard to convince (in general), and for *still* being a perfect room mate), Martin W. (you'll make it, too—soon!), Richard S. (for sharing an office; many hours of small talk and fruitful discussions; laughing nearly all the time—even without obvious reason; but most of all for finally agreeing that a shared-object space might really be helpful when implementing synchronous groupware ... it's just not that easy as it might seem at the first glance!), Shirley R. (formerly H.; for her lovely British accent and for helping me with the English language from a British perspective), Thorsten P. (for a lot of reasons, but most of all for putting a great strain on my nerves when co-writing papers by his very special way of treating submission deadlines), Torsten H. (for drawing the famous i-LAND scribble (fig 1-1), but most important: for consequently being in an excellent mood).

For the implementation of BEACH, I was in the lucky situation to be able to get support from several university students working for AMBIENTE (again in alphabetical order): Alexander K. "Krugar" (re-implementation of Mag-

## Acknowledgements

Nets with less bugs, BEACH maps, and Shadow Throw (still unpublished); and for being one of the few students, of who I believe that they *really* understood what I was trying to say when explaining my ideas; I was really amazed that he was able to give really brilliant presentations of BEACH and roomware—just two weeks after he joined AMBIENTE), Georg F. “Boeli B.” (implementing large parts of BEACH; always being able to smile; for being the first one who applied for a job offered on our web pages and bringing all his fellow students to work for us as well; and for coming back to IPSI after he has left to work in industry for some time), Jan Z. “Janman” (for all the hassle with the transformations—I’m really grateful that you never got lost in the labyrinth of differently transformed local coordinate systems, although it sometimes seemed so; implementation of the BEACH web interface), Klaus V. (implementation of the workspace query functionality), Oliver S. (gesture recognition and millions of bug-fixes and the high engagement to implement diverse other functionality and help giving presentations of our work), Sascha N. (the master of Photoshop, for the new look-and-feel of the BEACH user interface, and the well-designed icons for the BEACH model and for several other illustrations), Sascha S. (integration of the audio support), Sebastian P. “Spax” (the connect functionality for ConnecTables, the configuration module, lots of other stuff, bug-fixing, code-cleanup, refactoring), Stefan S. (for not giving up on the re-implementation of the gesture recognition algorithm), Thomas G. (implementation of a preliminary version of the BEACH model), Toni G. (improvement of PBib, see Appendix D).

The cooperation within the FOD (future office dynamics) R&D consortium was very engaging. It gave the opportunity to apply the BEACH model and software for really excellently designed roomware components (not to be compared with the first version we created ourselves ...), which developed into commercially available products in the meantime. Especially, I want to thank Frank S. and Burkhard R. at Wilkhahn for the constructive collaboration and for providing the professional pictures of the roomware components. I want to thank Michael F. who left Nemetschek and is now also working for Wilkhahn—“just” to be able to continue work within FOD after Nemetschek has left the consortium.

For giving feedback on my thesis and for discussing my ideas I also want to thank some people not working at IPSI: Arno S., Brad J. (Stanford University; inspiring discussion and valuable comments on the BEACH model; help on mapping iROS to the BEACH model), Dan R. (formerly Xerox PARC, then Apple Advanced Technology Group, then Xerox PARC, now IBM Almaden Research Center; he was one of the firsts who discussed with me the requirements and issues of software for roomware components while visiting AMBIENTE in 1997), Dieter T. (feedback from a non-IT perspective), Frank E. (for being willing to read my thesis if I had written in German), Jörg B. (Wibas GmbH; for also taking his wife’s last name), Malte F. (Wibas GmbH; for giving feedback from an industry perspective, and for stressing the importance of the layout ...), Patrick B. (formerly IPSI, then Xerox PARC, now Microsoft Research; for sharing his experiences of writing a thesis at IPSI and frequent motivation to continue: „Na, dann hau in die Tasten!“), Pierre M. (Wibas GmbH). The anonymous reviewers of the JSS special issue (Tandler, 2004) and of IEEE Micro (Tandler *et al.*, 2002b) provided very detailed and constructive feedback for the journal submissions about some topics of my thesis. They found several weak points in the presentation of the ideas.

As this thesis would have been not as nice to view as it is without all the people visible on the pictures, I want to thank: Carsten M., Carsten R., Christian M.-T., Frank S., Jochen D., Jörg G., Petra R., Thomas G., Thorsten P., Torsten H., Wolfgang R. In this context I want to express my admiration for Rike J. for providing the voice for the famous i-LAND video (Streitz *et al.*, 2002). It’s always a pleasure to listen to—and believe me, I’ve heard it nearly a million times (“incremental gesture recognition”).

Not last, but also not least, I want to thank Microsoft for the English grammar checker integrated in Word. It helped me a lot to create readable sentences. I experienced that if the grammar checker gets confused, there is a good chance that the sentence is too complicated to be understood by someone else as well. And, by the way, Word did not crash as often as I did expect. Maybe, here is also a good place to announce my little dissertation quiz. You simply have to answer the question: on how many pictures in this thesis can you see the author? (Additional bonus question: who are the two people shown on the “environment model” icon, see section 4.3?) Please email your answers; but I’m not sure yet if I will offer any prize, you’ll see. Next, of course, I want to thank the English Language for lots of nice words and being not as complicated as I first thought.

Of course, I also want to thank Marion T. for a variety of reasons (for allowing to share her beautiful last name, which I can use for my publications since summer 2000 (an additional benefit is that it can be pronounced by English-speaking people ...); for buying a new office chair for use at my desk at home, which improved the writing comfort and helped reducing the pain in my back when sitting too long in front of a computer; for much distraction while trying to write the thesis (for instance when discussing her interesting relationship to other people sharing the same last name)—but also for the encouragement to keep on writing), Fabia T. (“Postpone major life decisions until you finish your degree” (Bergin, 2002), I guess I should have read this earlier; next time I know), Michael T. (for lending me his standing desk that he does not need at the moment). However, I do *not* want to thank the city of Palm Beach (although this thesis presents a tool with this name) for all the trouble when counting the ballots during the last election of the president of the United States.

However, the opinions expressed in this thesis are the author’s own, and do not necessarily reflect the opinions of the above individuals, of Fraunhofer IPSI, or that of any other party. Thanks again to everyone!



## Appendix



## Appendix A Notations Used

The notation used in this thesis for class, object, and interaction diagrams is based on UML (Object Management Group, Inc., 2003). This appendix gives a brief summary of important aspects of the used notation, with the focus on the features added to illustrate important aspects of the BEACH framework. For the notation used to illustrate the BEACH model, refer to chapter 6 (see figure 4-2 on page 61).

Figure A-2 shows how associations between classes or between objects can be qualified. The “uses” association can also be used for modules. Dotted associations denote simplifications when classes or objects that are not essential in this context are not shown.

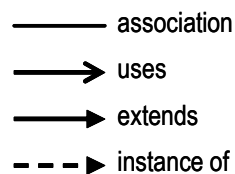


Figure A-2. Associations between classes

If not specified otherwise, an association denotes an association between two objects (or two instances of the classes shown). If an association exists between multiple instances of a class, this is specified as shown in figure A-3. For instance, a station can have zero, one, or multiple associated devices (fig. A-3a); a Passenger object can have a single associated workspace (fig. A-3b), i.e. the cardinality of the associated workspace application model is zero or one.

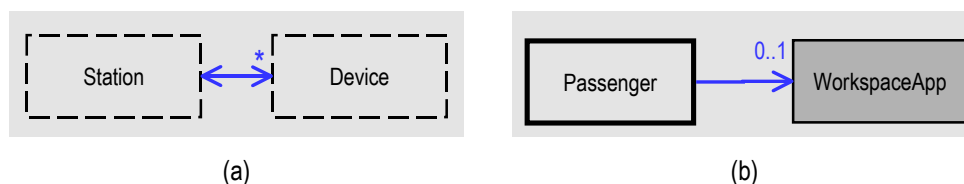


Figure A-3. Cardinality of associations. (a) The star marks an arbitrary number of associated instances, which can be zero. (b) A range can be used to specify the cardinality.

In section 3.1 the extension mechanism is introduced that allows adding features by another module to a class (see also example 6-1 in section 6.1.2 on page 104). To distinguish it from normal inheritance, the associations denoting the extension mechanism are marked with “<<extends>>” (fig. A-4).

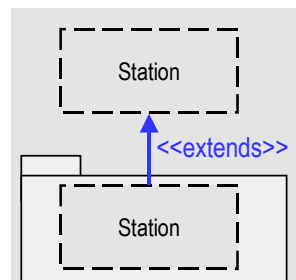


Figure A-4. To distinguish the extensions made by inheritance from extensions made to the same class in another module, the later case is marked with “<<extends>>”.

The BEACH model separates five basic concerns (see section 4.3) that are represented by five models in the BEACH architecture and framework. When a concept represents aspects that belong to different concerns, the extension mechanism can be used to clearly separate the concerns in the design while keeping a single class to represent the concept. For example, the display area is a concept introduced in the environment model to represent the available display space (see section 5.3). It is extended in the user interface model, as the display area also defines the space that is available in the user interface (see fig. A-5).

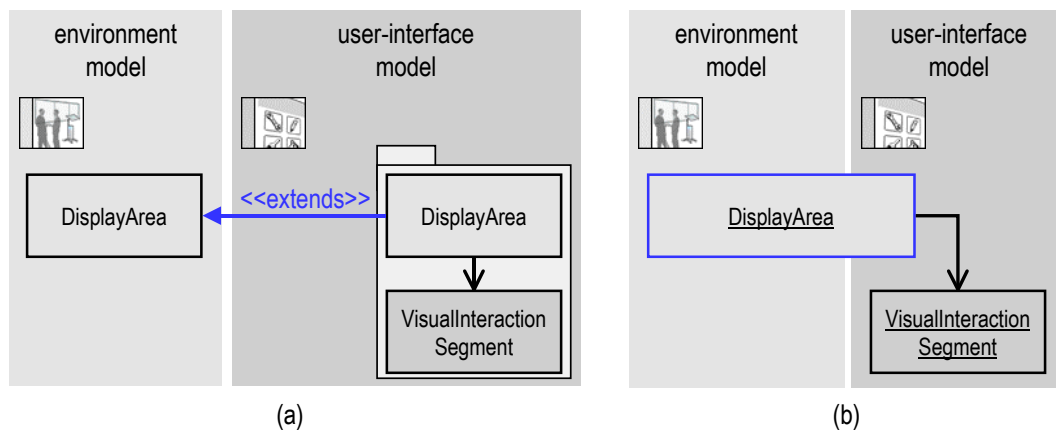


Figure A-5. (a) The extension mechanism is also used to extend a concept into a model representing another concern. (b) To simplify class and object diagrams a shorthand notation is used.

In the BEACH architecture and framework dependencies can be used (see section 5.5.3). If the dependency mechanism is used to compute associations between objects, it is possible that associations are automatically updated whenever any of the associations used in this computation is changed. These dependencies can be visualized in class and object diagrams (fig. A-6).

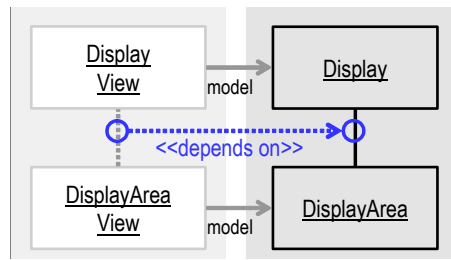


Figure A-6. When the dependency mechanism is used to compute associations, the dependencies to other associations can be visualized.



## Appendix B Abbreviations

ALV	<i>Abstraction-link-view</i> (Hill <i>et al.</i> , 1994), an architectural style for multi-user applications, see section 3.4, page 50.
API	<i>Application programming interface</i>
BEACH	<i>Basic environment for active collaboration with hypermedia</i> , the project working on ubiquitous computing software supporting synchronous collaboration with multiple heterogeneous devices (the topic of this thesis). See also: <i>BEACH model</i> , <i>BEACH architecture</i> , and <i>BEACH framework</i> .
C-2	<i>Chiron-2</i> , see section 3.4, page 52.
CC	<i>CommChair</i> , a roomware component, see section 2.1.
CSCW	<i>Computer-supported cooperative work</i> , see section 2.5.
CT	<i>ConnecTable</i> , a roomware component, see section 8.2.
DW	<i>DynaWall</i> , a roomware component, see section 2.1.
HCI	<i>Human-computer interaction</i> , see section 2.3.
i-LAND	<i>Interactive landscape for creativity and innovation</i> , the ubiquitous computing environment at Fraunhofer IPSI, which was used as a test bed for BEACH, see section 1.2.
iROS	<i>Interactive room operating system</i> (Johanson <i>et al.</i> , 2002a), see section 3.3, page 46.
IT	<i>Information technology</i> or <i>InteracTable</i> (a roomware component, see section 2.1), depending on the context.
MCRpd	<i>Model-controller-physical/digital representation</i> (Ullmer and Ishii, 2000) is an extension of MVC for tangible user interfaces, see section 3.2, page 40.
MIDI	<i>Musical instrument digital interface</i>
MMUI	<i>Multi-machine user interface</i> , see <i>cross-device user interface</i> .
MPACC	<i>Model-presentation-adapter-controller-coordinator</i> (Román and Campbell, 2001) is an extension of MVC for interactive rooms, see section 3.3, page 46.
MVC	<i>Model-view-controller</i> (Krasner and Pope, 1988a), an architectural style that separates input ( <i>controller</i> ) and output ( <i>view</i> ) from the functional core ( <i>model</i> ), see section 3.2, page 38.
OS	<i>Operating system</i>

PAC	<i>Presentation–abstraction–control</i> (Nigay and Coutaz, 1991), an architectural style, see section 3.2, page 39.
PDA	<i>Personal digital assistant</i> , a small and mobile information appliance, i.e. a very small special-purpose computer, such as a Palm.
req. *-*	Denotes <i>requirements</i> defined in this thesis (chapter 2): H-1, H-2 – HCI requirements (section 2.3) U-1, U-2, U-3, U-4 – UbiComp requirements (section 2.4) UH-1, UH-2, UH-3 – UbiComp and HCI requirements (section 2.4) C-1, C-2, C-3 – CSCW requirements (section 2.5) UC-1 – UbiComp and HCI requirement (section 2.5) S-1, S-2 –Software engineering requirements (section 2.6)
RW	<i>Roomware</i> , see section 2.1
RWC	<i>Roomware component</i> , see section 2.1
SE	<i>Software engineering</i>
SW	<i>Software</i>
TUI	<i>Tangible user interface</i> , see sections 2.4.2 and 3.2.1.
UbiComp	<i>Ubiquitous computing</i> , a term coined by Weiser (1991).
UI	<i>User interface</i>
WIMP	<i>Windows, icons, menus, pointing</i> (Beaudouin-Lafon, 2000), a metaphor for the traditional user interface of the desktop PC, see section 2.3.



## Appendix C Glossary

Terms that are related to BEACH or COAST are marked using brackets.

abstraction level	See <i>level of abstraction</i>
application framework	An application framework is a <i>framework</i> for complete applications in a specific domain (Buschmann <i>et al.</i> , 1996). Sometimes called enterprise application framework (Fayad <i>et al.</i> , 1999).
application model [BEACH]	The <i>BEACH model</i> separates five basic concerns, see section 4.3. The <i>application model</i> provides functionality to work with, create, and modify the information that is defined by the data model. To ensure the reusability of the application model in ubiquitous computing environments, it is independent of the current environment, such as available interaction devices.
architectural style	Software architectures are often described by a set of components, connectors, and additional constraints or properties (Gregory D. Abowd <i>et al.</i> , 1993; Garlan, 2001). An <i>architectural style</i> suggests a vocabulary of component and connector types, together with a topology of how they are combined (Bass <i>et al.</i> , 1999; Perry and Wolf, 1992; Phillips, 1999; Gregory D. Abowd <i>et al.</i> , 1993). See also section 3.1.
architecture	See <i>software architecture</i>
BEACH application framework	See <i>BEACH framework</i>
BEACH architecture	The <i>BEACH architecture</i> is the software architecture for roomware environments. It defines the software components that are necessary to use roomware components. It is based on the <i>BEACH model</i> and tailors the conceptual model to fit to the concrete needs of applications supporting synchronous collaboration taking the example of roomware environments. It is one of the main contributions of this thesis, see chapter 5.
BEACH conceptual model	See <i>BEACH model</i>

BEACH framework	The <i>BEACH framework</i> is an implementation of the BEACH architecture as an <i>object-oriented framework</i> . It constitutes the software infrastructure for roomware environments, providing services and reusable software components. Using this framework, developers can implement applications that acknowledge the specific properties of roomware environments much more efficiently. The design of the framework reflects abstracted knowledge about common and variable aspects of ubiquitous computing applications. It is one of the main contributions of this thesis. The design and properties of the BEACH framework are explained in chapters 6 and 7.
BEACH model	The <i>BEACH model</i> is a generic application model providing a structure for all kinds of applications for roomware and ubiquitous computing environments. It is one of the main contributions of this thesis. See chapter 4.
BEACH software architecture	See <i>BEACH architecture</i>
black-box framework	Depending on the techniques used to create extensions and to add application-specific behavior, frameworks can be classified into white-box and black-box frameworks (Fayad, 1999; Gamma <i>et al.</i> , 1995; Johnson and Foote, 1988). <i>Black-box frameworks</i> define interfaces for components that can be integrated. Instead of using inheritance as main extension technique, they rely on composition and delegation, which is often easier to use for application developers. See also: <i>white-box framework</i> .
class library	A <i>class library</i> is a collection of utility classes (Johnson and Foote, 1988), for example several “container” classes, such as <code>Set</code> , <code>Bag</code> , <code>Dictionary</code> , or <code>OrderedCollection</code> . It refines the concept of a <i>library</i> by encapsulating algorithms in (utility) classes.
cluster [COAST]	In COAST, shared objects are grouped into <i>clusters</i> that represent the atomic entities for replication. See section 6.1.3.
component	See <i>software component</i>
composite pattern	Using the <i>composite pattern</i> , objects that form a hierarchy can be access in a uniform way (Gamma <i>et al.</i> , 1995). See section 6.3.
computed slot	See <i>slot</i> , <i>virtual</i>
conceptual architecture	In this thesis, <i>conceptual architecture</i> is used to refer to the <i>conceptual model</i> of software architecture, see <i>conceptual model</i> .
conceptual model	In this thesis, <i>conceptual model</i> is used as defining the very high-level structure of an application (Phillips, 1999, p. 3; Coutaz, 1997, p. 5). See also chapter 4.
core level [BEACH]	The <i>BEACH model</i> defines four levels of abstraction. The <i>core level</i> defines the platform-dependent low-level infrastructure, see section 4.5. It encapsulates the platform-dependent issues in order to ensure portability and reusability.
coupling [BEACH]	In this thesis, Dewan’s definition of coupling (section 2.5, page 29) is refined based on the <i>separation of concerns</i> dimension of the <i>BEACH model</i> . <i>Coupling</i> can now be defined as <i>sharing the same interaction, user interface, environment, application (editing), or data state</i> among several users or devices.
coupling and sharing	<i>Coupling and sharing</i> is the second dimension of the <i>BEACH model</i> (section 4.4). Whenever multiple devices are involved in a software system, the question arises of which parts of the system should be local to a device or shared between several. This has to be clarified for both the application code and its state. While <i>distributing code</i> among devices is a technical question unique to

	every application, <i>sharing state</i> has conceptual implications. Depending on the application context of a software system, none, some, or all models defined for the five basic concerns (see <i>separation of concerns</i> ) have to be shared.
cross-device user interface	A multi-machine or <i>cross-device user interface</i> dynamically distributes elements of the user interface across several interaction devices, such as public displays or handheld devices (see section 2.4.2).
data model [BEACH]	The BEACH model separates five basic concerns, see section 4.3. The <i>data model</i> specifies the kind of data the users can create and interact with. It is independent of the currently used devices.
display area [BEACH]	A roomware component can have several attached displays that form a homogeneous <i>display area</i> , see section 7.1.1. An example is the DynaWall, which is described in section 2.1.
display transaction [COAST]	See <i>transaction</i> , <i>display</i>
distribution architecture	<i>Distribution architectures</i> describe the run time distribution of system state and components across computing platforms connected by a network. (Phillips, 1999). It addresses issues such as client/server or replication.
domain model	The term <i>domain model</i> is sometimes used for the concept of a <i>data model</i> (ParcPlace-Digitalk, Inc., 1995; Schuckmann <i>et al.</i> , 1999). The word "domain" stresses that the model represents artifacts of a given application domain, which may not be necessarily documents, see section 4.3.
environment model [BEACH]	The BEACH model separates five basic concerns, see section 4.3. The <i>environment model</i> describes relevant context information, such as available devices, physical environment, and logical context.
frame [COAST]	<i>Frame</i> is the name for objects in the COAST framework that contain <i>slots</i> to store values that can be shared among distributed machines, see section 6.1.3.
framework	See <i>software framework</i>
frozen spot	To be useful for many applications within the same domain, a framework has to support the parts common to the applications within a domain. These parts are called <i>frozen spots</i> (Pree, 1999). See also: <i>hot spot</i> .
generic level [BEACH]	The <i>BEACH model</i> defines four levels of abstraction. The <i>generic level</i> defines application-independent reusable functionality for a given domain, see section 4.5.
groupware	<i>Groupware</i> is a "computer-based system that supports groups of people engaged in a common task (or goal) and that provides an interface to a shared environment" (Ellis <i>et al.</i> , 1991). The term <i>groupware</i> was initially coined by Peter and Trudy Johnson-Lenz in 1981 (Johnson-Lenz and Johnson-Lenz, 1994). See also section 2.5.
hook	See <i>hot spot</i>
horizontal structure	The <i>horizontal structure</i> of a software system is defined by the <i>separation of concerns</i> , see sections 3.1 and 4.3.
hot spot	The parts of a framework that are designed to be extended are called <i>hot spots</i> (Schmid, 1997; Schmid, 1999; Pree, 1999) or <i>hooks</i> (Froehlich <i>et al.</i> , 1997; Froehlich <i>et al.</i> , 1999). See also: <i>frozen spot</i> .
interaction model [BEACH]	The <i>BEACH model</i> separates five basic concerns, see section 4.3. The <i>interaction model</i> defines how users and the software can communicate. This includes



the specification of how information is presented to the user and how the user can invoke functionality. Typically, the interaction model describes how the user interface is rendered onto the screen and what happens if users click buttons of a mouse or press keys. By strictly separating interaction issues from the rest of software systems, other forms of interaction can be employed without needing to change existing code.

The interaction model as it is defined here extends existing definitions. While Beaudouin-Lafon (2000) places the interaction model at the meta-level only, in this thesis, the interaction model also refers to the *instantiation* of Beaudouin-Lafon's definition, i.e. the set of *concrete* interactions that are possible for a given software system. Dewan and Choudhary (1995) defines an interaction model as describing the nature of the state maintained by the user interface. In contrast, we see the interaction model as describing the means for how to interact with the user interface.


layer, software layer	A <i>layer</i> is a software technique to structure software architecture and can be used to reflect different levels of abstraction in architecture and implementation. The term <i>level</i> is used in contrast to <i>layer</i> to denote a conceptual level of abstraction, see section 4.5.
level (of abstraction)	Separating software into <i>levels of abstraction</i> is a common software engineering technique, see section 3.1. It reduces the complexity of each level (Nigay and Coutaz, 1991) and ensures interoperability (Hong and Landay, 2001). Introducing levels of abstraction into a software system is seen as its <i>vertical</i> structure.  The four levels of abstraction that constitute the third dimension of the <i>BEACH model</i> are discussed in section 4.5. The <i>BEACH model</i> proposes to separate four conceptual abstraction levels: the <i>core</i> , <i>model</i> , <i>generic</i> , and <i>task level</i> .
library	A <i>library</i> is a generalized set of related algorithms (Hong and Landay, 2001). Examples are code for manipulating strings or for performing mathematical calculations. The focus is on code reuse. See also: <i>class library</i> .
local model	A <i>local model</i> (in contrast to a <i>shared model</i> ) is a model that is local to a single computer, thus it defines <i>local classes</i> .
local object or class	An object is called <i>local</i> if it is local to a single computer, i.e. it can be accessed only by processes running on this computer. Objects that can be accessed by multiple computers are called <i>shared</i> objects, see section 3.4.
local station or roomware component	The computer a <i>BEACH</i> client is running on is referred to as its <i>local station</i> (see section 6.6). The <i>local roomware component</i> is the roomware component to which the local station belongs (see section 7.1).
MagNets [BEACH]	“Magnetic cards to form idea-Networks” (Prante <i>et al.</i> , 2002), a software application built on top of the <i>BEACH</i> framework supporting collaborative generation and structuring of ideas in creativity sessions. It provides enhanced interaction mechanisms by analogy to the magnetism metaphor, see section 9.1.
model	A <i>model</i> is an abstraction of a system (ter Hofte, 1998, p. 61). A model focuses on aspects of a system that are considered relevant in a specific context. Jacobsen (2000), p. 10 regards the activity of model building as a part of the more general activity of problem solving. See also chapter 4.
model level [BEACH]	The <i>BEACH model</i> defines four levels of abstraction. The <i>model level</i> defines application- and domain-independent abstractions to ensure platform-independent separation of concerns, see section 4.5.



module	<p>A <i>module</i> is a conceptual entity of a software system. Often a synonym for <i>component</i> or subsystem (Buschmann <i>et al.</i>, 1996).</p> <p>The <i>BEACH framework</i> uses the term <i>module</i> to refer to software components that can be plugged into the framework, see section 6.1.2.</p>
multi-machine user interface	See <i>cross-device user interface</i> .
optimistic transaction [COAST]	See <i>transaction, optimistic</i>
PalmBeach [BEACH]	<i>PalmBeach</i> (Prante <i>et al.</i> , 2002), a creativity application for the Palm, is focused on collecting ideas in between scheduled sessions, using small, mobile devices. See section 9.2.
Passage [BEACH]	<i>Passage</i> (Konomi <i>et al.</i> , 1999) is a mechanism for establishing relations between physical objects and virtual information structures. See section 8.1.
pervasive computing	The concept of <i>pervasive computing</i> implies that the computer has the capability to obtain information from the environment in which it is embedded and utilize it to dynamically build models of computing (Lyytinen and Yoo, 2002).
pessimistic transaction [COAST]	See <i>transaction, pessimistic</i>
policy	See <i>strategy pattern</i> .
reference model	A <i>reference model</i> specifies the complete structure of some class of systems at a relatively large granularity (Phillips, 1999; ter Hofte, 1998). Bass <i>et al.</i> (1999) further distinguish between a <i>reference model</i> , seen as the division of functionality, and a <i>reference architecture</i> , which maps a reference model on software components. A reference model shows the conceptual structure with its fundamental functional elements. It should be possible to map all systems of this class to the structure defined by the reference model. See also section 3.1.
roomware	<p><i>Roomware</i> (Streitz <i>et al.</i>, 1997; Streitz <i>et al.</i>, 2001) refers to the integration of room elements with information technology.</p> <p>See also: <i>roomware component</i>.</p>
roomware component	A <i>roomware component</i> is a room element with integrated information technology, such as an interactive table, wall, or chair.
roomware environment	In this thesis, a <i>roomware environment</i> refers to a room or building that is equipped with roomware components. These rooms and building provide the necessary infrastructure for roomware applications.
separation of concerns	<p><i>Separation of concerns</i> is a principle to structure a software system. Abstractions are defined at the <i>same level of abstraction</i>, in order to simplify a problem (Jacobsen, 2000). This structure is also called <i>horizontal decomposition</i>.</p> <p>Separation of concerns is the first dimension of the <i>BEACH model</i>, see section 4.3. The <i>BEACH model</i> proposes to separate five basic concerns: <i>data, application, user interface, environment, and interaction models</i>.</p>
shared model	A <i>shared model</i> (in contrast to a <i>local model</i> ) is a model that is shared by multiple computers, thus it defines <i>shared classes</i> .
shared object or class	An object is called <i>shared</i> if it can be accessed and modified by several distributed computers. A class is called <i>shared</i> if all its instances are shared objects, see section 3.4. Objects that belong to a single computer only are called <i>local</i>

	objects.
<b>shared-object space</b>	The set of all shared objects forms a shared-object space, see section 3.4.
<b>slot [COAST]</b>	A <i>slot</i> stores values that can be shared among distributed machines. In addition to storing values, slots are capable of carrying supplementary semantics such as bi-directional references and cardinality constraints. See section 6.1.3, page 106.
<b>slot, computed</b>	See <i>slot, virtual</i>
<b>slot, virtual [COAST]</b>	A <i>computed</i> or <i>virtual slot</i> is a slot that does not store a value directly; it rather computes and caches it based on the values of other slots. In COAST, re-computation of virtual slots is triggered automatically. See section 6.1.3, page 108.
<b>software architecture</b>	<p>The <i>software architecture</i> defines the high-level structures for software systems (see section 3.1). In the literature, several definitions are given:</p> <p>The software architecture of a program [...] is the structure or structures of the system that comprise software components, the externally visible properties of those components, and the relationships among them (Bass <i>et al.</i>, 1999).</p> <p>While there are numerous definitions of software architecture, at the core of all of them is the notion that the architecture of a system describes its gross structure. (Garlan, 2000).</p> <p>The architecture of a software system defines that system in terms of components and of interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system (Shaw <i>et al.</i>, 1995).</p>
<b>software framework</b>	<p><i>Software frameworks</i> allow the reuse of implemented software architectures, offering specific support for extensibility (see section 3.1). In contrast to <i>class libraries</i> they focus on design reuse, not code reuse only. Often, a framework defines the main flow of control (Fayad, 1999). In literature, several definitions are given:</p> <p>A <i>framework</i> is a software architecture, often object-oriented, that guides the programmer so that implementing user interface software is easier (Myers, 2003).</p> <p>An object-oriented abstract design is called a <i>framework</i>. It consists of an abstract class for each major component (Johnson and Foote, 1988).</p> <p>A <i>framework</i> provides the basic structure for a certain class of applications. In contrast to <i>libraries</i> the focus is more on design reuse (Hong and Landay, 2001).</p> <p>A <i>framework</i> is a semi-finished software (sub-)system intended to be instantiated. It defines the architecture for a family of systems and provides basic building blocks to create them. It also defines the parts of itself that can be adapted to achieve a specific functionality (Buschmann <i>et al.</i>, 1996).</p> <p>A <i>framework</i> is a reusable, semi-complete application that can be specialized to produce custom applications (Fayad <i>et al.</i>, 1999).</p> <p>See also: <i>application framework</i></p>
<b>software infrastructure</b>	A <i>software infrastructure</i> is a set of technologies that acts as a foundation for other systems (Hong and Landay, 2001). The term <i>software infrastructure</i> is

	used in this thesis to describe everything that is needed be able to build and execute applications within their destination environment. This includes the <i>software architecture</i> and an implementation of this architecture that can be used as a platform to easily develop applications, e.g. an <i>application framework</i> (see fig. 1-3).
software layer	See <i>layer</i>
station [BEACH]	The term “station” refers to computers running a BEACH client. Stations can have attached devices, such as displays, keyboards, pens, or sensors, see section 6.6.
strategy pattern	The <i>strategy pattern</i> enables to exchange algorithms by encapsulating the algorithm in a strategy object (Gamma <i>et al.</i> , 1995). Also known as <i>policy</i> .
Tangible user interface	<i>Tangible user interfaces</i> (TUIs) include physical objects in the user interface, see sections 2.4.2 and 3.2.1.
task level [BEACH]	The <i>BEACH model</i> defines four levels of abstraction. The <i>task level</i> is concerned with application-specific support for specific tasks, see section 4.5. It covers abstractions that are unique to small application areas and are not likely to be further refined.
toolkit	In literature several different definitions of <i>toolkits</i> are given: Myers (2003) defines a user interface <i>toolkit</i> as a library of widgets that can be called by application programs. Hong and Landay (2001) say that <i>toolkits</i> build on frameworks, offering a large number of reusable components for common functionality. Using the terminology of this thesis, this is referred to as a <i>black-box framework</i> . According to Johnson and Foote (1988), a <i>toolkit</i> is an object-oriented application construction environment, i.e. a collection of high-level tools that allow a user to interact with an (usually black-box) application framework to configure and construct new applications. All toolkits are based on one or more frameworks. In this thesis, the term <i>toolkit</i> is used describe a software component or a collection of components that help build applications. If the support is given for the user interface and interaction part of applications, the term <i>user interface toolkit</i> is used.
transaction [COAST]	A <i>transaction</i> groups slot accesses that have to be executed atomically to avoid inconsistencies. See section 6.1.3, page 107.
transaction, display [COAST]	A <i>display transaction</i> is used by COAST for accessing slot values during display updates. During display transactions no modifications of slot values, but only read-accesses are allowed, allowing faster processing of display updates, see section 6.1.3, page 108.
transaction, optimistic	For <i>optimistic transactions</i> , the client does not wait for the server’s commit, in order to speed-up interaction. When the transaction is rejected by the server, the client has to rollback this transaction and all others that rely on its changes, see section 6.1.3, page 108.
transaction, pessimistic	In contrast to an optimistic transaction, a <i>pessimistic transaction</i> blocks the client process that executes the transaction until the commit (or cancel) is received. See section 6.1.3, page 108.
ubiquitous computing	<i>Ubiquitous computing</i> is a term coined by Weiser (1991), describing the vision that computers vanish into the background. Heterogeneous devices complement each other to provide a consistent usage experience. User interfaces will

	take advantage of the different properties of the devices. The devices will be closely interconnected and integrated with the environment and context in which people use them.
<b>ubiquitous computing environment</b>	A <i>ubiquitous computing environment</i> consists of multiple heterogeneous networked devices. It is essential that the devices are not treated in isolation. Instead, they must always be seen related with other devices, users, and objects that are part of their current environment, see section 2.4.
<b>user interface model [BEACH]</b> 	The <i>BEACH model</i> separates five basic concerns, see section 4.3. The <i>user-interface model</i> deals with everything that is needed to describe the user interface of an application, but without the application-specific parts. Typical examples are windows, menus, scrollbars, and toolbars. The user interface model, however, does <i>not</i> describe how these concepts are presented to the user or how the user can interact, as these issues are modeled separately by the <i>interaction model</i> . In order to be able to adapt the user interface to the current context, such as available interaction devices, the user interface model draws upon information provided by the <i>environment model</i> .
<b>user interface toolkit</b>	See <i>toolkit</i>
<b>vertical structure</b>	Introducing <i>levels of abstraction</i> into a software system is seen as its <i>vertical structure</i> , see sections 3.1 and 4.5.
<b>virtual slot</b>	See <i>slot, virtual</i>
<b>white-box framework</b>	Depending on the techniques used to create extensions and to add application-specific behavior, frameworks can be classified into white-box and black-box frameworks (Fayad, 1999; Gamma <i>et al.</i> , 1995; Johnson and Foote, 1988). <i>White-box frameworks</i> use the techniques of object-oriented languages to add extensions. Typically, sub-classes can be derived from dedicated base-classes, which provide a set of methods to be overridden and refined. This requires a detailed understanding of the framework's architecture. See also: <i>black-box framework</i> .
<b>wrapper</b>	<i>Wrappers</i> add behavior to other object using the decorator pattern (Gamma <i>et al.</i> , 1995). As an advantage over class extensions or sub-classing, wrappers can be added dynamically to <i>individual</i> objects in contrast to adding functionality to the <i>class</i> . Besides offering a hook for extensions, wrapper objects also increase reusability as the same add-on functionality can be reused in different contexts, see section 6.3.



## Appendix D Development Tools Used

This appendix describes the development tools that were used in the context of this dissertation.

BEACH is implemented in VisualWorks Smalltalk (Cincom, 2002). ENVY (from OTI) was used as versioning system. ENVY also adds the possibility of class extensions to VisualWorks Smalltalk. For design, we used a cooperative UML editor, which is also written in Smalltalk and with COAST. It is available from <http://www.opencoast.org/>.

The thesis was written with Microsoft Word (see Acknowledgements). The bibliographic database was first stored in an Adabas D database (Software AG, 2004), then using MySQL (MySQL AB, 1995-2004). The bibliography is available online at <http://tandlers.de/peter/beach/> in BibTeX format.

To edit the database, first StarOffice (Sun Microsystems, 2004), and later OpenOffice (OpenOffice.org, 2004) was used. To format the references in the document and to compile the References section, I wrote a couple of Perl scripts, called "PBib". They work similar to the classical BibTeX, but can be extended for arbitrary bibliography databases (including BibTeX files), arbitrary bibliography styles (e.g. ACM, IEEE), and arbitrary document formats (including RTF and OpenOffice). To browse the bibliography database and the references used in a document, I implemented a simple user interface in Perl/Tk.

PBib is available online at <http://tandlers.de/peter/pbib/>.



## Figures

Figure 1-1. In ubiquitous computing environments, heterogeneous devices are used to support synchronous collaboration: (a) an interactive wall with a large visual interaction area, (b) interactive chairs or handhelds with small displays, (c) a horizontal tabletop surface. The drawing is based on a vision scribble we created in early 1997.....	2
Figure 1-2. Related research areas for the design of collaborative ubiquitous computing applications .....	4
Figure 1-3. Relationship between conceptual model, architecture, framework, and infrastructure that are developed as part of this thesis. The conceptual model (1) provides the structure for the software architecture (2). The application framework (3) implements the architecture. Roomware applications (4) use the application framework. The numbers refer to the four levels of contributions.....	7
Figure 2-1. Two people working at the DynaWall .....	17
Figure 2-2. (a) The initial two prototypes of CommChairs developed by IPSI. (b) The redesigned version of the CommChair being part of the second generation of roomware components that have been created by IPSI and Wilkhahn/Wiege within the “Future Office Dynamics” R&D consortium.....	17
Figure 2-3. (a) First prototype of the InteracTable, developed by IPSI. (b) Second version of the InteracTable, which is commercially available. It allows collaboration of up to six co-located people. ....	18
Figure 2-4. Tight collaboration, using a CommChair in front of a DynaWall. The user in the chair can remotely interact with information displayed at the wall by using the display attached to the chair. In addition, he can access his personal workspace. ....	19
Figure 2-5. To support horizontal displays, BEACH lets different users rotate documents to a preferred orientation. For collaboration, a second view of a document can be opened that remains synchronized with the original. ....	20
Figure 3-1. Arch reference model (Kazman and Bass, 1995) .....	38
Figure 3-2. Model–View–Controller.....	38
Figure 3-3. Reference architecture of PAC-AMODEUS (Nigay and Coutaz, 1991) .....	39
Figure 3-4. Layers defined in Amulet (Myers <i>et al.</i> , 1997) .....	42

Figure 3-5. Dewan's generic architecture (Dewan, 1999)..... 50

Figure 4-1. Contributions to collaborative ubiquitous computing applications by the different relevant research areas, extending figure 1-2..... 60

Figure 4-2. Notation for the three design dimensions of the BEACH conceptual model..... 61

Figure 4-3. Dependencies between data, application, environment, user interface, and interaction models. The user interface model can draw on information available in the environment model to define an application's user interface that is adapted to the current environment. .... 62

Figure 4-4. The text editor sample application's interaction model visualizes information that is defined by data, application, user interface, and environment models..... 63

Figure 4-5. The environment model provides information about the platform, available devices, the surrounding physical environment, and the logical context. This may include information about the presence of other people and their tasks. .... 66

Figure 4-6. A shared data model is essential for synchronous collaboration. .... 70

Figure 4-7. Local application models enable separate cursor positions, but no awareness and coupling. (a) Sharing the application model enables tightly-coupled collaboration and activity awareness. (b) If separate instances of the shared application model are used for different clients, some aspects of collaboration can be loosely coupled. .... 71

Figure 4-8. Local user interface models enable independent window and scroll positions, but no awareness and coupling. (a) Shared user interface can cross the boundaries of several devices. (b) Independent instances of shared user interface models allow for awareness. As the user interface relies on context information, the environment model must also be shared. .... 72

Figure 4-9. Local environment models can represent the local platform and devices, but not much of the surrounding context. A shared environment model allows drawing on information detected by remote devices and adapting to a broader context..... 73

Figure 4-10. Local interaction models enable independent presentation and user control for each device. While the complete interaction model cannot be shared, sharing some of its parts enables disaggregated computing. .... 75

Figure 4-11. Four conceptual levels of abstraction: core, model, generic, and task level ..... 76

Figure 4-12. The different categories of software frameworks can be used to implement different parts of an application that is structured according to the BEACH conceptual model..... 78

Figure 4-13. Comparison with Model–View–Controller..... 79

Figure 4-14. Comparison with PAC-AMODEUS: The dialogue component includes aspects of interaction, user interface, and application models. PAC-AMODEUS has no notion of environment model. .... 80

Figure 4-15. Comparison with MPACC (model, presentation, adapter, controller, coordinator): Adapter and coordinator are components introduced to support UbiComp environments. .... 80

Figure 4-16. Using the BEACH conceptual model to structure the iROS meta-operating system ..... 81

Figure 5-1. Following the BEACH model, the software architecture is vertically organized in four layers defining different levels of abstractions and horizontally by five models separating basic concerns. Crucial for synchronous collaboration is the shared-object space provided by the core layer..... 88

Figure 5-2. Components of the model layer. Every basic concern identified in the conceptual model is implemented by a corresponding component. .... 89

Figure 5-3. Generic components for roomware environments that are defined as part of the BEACH architecture. The dotted boxes represent the functionality within a software component that belongs to a specific model. .... 91

Figure 5-4. Components defined by the document browser and creativity support modules. The task layer represents the place for application-specific definitions. .... 93

Figure 5-5. Components of the core layer. They provide a shared-object space, constraints and enable the handling of custom modules and services. On top of this, core support for visual presentation and event-based interaction is realized. .... 94

Figure 5-6. BEACH clients running on different roomware component are synchronised by a server. Mobile components communicate with the server via a wireless network. .... 95

Figure 5-7. Overview of the main hooks and abstractions defined by the BEACH framework. The task layer lists the applications discussed in this thesis. The numbers refer to the section in this thesis where this topic is discussed. .... 100

Figure 6-1. Different kinds of services ..... 103

Figure 6-2. Classes Module and Service. .... 104

Figure 6-3. Example how to use the extension mechanism to add a `toolbars` method to the `Module` metaclass. .... 105

Figure 6-4. Relationship between BEACH and COAST. The COAST framework is used to implement the shared-object space and the constraint system. .... 105

Figure 6-5. The core level support for the interaction model defines classes for event handling. .... 110

Figure 6-6. Event dispatching strategy for window events. Window events are always dispatched to the window controller. .... 111

Figure 6-7. The base classes defined at the core level for the interaction model in `BEACHcore.views`. Class `CoastAutomaticVisualPart` uses COAST's dependency mechanism for redraw and it supports the definition of custom computed slots. .... 112

Figure 6-8. Base classes for all shared models. The graphical notation introduced for the BEACH model (see fig. 4-2) is used in class diagrams to ease the mapping of classes to the model's dimensions. The grayscale of the background indicates the basic concern the class belongs to, the grayscale of the border the degree of coupling, and the border style the level of abstraction. .... 113

Figure 6-9. BEACH Data Model. The BEACH data model defines a containment hierarchy and the ability to add wrappers for all document elements. .... 114

Figure 6-10. An arbitrary number of document *wrapper objects* can be added around a document element. Additionally, a *relation wrapper* can be inserted between a container and its sub-components to describe the properties of their relationship. ... 115

Figure 6-11. Classes `AppModel`, `DataApp`, and `DataModel`. .... 116

Figure 6-12. The environment model of BEACH defines classes for the stations and devices it can attach. The device service handles the communication with the physical device. .... 117

Figure 6-13. BEACH User Interface Model. The dotted lines, in this case, denote possible relationships in subclasses. Class `UIApplication` typically uses information provided by the environment model to configure the user interface, while tools encapsulate the interface to application behavior, i.e. the application model. .... 118

Figure 6-14. Relationships between interaction application, user interface application, window, and views ..... 120

Figure 6-15. Example view hierarchy with attached controllers.....	121
Figure 6-16. Class <code>EventDispatcher</code> is extended with a dispatching strategy for mouse events using the view hierarchy.....	121
Figure 6-17. Predefined tracker classes. They handle interaction sequences for moving, resizing, and scaling objects.....	122
Figure 6-18. Transformation class hierarchy.....	122
Figure 6-19. Transformations and views.....	123
Figure 6-20. Transformations and views example—part 1. (a) The display in this example has a different rotation and resolution than the display area. (b) The display view is wrapped with a scale and a rotation wrapper, to be able to adjust the output properties of the display relative to its display area. The global transformation accumulates all previous transformation to speedup redraw.....	124
Figure 6-21. Transformations and views example—part 2: The translation wrapper between the display and display area view is used to select the part of the display area that corresponds to the display. The global transformation merges the overlay's and display's rotation.....	125
Figure 6-22. This extends the perspective of figures 6-20 and 6-21 to also show the graphics context and transformation graphics contexts that are used.....	126
Figure 7-1. Class diagram showing relationships between roomware components, stations, displays, and display areas.....	130
Figure 7-2. The representation for a DynaWall consisting of three computers that are combined to form a homogeneous display area. This allows the complete area to be used to show one large workspace.....	131
Figure 7-3. Atomic document elements that are defined by BEACH's generic document model.....	132
Figure 7-4. Container document elements that are defined by BEACH's generic document model.....	133
Figure 7-5. The position wrappers is the only relation wrapper that is defined by BEACH's generic document model.....	133
Figure 7-6. Atomic Document Application Models. The application model for text elements allows several input devices to provide input concurrently.....	134
Figure 7-7. Wrapper Document Application Models. The application model for the position wrapper handles the throwing of document element across large display areas.....	135
Figure 7-8. The display user interface class <code>DisplayUIApp</code> is instantiated by every station to refer to its local display. This is used as the basis for opening a window and views....	136
Figure 7-9. The document browser is a tool allowing navigation between different workspaces. Its history stores the visited application models with the associated data model (instead of data models only) in order to be able to restore editing state.	136
Figure 7-10. Overlays can be used to provide space for tools, such as document browsers or toolbars.....	137
Figure 7-11. Overlays and segments are the two concrete visual interaction areas defined by the BEACH Generic Collaboration framework.....	138
Figure 7-12. Object diagram, showing instances of all models (except the interaction model). The display belongs to a display area, which shows a segment containing a document browser pointing at a workspace. This models a part of the scene shown in the screenshot in figure 7-10.....	138

Figure 7-13. Example showing a toolbar within an overlay. (a) A screenshot showing the root and modules toolbars, and a text object and a mind-map that have been created. (b) The objects used to model this part of the user interface..... 139

Figure 7-14. The view subsystem of module `BEACHroomwareInterface` defines the basis to open a window that shows the local display's part of the display area. .... 140

Figure 7-15. Object diagram for display interaction and user interface applications (instantiation of figure 7-14). The display *interaction* application opens a view for the display specified by its associated display *user interface* application..... 141

Figure 7-16. Example view hierarchy (continued from figure 7-15): display to workspace. Interaction models are added to a part of figure 7-12. The dependencies between associations in the view hierarchy and corresponding models are highlighted. Dotted lines denote omitted wrappers (for simplification)..... 142

Figure 7-17: Assigning the position within a display area to the local display view. In addition to figure 7-16, the three transformation wrappers shown in this diagram adapt their sub-views' local coordinate systems. (Unlike figure 6-20, the display layouter is shown here instead of the transformation objects.) ..... 143

Figure 7-18. Class `EventDispatcher` is extended with a dispatch strategy for focus-oriented input devices, such as a keyboard..... 144

Figure 7-19: Intermediate steps while a user is drawing a box shape. If a shape is recognized the color changes to red (a, c, e)..... 145

Figure 7-20. Design of the gesture recognition module ..... 145

Figure 7-21: The recognizer receives mouse events from the tracker and incrementally updates the features and the list of matching shapes..... 146

Figure 7-22: The features are updated incrementally while the stroke is drawn, if shapes cannot match anymore, they are removed from the recognizer. .... 146

Figure 7-23. Dispatching strategy for gesture events..... 147

Figure 7-24. The multi-device event-dispatcher can manage several concurrent trackers. As the tracker receives events from the dispatcher, it can be reused directly and does not need to be extended..... 148

Figure 7-25. The `CommChair` splits its display area into two segments. One connects to the document browser shown at the `DynaWall`; the other is used to show a private workspace..... 150

Figure 7-26. Two users working at an `InteracTable` with the same workspace. By using separate browsers with separate application models, each user can look at the workspace with the preferred orientation. (The figure is simplified omitting the wrappers.) ..... 151

Figure 8-1. Passage: a key-chain as a passenger object on the bridge of the `InteracTable`. The interface area in the front of the display represents the virtual part of the bridge..... 156

Figure 8-2. Passage architecture overview. A new interaction model is defined to allow physical objects act as part of the user interface. Sensor-related functionality is placed in separate modules to allow reuse of these components. .... 158

Figure 8-3. The sensor interaction model supports physical objects and context information as part of the user interface. Classes added for the Passage implementation are marked by a diagonal background pattern. .... 159

Figure 8-4. The scale sensor handles the communication with an attached digital scale. It represents the currently sensed weight as part of its object state. .... 159

Figure 8-5. The tag sensor provides an interface to an RFID tag reader. The currently sensed tag ID is represented as part of its object state. The tag is a model of the tags that can be sensed..... 160

Figure 8-6. Design of the Passage module. The bridge observer observes an attached sensor. The passage registry manages all passenger objects that are in use. The bridge shows information attached to a detected passenger object..... 161

Figure 8-7. Passage interaction diagram. The bridge observer informs the bridge if a different passenger object is detected. The view of the workspace currently associated with the detected passenger is updated automatically whenever a different passenger is detected or a different workspace is attached to the passenger. 162

Figure 8-8. (a) Three working-modes of the ConnecTable: connect, stand-up, sitting. (b) Two connected ConnecTables..... 163

Figure 8-9. Sensor technology integrated in the ConnecTable. Coil and tag are placed left and right at the top of the display to detect other tables..... 164

Figure 8-10. Design of the connection module. Connection clients watch for other ConnecTables detected by the sensors. The connection monitor combines two adjacent ConnecTables to form a logically combined roomware component. .... 165

Figure 8-11. Sensors attached to a station. The ConnectionMonitor handles the connection if a sensor recognizes another ConnecTable. Connection client and monitor communicate indirectly by modifying the same station object..... 166

Figure 8-12. To switch between individual work (a) and tightly-coupled work, two ConnecTables are placed next to each other (b). This results in a homogeneous display area enabling the exchange of information objects (c). Moreover, users can have their own but shared views of the same information object for tight collaboration (d). .... 167

Figure 8-13. Two connected ConnecTables. Both displays are temporarily assigned to the common display area (`displayArea3`) of the composite roomware component. .... 167

Figure 8-14. Combining the contents of two individual workspaces to a larger common one. To keep the contents at the same physical position they have to be rearranged with respect to the coordinate system of the common workspace. .... 168

Figure 8-15. Audio hardware setup. The audio PC is connected to the DynaWall via MIDI to ensure a real-time transmission of audio information. .... 171

Figure 8-16. The audio software architecture is structured according to the BEACH conceptual model. The audio model defines an abstraction for audio output. MIDI support is implemented as a concrete output device. Module document audio provides acoustic awareness about activity in the document application model..... 172

Figure 8-17. Interaction model for audio output. The interaction application is extended to handle a hierarchy of audio presenter objects. .... 173

Figure 8-18. MIDI output is a concrete implementation of the abstract audio-output device and channel. .... 173

Figure 8-19. The audio presenter for position wrappers monitors movements happening in the position-wrapper application model..... 174

Figure 8-20. An interaction model for acoustic output can be added without interfering with the existing interaction modalities. The audio interaction application uses the same user interface application to be able to augment the visual presentation. .... 175

Figure 8-21. The BEACH architecture enables horizontal (a) and vertical (b) cuts through the architecture. This is essential to ensure reusability and extensibility. .... 178

Figure 9-1. The animated collapsing of a magnetic card cluster..... 181



Figure 9-2. Classes defined by the MagNets module..... 182

Figure 9-3. The MagNets toolbar is plugged into the modules toolbar. It can be used to create MagNets cards during creativity sessions. *Element cards* are used to write down ideas. *Title cards* are magnetic attracting and can be used to cluster cards into categories..... 183

Figure 9-4. Screenshots of PalmBeach’s Relation View (left) and Detail View (right). ..... 185

Figure 9-5. A PDA running PalmBeach pointing at an electronic whiteboard to transfer a card..... 185

Figure 9-6. Software design of PalmBeach. Due to the constraints of the PDA, data and application models, and user-interface and interaction models are combined. .... 186

Figure 9-7. Hardware setup for data exchange between Palm and DynaWall ..... 187

Figure 9-8. Software architecture to handle the data transmission between PalmBeach and roomware components ..... 187

Figure 10-1. Key contributions of the BEACH conceptual model are: the separation of user interface and interaction concerns (orthogonal to the level of abstraction), and an extended view on the concept of sharing..... 193

Figure A-2. Associations between classes..... 207

Figure A-3. Cardinality of associations. (a) The star marks an arbitrary number of associated instances, which can be zero. (b) A range can be used to specify the cardinality. .... 207

Figure A-4. To distinguish the extensions made by inheritance from extensions made to the same class in another module, the later case is marked with “<<extends>>”.... 208

Figure A-5. (a) The extension mechanism is also used to extend a concept into a model representing another concern. (b) To simplify class and object diagrams a shorthand notation is used. .... 208

Figure A-6. When the dependency mechanism is used to compute associations, the dependencies to other associations can be visualized..... 209



## Tables

Table 2-1. Summary of requirements for the software infrastructure of ubiquitous computing environments.....	32
Table 3-1. Comparison of the HCI models against the requirements for a conceptual model for the software infrastructure for roomware environments. Requirements H-1 and H-2 are supported by most HCI models and frameworks. Requirements U-2, UH-2, C-1, C-2, C-3, and UC-1 are not explicitly addressed. This is indicated in the table by the background color of the columns. ....	43
Table 3-2. Comparison of the UbiComp models and infrastructures against the requirements for a conceptual model for the software infrastructure for roomware environments. Requirements U-1, U-2, and U-4 are supported by most UbiComp systems. Requirements C-1, C-2, C-3, and UC-1 are not explicitly addressed. This is indicated in the table by the background color of the columns.....	48
Table 3-3. Comparison of the UbiComp models and infrastructures against the requirements for a conceptual model for the software infrastructure for roomware environments. Requirements C-1 and C-2 are supported by most CSCW systems; requirement C-3 is supported by SDG systems. Requirements U-2, U-3, and UH-3 are not explicitly addressed. This is indicated in the table by the background color of the columns.....	55
Table 10-1. Comparison of contributions against requirements. It shows that all requirements are fulfilled and that all aspects of the BEACH model are necessary. ....	199



## Examples

Example 2-1: Context .....	24
Example 2-2: Interactive table.....	26
Example 3-1: Apertos.....	37
Example 4-1: Data model.....	64
Example 4-2: Application model .....	64
Example 4-3: User interface model.....	66
Example 4-4: Environment model .....	67
Example 4-5: Interaction model .....	68
Example 4-6: Sharing the data model.....	70
Example 4-7: Sharing the application model .....	71
Example 4-8: Sharing the user interface model.....	72
Example 4-9: Sharing the environment model—ConnecTables .....	73
Example 4-10: Sharing the interaction model.....	74
Example 6-1: Toolbar hook .....	104
Example 6-2: Event dispatching strategy.....	111
Example 6-3: Transformations.....	123
Example 6-4: Display area.....	125
Example 7-1: Workspace browser.....	138
Example 7-2: Modules toolbar .....	139
Example 7-3: Display application and view .....	141
Example 7-4: View dependencies .....	142
Example 7-5: Immediate feedback.....	144
Example 7-6: Feature re-computation.....	146
Example 7-7: Display area; view transformations.....	149
Example 7-8: Shared document browser .....	149
Example 7-9: Multiple views on one document .....	150
Example 8-1: Adding acoustic interaction .....	174

Examples

Example 9-1: Synchronous insertion into a workspace..... 189

## References

Please note that the author before his wedding in summer 2000 had published papers under his former name Peter Seitz. The bibliography database used to create this thesis is available online at <http://tandlers.de/peter/beach/> in BibTeX format.

- [Abowd, 1999] Abowd, G. D., 1999. Software Engineering Issues for Ubiquitous Computing. In: Proceedings of the 21st International Conference on Software Engineering (ICSE'99), ACM Press, New York, NY, pp. 75–84.
- [Abowd and Mynatt, 2000] Abowd, G. D. and Mynatt, E. D., 2000. Charting Past, Present, and Future Research in Ubiquitous Computing, *ACM Transactions on Computer-Human Interaction* 7 (1), 29–58.
- [Ahuja *et al.*, 1986] Ahuja, S., Carriero, N., and Gelernter, D., 1986. Linda and Friends, *Computer* 19 (8), 26–34.
- [Alencar *et al.*, 1999] Alencar, P. S. C., Cowan, D. D., Nelson, T., Fontoura, M. F., and Lucena, C. J. P., 1999. Viewpoints and Frameworks in Component-Based Software Design. In: Fayad, M. E., Schmidt, D. C., and Johnson, R. E. (eds.), *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, John Wiley & Sons, New York, NY, USA, pp. 163–165.
- [Ambient Agoras, 2001] Ambient Agoras, 2001. Ambient Agoras: Dynamic Information Clouds in a Hybrid World, Project Review Report, EU-Project IST-2000-25134.
- [Ambient Agoras, 2003] Ambient Agoras: Dynamic Information Clouds in a Hybrid World. EU-Project IST-2000-25134, <http://www.ambient-agoras.org/> (2003).
- [Anderson *et al.*, 2000] Anderson, G. E., Graham, T. N., and Wright, T. N., 2000. Dragonfly: Linking Conceptual and Implementation Architectures of Multiuser Interactive Systems. In: Proceedings of the 22st International Conference on Software Engineering (ICSE'00), ACM Press, New York, NY, pp. 252–261.
- [Arvo and Novins, 2000] Arvo, J. and Novins, K., 2000. Fluid Sketches: Continuous Recognition and Morphing of Simple Hand-Drawn Shapes. In: Proceedings of the 13th annual ACM symposium on User interface software and technology (UIST'00), ACM Press, New York, NY, pp. 73–80. <http://www.cs.caltech.edu/~arvo/papers.html>.
- [Ayatsuka *et al.*, 2000] Ayatsuka, Y., Matsushita, N., and Rekimoto, J., 2000. Hyper Palette: a Hybrid Computing Environment for Small Computing Devices. In: CHI 2000 Extended Abstracts, ACM Press, New York, NY, pp. 133–134. <http://www.csl.sony.co.jp/person/aya/HyPa/>.
- [Banavar and Bernstein, 2002] Banavar, G. and Bernstein, A., 2002. Software infrastructure and design challenges for ubiquitous computing applications, *Communications of the ACM* 45 (12), 92–96. <http://doi.acm.org/10.1145/585597.585622>.
- [Banavar *et al.*, 2000] Banavar, G., Beck, J., Gluzberg, E., Munson, J., Sussman, J., and Zukowski, D., 2000. An Application Model for Interactive Environments. In: ICSE'00 workshop on Software Engineering for Wearable and Pervasive Computing (SEWPC'00). <http://www.cs.washington.edu/sewpc/papers/banavar.pdf>.
- [Bass *et al.*, 1992] Bass, L., Faneuf, R., Little, R., Mayer, N., Pellegrino, B., Reed, S., Seacord, R., Sheppard, S., and Szczur, M., 1992. A metamodel for the runtime architecture of an interactive system: the UIMS tool developers workshop, *ACM SIGCHI Bulletin* 24 (1), 32–37. <http://doi.acm.org/10.1145/142394.142401>, <http://www.sei.cmu.edu/staff/rcs/UI-papers.html>.

## References

- [Bass *et al.*, 1999] Bass, L., Clements, P., and Kazman, R., 1999. *Software Architecture in Practice*, Addison Wesley.
- [Baudel and Beaudouin-Lafon, 1993] Baudel, T. and Beaudouin-Lafon, M., 1993. Charade: Remote Control of Objects using Free-Hand Gestures, *Communications of the ACM* 36 (7), 28–35.
- [Beaudouin-Lafon, 2000] Beaudouin-Lafon, M., 2000. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. In: *Proceedings of the CHI 2000 conference on Human factors in computing systems (CHI'00)*, ACM Press, New York, NY.
- [Beck and Johnson, 1994] Beck, K. and Johnson, R., 1994. Patterns Generate Architectures. In: *Proceedings of ECOOP'94*, vol. 821 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 139–149. <http://st-www.cs.uiuc.edu/users/patterns/patterns.html>, <http://citeseer.nj.nec.com/27318.html>.
- [Bederson and Hollan, 1994] Bederson, B. B. and Hollan, J. D., 1994. Pad++: a zooming graphical interface for exploring alternate interface physics. In: *Proceedings of the 7th annual ACM symposium on User interface software and technology*, ACM Press, pp. 17–26. <http://doi.acm.org/10.1145/192426.192435>.
- [Bederson and Meyer, 1998] Bederson, B. and Meyer, J., 1998. Implementing a Zooming user Interface: Experience Building Pad++, *Software: Practice and Experience* 28 (10), 1101–1135.
- [Bederson *et al.*, 1996] Bederson, B. B., Hollan, J. D., Perlin, K., Meyer, J., Bacon, D., and Furnas, G., 1996. Pad++: A Zoomable Graphical Sketchpad For Exploring Alternate Interface Physics, *Journal of Visual Languages and Computing* 7 (1), 3–31. <http://www.cs.umd.edu/~bederson/papers/>.
- [Bederson *et al.*, 2000] Bederson, B. B., Meyer, J., and Good, L., 2000. Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. In: *Proceedings of the 13th annual ACM symposium on User interface software and technology (UIST'00)*, ACM Press, New York, NY, pp. 171–180. <http://www.cs.umd.edu/~bederson/papers/>.
- [Bergin, 2002] Bergin, J., *Patterns for the Doctoral Student*, Pace University, <http://csis.pace.edu/~bergin/patterns/DoctoralPatterns.html> (2002).
- [Berlage and Genau, 1993] Berlage, T. and Genau, A., 1993. A Framework for Shared Applications with a Replicated Architecture. In: *Proc. of UIST'93*, ACM Press, New York, NY, pp. 249–257.
- [Bharat and Hudson, 1995] Bharat, K. A. and Hudson, S. E., 1995. Supporting distributed, concurrent, one-way constraints in user interface applications. In: *Proceedings of the 8th annual ACM symposium on User interface and software technology*, ACM Press, pp. 121–132. <http://doi.acm.org/10.1145/215585.215708>.
- [Bier and Freeman, 1991] Bier, E. A. and Freeman, S., 1991. MMM: A User Interface Architecture for Shared Editors on a Single Screen. In: *Proceedings of ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST'91)*, ACM Press, New York, NY, pp. 79–86.
- [Bolt, 1980] Bolt, R. A., 1980. "Put-That-There": Voice and Gesture at the Graphics Interface. In: *Proceedings of the 7th annual conference on Computer graphics and interactive techniques 0-89791-021-4*, ACM Press, New York, NY, pp. 262–270.
- [Bosch *et al.*, 1999] Bosch, J., Molin, P., Mattsson, M., Bengtsson, P., and Fayad, M. E., 1999. Framework Problems and Experiences. In: Fayad, M. E., Schmidt, D. C., and Johnson, R. E. (eds.), *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, vol. 1, John Wiley & Sons, New York, NY, USA.
- [Bouraqadi-Saadani *et al.*, 1998] Bouraqadi-Saadani, N. M. N., Ledoux, T., and Rivard, F., 1998. Safe Metaclass Programming. In: Freeman-Benson, B. and Chambers, C. (eds.), *Proceedings of the conference on Object-oriented programming, systems, languages, and applications (OOPSLA '98)*, ACM Press, New York, NY, pp. 84–96. <http://www.emn.fr/cs/object/tools/neoclasstalk/publications/abstractOopsla98.html>.
- [Bower and McGlashan, 2000] Bower, A. and McGlashan, B., 2000. TWISTING THE TRIAD: The evolution of the Dolphin Smalltalk MVP application framework. In: *Tutorial Paper for ESUG 2000*. <http://www.object-arts.com/>.
- [Brant *et al.*, 1998] Brant, J., Foote, B., Johnson, R. E., and Roberts, D., 1998. Wrappers to the Rescue. In: *Proceedings of the 12th European Conferences on Object-Oriented Programming (ECOOP '98)*, no. 1445 in *Lecture Notes in Computer Science*, Springer, Heidelberg, New York, p. 396ff. <http://www.laputan.org/brant/brant.html>.
- [Bruce and Elrod, 1992] Bruce, R. and Elrod, S., 1992. Liveboard: a large interactive display supporting group meetings, presentations, and remote collaboration. In: *Conference proceedings on Human factors in computing systems*, ACM Press, New York, NY, pp. 599–607. <http://doi.acm.org/10.1145/142750.143052>.
- [Bruijn and Spence, 2001] Bruijn, O. d. and Spence, R., 2001. Serendipity within a Ubiquitous Computing Environment: A Case for Opportunistic Browsing. In: Abowd, G. D., Brummitt, B., and Shafer, S. (eds.), *Proceedings of UbiComp'01: Ubiquitous Computing*, vol. 2201 of *Lecture Notes in Computer Science*, Springer, Heidelberg, New York.
- [Brummit *et al.*, 2000] Brummit, B., Meyers, B., Krumm, J., Kern, A., and Shafer, S., 2000. Easyliving: Technologies for Intelligent Environments. In: *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC'00)*, vol. 1927, no. 1927 in *Lecture Notes in Computer Science*, Springer, Heidelberg, New York, pp. 12–29.



- [Buschmann *et al.*, 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., 1996. *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons.
- [Buzan, 2002] Buzan, T., 2002. *How to Mind Map The Ultimate Thinking Tool That Will Change Your Life*, Harper Collins Publ.
- [Callahan *et al.*, 1988] Callahan, J., Hopkins, D., Weiser, M., and Shneiderman, B., 1988. An empirical comparison of pie vs. Linear Menus. In: CHI88, ACM Press, New York, NY, pp. 95–100. <http://doi.acm.org/10.1145/57167.57182>.
- [Calvary *et al.*, 1997] Calvary, G., Coutaz, J., and Nigay, L., 1997. From Single-User Architectural Design to PAC\*: a Generic Software Architecture Model for CSCW. In: *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'97)*, ACM Press, New York, NY, pp. 242–249.
- [Carnegie Mellon University, 2000] The Pittsburgh Pebbles PDA Project Page, Carnegie Mellon University, <http://www.pebbles.hcii.cmu.edu/> (2000).
- [Carnegie Mellon University, 2002] The Aura Project: Pervasive Invisible Computing—Homepage, Carnegie Mellon University, <http://www.cs.cmu.edu/~aura/> (2002).
- [Carriero *et al.*, 1994] Carriero, N. J., Gelernter, D., Mattson, T. G., and Sherman, A. H., 1994. The Linda alternative to message-passing systems, *Parallel Computing* 20 (4), 633–655.
- [Castro and Kramer, 2001] Castro, J. and Kramer, J., First International Workshop From Software Requirements to Architectures (STRAW'01) at ICSE'01, <http://www.cin.ufpe.br/~straw01/> (2001).
- [Caswell and Debaty, 2000] Caswell, D. and Debaty, P., 2000. Creating Web Representations for Places. In: *Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC'00)*, vol. 1927, no. 1927 in *Lecture Notes in Computer Science*, Springer Verlag, pp. 114–126. <http://www.cooltown.com/>.
- [Chambers, 1992] Chambers, C., 1992. Object-Oriented Multi-Methods in Cecil. In: Madsen, O. L. (ed.), *ECOOP '92, European Conference on Object-Oriented Programming*, Utrecht, The Netherlands, vol. 615 of *Lecture Notes in Computer Science*, Springer, Heidelberg, New York, pp. 33–56. <file://cs.washington.edu/pub/chambers/cecil-oo-mm.ps.Z>.
- [Chambers, 1995] Chambers, C., 1995. *The Cecil Language: Specification and Rationale, Version 2.0*, Tech. Rep., Department of Computer Science and Engineering, University of Washington. <http://www.cs.washington.edu/research/projects/cecil/www/pubs/cecil-spec.html>.
- [Chen and Kotz, 2000] Chen, G. and Kotz, D., *A Survey of Context-Aware Mobile Computing Research*. Dartmouth Collage, Dept. of Computer Science, Hannover, NH 03755, 2000. <http://www.cs.dartmouth.edu/reports/abstracts/TR2000-381/>, <http://citeseer.nj.nec.com/chen00survey.html>.
- [Cheng *et al.*, 2002] Cheng, S., Garlan, D., Schmerl, B., Sousa, J. P., Spitznagel, B., Steenkiste, P., and Hu, N., 2002. Software Architecture-based Adaptation for Pervasive Systems. In: Schmeck, H., Ungerer, T., and Wolf, L. (eds.), *International Conference on Architecture of Computing Systems (ARCS'02): Trends in Network and Pervasive Computing*, vol. 2299 of *Lecture Notes in Computer Science*, Springer, Heidelberg, New York, pp. 67–82. <http://www.cs.cmu.edu/~aura/>.
- [Cincom, 2002] VisualWorks Homepage, Cincom, <http://www.cincom.com> (2002).
- [Clifton *et al.*, 2000] Clifton, C., Leavens, G. T., Chambers, C., and Millstein, T., 2000. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In: *Proceedings of the conference on Object-oriented programming, systems, languages, and applications (OOPSLA '00)*, ACM Press, New York, NY, pp. 130–145.
- [COAST, 2000a] *COAST4 Reference Manual*. 1st edition, 2000a. <http://www.opencoast.org/documentation/COAST4-reference.PDF>.
- [COAST, 2000b] *COAST overview*. 1st edition, 2000b. <http://www.opencoast.org/documentation/COAST4-overview.PDF>.
- [COAST, 2003] OpenCOAST Homepage, <http://www.opencoast.org> (2003).
- [Coen, 1998] Coen, M. H., 1998. Design Principles for Intelligent Environments. In: *Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, no. SS-98-92, AAAI Press, pp. 547–554. <http://www.ai.mit.edu/projects/aire/papers.shtml>.
- [Coen *et al.*, 1999] Coen, M., Phillips, B., Warshawsky, N., Weisman, L., Peters, S., and Finin, P., 1999. Meeting the Computational Needs of Intelligent Environments: The Metaglu System. In: Nixon, P., Lacey, G., and Dobson, S. (eds.), *1st International Workshop on Managing Interactions in Smart Environments (MANSE'99)*, Springer-Verlag, pp. 201–212. <http://www.ai.mit.edu/projects/aire/papers.shtml>.
- [Conklin, 1987] Conklin, J., 1987. Hypertext: An Introduction and Survey, *Computer Magazine* 20 (9), 17–41.
- [Cooperstock *et al.*, 1997] Cooperstock, J. R., Fels, S. S., Buxton, W., and Smith, K. C., 1997. Reactive environments: throwing away your keyboard and mouse, *Communications of the ACM* 40 (9), 65–73. <http://doi.acm.org/10.1145/260750.260774>.
- [Coutaz, 1997] Coutaz, J., 1997. PAC-ing the Architecture of Your User Interface. In: *Proceedings of the 4th Eurographics Workshop on Design, Specication and Verication of Interactive Systems (DSV-IS'97)*, Springer, Heidelberg, New York, pp. 15–32. <http://citeseer.nj.nec.com/coutaz97pacing.html>.

## References

- [Coutaz *et al.*, 2003] Coutaz, J., Balme, L., Lachenal, C., and Barralon, N., 2003. Software Infrastructure for Distributed Migratable user Interfaces. In: Johanson, B., Borchers, J., Schiele, B., Tandler, P., and Edwards, K. (eds.), *At the Crossroads: The Interaction of HCI and Systems Issues in UbiComp, UbiComp '03 workshop*. <http://ubihcisys.stanford.edu/>.
- [Czerwinski *et al.*, 1999] Czerwinski, S. E., Zhao, B. Y., Hodes, T. D., Joseph, A. D., and Katz, R. H., 1999. An Architecture for a Secure Service Discovery Service. In: *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom'99)*, ACM Press, New York, NY, pp. 24–35. <http://citeseer.nj.nec.com/czerwinski99architecture.html>.
- [Da Campo, 2001] Da Campo, S., 2001. Respect for Privacy in Future Office Environments. A study on privacy implications of ubiquitous computing in work context by example of the Ambient Agoras environment, Diplomarbeit, Technische Universität Darmstadt, Fachbereich für Elektrotechnik und Informationstechnik.
- [Demers *et al.*, 1994] Demers, A. J., Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., and Welch, B. B., 1994. The Bayou architecture: Support for data sharing among mobile users. In: *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pp. 2–7. <http://www2.parc.com/csl/projects/bayou/>, <http://citeseer.nj.nec.com/demers94bayou.html>.
- [Demeyer, 1996] Demeyer, S., 1996. Zyper-Tailorability as a Link from Object-Oriented Software Engineering to Open Hypermedia, Ph.D. thesis, Vrije Universiteit Brussel, Departement Informatica.
- [Demeyer *et al.*, 1997] Demeyer, S., Meijler, T. D., Nierstrasz, O., and Steyaert, P., 1997. Design Guidelines for 'Tailorable' Frameworks, *Communications of the ACM* 40 (10), 60–64.
- [Dewan, 1999] Dewan, P., 1999. Architectures for Collaborative Applications. In: Beaudouin-Lafon, M. (ed.), *Computer Supported Co-operative Work*, no. 7 in *Trends in Software*, John Wiley & Sons, New York, NY, USA, ch. 7, pp. 169–193.
- [Dewan and Choudhard, 1991] Dewan, P. and Choudhard, R., 1991. Flexible user interface coupling in a collaborative system. In: *Human factors in computing systems conference proceedings on Reaching through technology*, ACM Press, pp. 41–48. <http://doi.acm.org/10.1145/108844.108851>.
- [Dewan and Choudhary, 1992] Dewan, P. and Choudhary, R., 1992. A high-level and flexible framework for implementing multiuser user interfaces, *ACM Transactions on Information Systems* 10 (4), 345–380. <http://doi.acm.org/10.1145/146486.146495>.
- [Dewan and Choudhary, 1995] Dewan, P. and Choudhary, R., 1995. Coupling the User Interfaces of a Multiuser Program, *ACM Transactions on Computer-Human Interaction* 2 (1), 1–39.
- [Dey, 2000] Dey, A. K., 2000. Providing Architectural Support for Building Context-Aware Applications, Ph.D. thesis, Georgia Institute of Technology.
- [Dey *et al.*, 2001a] Dey, A. K., Abowd, G. D., and Salber, D., 2001a. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications, *Human-Computer Interaction* 16 (2–4), 97–166. <http://www.cc.gatech.edu/fce/contexttoolkit>.
- [Dey *et al.*, 2001b] Dey, A. K., Salber, D., and Abowd, G. D., The Context Toolkit Homepage, Georgia Institute of Technology, <http://www.cc.gatech.edu/fce/contexttoolkit/> (2001b).
- [Dietz and Leigh, 2001] Dietz, P. H. and Leigh, D., 2001. DiamondTouch: A Multi-User Touch Technology. In: *Proceedings of 14th Annual ACM Symposium on User Interface and Software Technology (UIST'01)*, vol. 3, no. 2 in *CHI Letters*, ACM Press, New York, NY, pp. 219–226.
- [Dijkstra, 1968] Dijkstra, E. W., 1968. The structure of the "THE"-multiprogramming system, *Communications of the ACM* 11 (5), 341–346. <http://doi.acm.org/10.1145/363095.363143>.
- [Dourish *et al.*, 2000] Dourish, P., Edwards, W. K., Howell, J., LaMarca, A., Lamping, J., Petersen, K., Salisbury, M., Terry, D., and Thornton, J., 2000. A Programming Model for Active Documents. In: *Proc. of UIST'00*, vol. 2, no. 2 in *CHI Letters*, ACM Press, New York, NY, pp. 41–50.
- [Druin *et al.*, 1997] Druin, A., Stewart, J., Proft, D., Bederson, B., and Hollan, J., 1997. KidPad: a design collaboration between children, technologists, and educators. In: *conference proceedings on Human factors in computing systems*, ACM Press, pp. 463–470. <http://doi.acm.org/10.1145/258549.258866>.
- [Edwards and LaMarca, 1999] Edwards, W. K. and LaMarca, A., 1999. Balancing Generality and Specificity in Document Management Systems. In: *INTERACT '99*, IOS Press, pp. 187–195.
- [Edwards *et al.*, 1997] Edwards, W. K., Mynatt, E. D., Petersen, K., Spreitzer, M. J., Terry, D. B., and Theimer, M. M., 1997. Designing and implementing asynchronous collaborative applications with Bayou. In: *Proceedings of the 10th annual ACM symposium on User interface software and technology*, ACM Press, pp. 119–128. <http://doi.acm.org/10.1145/263407.263530>.
- [Ellis and Wainer, 1994] Ellis, C. and Wainer, J., 1994. A Conceptual Model of Groupware. In: *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94)*, ACM Press, New York, NY, pp. 79–88.
- [Ellis *et al.*, 1991] Ellis, C. A., Gibbs, S. J., and Rein, G., 1991. Groupware: some issues and experiences, *Communications of the ACM* 34 (1), 39–58. <http://doi.acm.org/10.1145/99977.99987>.
- [Ende, 1973] Ende, M., 1973. Momo, K. Thienemanns Verlag, Stuttgart.

- [Englich, 1999] Englich, M., 1999. Human Interfaces – Design für multimediale Arbeitsumgebungen. In: Streitz, N., Remmers, B., Pietzcker, M., and Grundmann, R. (eds.), *Arbeitswelten im Wandel—für die Zukunft?*, Deutsche Verlags-Anstalt GmbH, Stuttgart, pp. 21–35, (in German).
- [Epstein and LaLonde, 1988] Epstein, D. and LaLonde, W. R., 1988. A smalltalk window system based on constraints. In: *Conference proceedings on Object-oriented programming systems, languages and applications*, ACM Press, pp. 83–94. <http://doi.acm.org/10.1145/62083.62092>.
- [Esler *et al.*, 1999] Esler, M., Hightower, J., Anderson, T., and Borriello, G., 1999. Next century challenges: data-centric networking for invisible computing: the Portolano project at the University of Washington. In: *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, ACM Press, pp. 256–262. <http://doi.acm.org/10.1145/313451.313553>.
- [Fayad, 1999] Fayad, M. E., 1999. Application Frameworks. In: Fayad, M. E., Schmidt, D. C., and Johnson, R. E. (eds.), *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, vol. 1, John Wiley & Sons, New York, NY, USA, ch. 1, pp. 3–27.
- [Fayad and Schmidt, 1997] Fayad, M. E. and Schmidt, D. C., 1997. Object-Oriented Application Frameworks, *Communications of the ACM* 40 (10), 32–38.
- [Fayad *et al.*, 1999] Fayad, M. E., Schmidt, D. C., and Johnson, R. E. (eds.), 1999. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, vol. 1, John Wiley & Sons, New York, NY, USA.
- [Fernuni Hagen, 2002a] Mobie & Ubiquitous Computing Homepage, Fernuni Hagen, <http://dreamteam.fernuni-hagen.de/micro/micro.html> (2002a).
- [Fernuni Hagen, 2002b] DreamTeam Homepage, Fernuni Hagen, <http://dreamteam.fernuni-hagen.de/dreamteam/dreamteam.html> (2002b).
- [Fitzmaurice *et al.*, 1995] Fitzmaurice, G. W., Ishii, H., and Buxton, W. A. S., 1995. Bricks: laying the foundations for graspable user interfaces. In: *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'95)*, ACM Press/Addison-Wesley Publishing Co., pp. 442–449.
- [Flucher, 2001] Flucher, S., 2001. *Unterstützung kontextbewusster Datenverarbeitung*, Diplomarbeit, Fachbereich Informatik, TU Darmstadt, Germany, (in German).
- [FOD, 2002] Future Office Dynamics Research Consortium Homepage, <http://www.future-office.de> (2002).
- [Foegen and Battenfeld, 2001] Foegen, M. and Battenfeld, J., 2001. Die Rolle der Architektur in der Anwendungsentwicklung, *Informatik Spektrum* 24 (5), 290–301, (in German). [http://www.wibas.de/presentation/pages/de\\_whp\\_26.html](http://www.wibas.de/presentation/pages/de_whp_26.html).
- [Foote and Johnson, 1989] Foote, B. and Johnson, R. E., 1989. Reflective facilities in Smalltalk-80. In: *Conference proceedings on Object-oriented programming systems, languages and applications*, ACM Press, pp. 327–335. <http://doi.acm.org/10.1145/74877.74911>.
- [Fox *et al.*, 2000] Fox, A., Johanson, B., Hanrahan, P., and Winograd, T., 2000. Integrating Information Appliances into an Interactive Workspace, *IEEE CG&A* 20 (3), 54–65. <http://graphics.stanford.edu/projects/iwork/papers/ieee-pda00/>.
- [Freeman-Benson, 1990] Freeman-Benson, B. N., 1990. Kaleidoscope: mixing objects, constraints, and imperative programming. In: *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, ACM Press, pp. 77–88. <http://doi.acm.org/10.1145/97945.97957>.
- [Froehlich *et al.*, 1997] Froehlich, G., Hoover, H. J., Liu, L., and Sorenson, P., 1997. Hooking into Object-Oriented Application Frameworks. In: *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, IEEE CS Press, pp. 491–501. <http://citeseer.nj.nec.com/froehlich97hooking.html>.
- [Froehlich *et al.*, 1999] Froehlich, G., Hoover, H. J., Liu, L., and Sorenson, P., 1999. Reusing Hooks. In: Fayad, M. E., Schmidt, D. C., and Johnson, R. E. (eds.), *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, vol. 1, John Wiley & Sons, New York, NY, USA, ch. 9, pp. 219–236.
- [Furnas and Bederson, 1995] Furnas, G. W. and Bederson, B. B., 1995. Space-Scale Diagrams: Understanding Multiscale Interfaces. In: *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'95)*, ACM Press, New York, NY, pp. 234–241.
- [Gamma *et al.*, 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley.
- [Garlan, 2000] Garlan, D., 2000. Software Architecture: a Roadmap. In: *Proceedings of the 22st International Conference on Software Engineering (ICSE'00)*, ACM Press, New York, NY, pp. 93–101.
- [Garlan, 2001] Garlan, D., 2001. Software Architecture. In: Marciniak, J. (ed.), *Encyclopedia of Software Engineering*, John Wiley & Sons, New York, NY, USA. <http://www-2.cs.cmu.edu/~able/publications/encycSE2001/>.
- [Garlan and Schmerl, 2001] Garlan, D. and Schmerl, B., 2001. Component-Based Software Engineering in Pervasive Computing Environments. In: *The 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction*. <http://www.cs.cmu.edu/~aura/>.

## References

- [Garlan and Shaw, 1993] Garlan, D. and Shaw, M., 1993. An Introduction to Software Architecture. In: Ambriola, V. and Tortora, G. (eds.), *Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, Singapore, pp. 1–39. [http://www-2.cs.cmu.edu/~able/paper\\_abstracts/intro\\_softarch.html](http://www-2.cs.cmu.edu/~able/paper_abstracts/intro_softarch.html).
- [Garlan *et al.*, 1995] Garlan, D., Allen, R., and Ockerbloom, J., 1995. Architectural Mismatch, or, Why it's hard to build systems out of existing parts. In: *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*, ACM Press, New York, NY, pp. 179–185. <http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/able/www/publications/>.
- [Geißler, 1995] Geißler, J., 1995. Gedrics: The next generation of icons. In: *Proceedings of the 5th International Conference on Human-Computer Interaction (INTERACT'95)*, pp. 73–78.
- [Geißler, 1998] Geißler, J., 1998. Shuffle, Throw or Take It!: Working Efficiently with an Interactive Wall. In: *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'98)*, Late-Breaking Results: "The Real and the Virtual: Integrating Architectural and Information Spaces (Suite)", ACM Press, New York, NY, pp. 265–266.
- [Geißler, 2001] Geißler, J., 2001. *Design und Implementierung einer stiftzentrierten Benutzungsoberfläche*, Ph.D. thesis, Technische Universität Darmstadt, (in German).
- [Goldberg and Robson, 1989] Goldberg, A. and Robson, D., 1989. *Smalltalk-80—The Language and its Implementation*, Addison-Wesley, 2. edition.
- [Graham and Grundy, 1999] Graham, T. N. and Grundy, J., 1999. External Requirements of Groupware Development Tools. In: *Proceedings of the 5th Conference on Engineering for Human-Computer Interaction (EHCI '98)*, Kluwer Academic Publishers, Amsterdam, NL, pp. 363–376. <http://stl.cs.queensu.ca/~graham/stl/pubs/ehci98.html>.
- [Graham *et al.*, 1996] Graham, T. N., Urnes, T., and Nejabi, R., 1996. Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware. In: *UIST'96*, ACM Press, New York, NY, pp. 1–10.
- [Greenberg, 1991] Greenberg, S., 1991. Personalizable groupware: Accomodating individual roles and group differences. In: *Proceedings of the ECSCW '91 European Conference of Computer Supported Cooperative Work*, Kluwer Academic Publishers, Amsterdam, NL, pp. 17–32. <http://www.cpsc.ucalgary.ca/grouplab/papers/1991/91.PersGw.ECSCW/abstract.html>.
- [Greenberg and Boyle, 2002] Greenberg, S. and Boyle, M., 2002. Customizable physical interfaces for interacting with conventional applications. In: *Proceedings of the 15th annual ACM symposium on User interface software and technology*, ACM Press, pp. 31–40. <http://doi.acm.org/10.1145/571985.571991>.
- [Greenberg and Fitchett, 2001] Greenberg, S. and Fitchett, C., 2001. Phidgets: Easy Development of Physical Interfaces through Physical Widgets. In: *Proceedings of 14th Annual ACM Symposium on User Interface and Software Technology (UIST'01)*, vol. 3, no. 2 in *CHI Letters*, ACM Press, New York, NY.
- [Greenberg and Roseman, 1999] Greenberg, S. and Roseman, M., 1999. Groupware Toolkits for Synchronous Work. In: Beaudouin-Lafon, M. (ed.), *Computer Supported Co-operative Work*, no. 7 in *Trends in Software*, John Wiley & Sons, New York, NY, USA, ch. 6, pp. 135–168. <http://grouplab.cpsc.ucalgary.ca/papers/1999/99-GroupwareToolkits.Wiley/abstract.html>.
- [Greenberg and Roseman, 2003] Greenberg, S. and Roseman, M., 2003. Using a Room Metaphor to Ease Transitions in Groupware. In: Ackerman, M., Pipek, V., and Wulf, V. (eds.), *Sharing Expertise: Beyond Knowledge Management*, Cambridge, MA, MIT Press, pp. 203–256. <http://www.cpsc.ucalgary.ca/grouplab/papers/>.
- [Greenberg *et al.*, 1999] Greenberg, S., Boyle, M., and LaBerge, J., 1999. PDAs and Shared Public Devices: Making Personal Information Public, and Public Information Personal, *Personal Technologies* 3 (1), 54–64.
- [Gregory D. Abowd *et al.*, 1993] Gregory D. Abowd, Robert Allen, and David Garlan, 1993. Using Style to Understand Descriptions of Software Architecture, *ACM Software Engineering Notes* 18 (5), 9–20. <http://www.cc.gatech.edu/fac/Gregory.Abowd/html/architecture-formal-pubs.html>.
- [Grimm *et al.*, 2001] Grimm, R., Davis, J., Lemar, E., MacBeth, A., Swanson, S., Gribble, S., Anderson, T., Bershada, B., Borriello, G., and Wetherall, D., 2001. System-Level Programming Abstractions for Ubiquitous Computing. In: *UbiTools'01—Workshop on Application Models and Programming Tools for Ubiquitous Computing* (held in conjunction with the *UbiComp'01*). <http://choices.cs.uiuc.edu/UbiTools01/>.
- [Grimm *et al.*, 2002] Grimm, R., Davis, J., Lemar, E., MacBeth, A., Swanson, S., Anderson, T., Bershada, B., Borriello, G., Gribble, S., and Wetherall, D., 2002. Programming for pervasive computing environments. <http://one.cs.washington.edu>.
- [Guimbretière and Winograd, 2000] Guimbretière, F. and Winograd, T., 2000. FlowMenu: combining command, text, and data entry. In: *Proceedings of the 13th annual ACM symposium on User interface software and technology*, ACM Press, New York, NY, pp. 213–216. <http://doi.acm.org/10.1145/354401.354778>.
- [Guimbretière *et al.*, 2001] Guimbretière, F., Stone, M., and Winograd, T., 2001. Fluid Interaction with High-resolution Wall-Size Displays. In: *Proceedings of the 14th annual ACM symposium on User interface software and technology*, ACM Press, pp. 21–30. <http://doi.acm.org/10.1145/502348.502353>.

- [Gutwin and Greenberg, 1998] Gutwin, C. and Greenberg, S., 1998. Design for Individuals, Design for Groups: Tradeoffs between power and workspace awareness. In: Proceedings of the ACM 1998 conference on Computer supported cooperative work, ACM Press, pp. 207–216. <http://doi.acm.org/10.1145/289444.289495>.
- [Gutwin *et al.*, 1996] Gutwin, C., Roseman, M., and Greenberg, S., 1996. A Usability Study of Awareness Widgets in a Shared Workspace Groupware System. In: Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96), ACM Press, New York, NY, pp. 258–267.
- [Haake and Wilson, 1992] Haake, J. M. and Wilson, B., 1992. Supporting Collaborative Writing of Hyperdocuments in SEPIA. In: Proceedings of the conference on Computer-supported cooperative work, ACM Press, pp. 138–146. <http://doi.acm.org/10.1145/143457.143472>.
- [Haake *et al.*, 1994] Haake, J. M., Neuwirth, C. M., and Streitz, N. A., 1994. Coexistence and Transformation of Informal and Formal Structures: Requirements for More Flexible Hypermedia Systems. In: ACM European Conference on Hypermedia Technology (ECHT'94), ACM Press, New York, NY, pp. 1–12.
- [Halasz and Schwartz, 1994] Halasz, F. and Schwartz, M., 1994. The Dexter hypertext reference model, Communications of the ACM 37 (2), 30–39. <http://doi.acm.org/10.1145/175235.175237>.
- [Hancock *et al.*, 2002] Hancock, M. S., Mandryk, R. L., Inkpen, K. M., and Booth, K. S., 2002. Adapting to a User's Location on an Interactive Tabletop, submitted to UIST'02.
- [Henry *et al.*, 1990] Henry, T. R., Hudson, S. E., and Newell, G. L., 1990. Integrating Gesture and Snapping into a User Interface Toolkit. In: Proceedings of the the third annual ACM SIGGRAPH symposium on User interface software and technology, ACM Press, pp. 112–122. <http://doi.acm.org/10.1145/97924.97938>.
- [Henry *et al.*, 1991] Henry, T. R., Hudson, S. E., Yeatts, A. K., Myers, B. A., and Feiner, S., 1991. A Nose Gesture Interface Device: Extending Virtual Realities. In: Proceedings of the 4th annual ACM symposium on User interface software and technology (UIST'91), ACM Press, New York, NY, pp. 65–68.
- [Herrmann and Mezini, 2000] Herrmann, S. and Mezini, M., 2000. PIROL: A Case Study for Multidimensional Separation of Concerns in Software Engineering Environments. In: OOPSLA00, vol. 35, no. 10 in SIGPLAN Notices, ACM Press, New York, NY. <http://www.st.informatik.tu-darmstadt.de>.
- [Hess *et al.*, 2001a] Hess, C. K., Ballesteros, F., Campbell, R. H., and Mickunas, M. D., 2001a. An Adaptive Data Object Service for Pervasive Computing Environments. In: Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01), pp. 31–45.
- [Hess *et al.*, 2001b] Hess, C. K., Ballesteros, F., Campbell, R., and Mickunas, M. D., 2001b. An Adaptable Data Object Service for Pervasive Computing Environments. In: Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001), pp. 31–45. <http://choices.cs.uiuc.edu/gaia/html/publications.htm>.
- [Hill, 1993] Hill, R. D., 1993. The Rendezvous constraint maintenance system. In: Proceedings of the sixth annual ACM symposium on User interface software and technology, ACM Press, pp. 225–234. <http://doi.acm.org/10.1145/168642.168665>.
- [Hill *et al.*, 1994] Hill, R. D., Brinck, T., Rohall, S. L., Patterson, J. F., and Wilner, W., 1994. The Rendezvous architecture and language for constructing multiuser applications, ACM Transactions on Computer-Human Interaction (ToCHI) 1 (2), 81–125. <http://doi.acm.org/10.1145/180171.180172>.
- [Hinckley, 2003] Hinckley, K., 2003. Synchronous Gestures for Multiple Persons and Computers. In: Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST'03), ACM Press, New York, NY, pp. 149–158. <http://research.microsoft.com/users/kenh/>.
- [Hong and Landay, 2000] Hong, J. I. and Landay, J. A., 2000. SATIN: A Toolkit for Informal Ink-based Applications. In: Proceedings of the 13th annual ACM symposium on User interface software and technology (UIST'00), vol. 2, no. 2 in CHI Letters, ACM Press, pp. 63–72. <http://guir.berkeley.edu/projects/satin>.
- [Hong and Landay, 2001] Hong, J. I. and Landay, J. A., 2001. An Infrastructure Approach to Context-Aware Computing, Human-Computer Interaction 16 (2–4), 287–303.
- [Hopkins, 2000] Hopkins, J., 2000. Component Primer, Communications of the ACM 43 (10).
- [Hourcade and Bederson, 1999] Hourcade, J. P. and Bederson, B. B., 1999. Architecture and Implementation of a Java Package for Multiple Input Devices (MID), Tech. Rep. HCIL-99-08, CS-TR-4018, UMIACS-TR-99-26, Computer Science Department, University of Maryland, College Park, MD. <http://www.cs.umd.edu/hcil>.
- [HP Labs, 2003] Cooltown Homepage, HP Labs, <http://www.cooltown.com/> (2003).
- [Hudson and Smith, 1996] Hudson, S. E. and Smith, I., 1996. Ultra-lightweight constraints. In: Proceedings of the 9th annual ACM symposium on User interface software and technology, ACM Press, pp. 147–155. <http://doi.acm.org/10.1145/237091.237112>, [http://www.cc.gatech.edu/gvu/ui/sub\\_arctic/](http://www.cc.gatech.edu/gvu/ui/sub_arctic/).
- [Ishii and Ullmer, 1997] Ishii, H. and Ullmer, B., 1997. Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. In: Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'97), ACM Press, New York, NY, pp. 234–241.
- [Jacobsen, 2000] Jacobsen, E. E., 2000. Concepts and Language Mechanisms in Software Modelling, Ph.D. thesis, Faculty of Science and Engineering, University of Southern Denmark. <http://www.mip.sdu.dk/sweat>.

## References

- [Jacobson *et al.*, 1992] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G., 1992. Object-Oriented Software Engineering, a Use Case Driven Approach, ACM Press, Addison Wesley.
- [Johanson and Fox, 2002] Johanson, B. and Fox, A., 2002. The Event Heap: A Coordination Infrastructure for Interactive Workspaces. In: Proc. of the 4th IEEE Workshop on Mobile Computer Systems and Applications (WMCSA-2002), IEEE Press. <http://graphics.stanford.edu/papers/eheap3/>.
- [Johanson and Fox, 2004] Johanson, B. and Fox, A., 2004. Extending Tuplespaces for Coordination in Interactive Workspaces, Journal of Systems & Software (JSS) Special issue on Ubiquitous Computing 69 (3), 243–266. <http://graphics.stanford.edu/papers/eheap-jss/>, <http://www.sciencedirect.com/science/article/B6V0N-49R5K3S-4/2/c6403c2b4f49b86148e15769d39d959f>.
- [Johanson *et al.*, 2002a] Johanson, B., Fox, A., and Winograd, T., 2002a. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms, IEEE Pervasive Computing, special issue on "Integrated Pervasive Computing Environments" 1 (2), 67–74. <http://graphics.stanford.edu/papers/iwork-overview/>.
- [Johanson *et al.*, 2002b] Johanson, B., Hutchins, G., Winograd, T., and Stone, M., 2002b. PointRight: Experience with Flexible Input Redirection in Interactive Workspaces. In: Proceedings of the 15th annual ACM symposium on User interface software and technology (UIST'02), vol. 4, no. 2 in CHI Letters, ACM Press, New York, NY, pp. 227–234. <http://graphics.stanford.edu/papers/pointright-uis2002/>.
- [Johnson, 1997] Johnson, R. E., 1997. Frameworks = (Components + Patterns). How frameworks compare to other object-oriented reuse techniques, Communications of the ACM 40 (10), 39–42.
- [Johnson and Foote, 1988] Johnson, R. E. and Foote, B., 1988. Designing Reusable Classes, Journal of Object-Oriented Programming 1 (2), 22–35. <http://www.laputan.org/drc/drc.html>.
- [Johnson-Lenz and Johnson-Lenz, 1994] Johnson-Lenz, P. and Johnson-Lenz, T., 1994. Groupware: Coining and Defining It, Tech. Rep., Awakening Technology. <http://www.awakentech.com>.
- [Kandogan and Shneiderman, 1997] Kandogan, E. and Shneiderman, B., 1997. Elastic Windows: A Hierarchical Multi-Window World-Wide Web Browser. In: Proceedings of the 10th annual ACM symposium on User interface software and technology (UIST'97), ACM Press, New York, NY, pp. 169–177.
- [Kazman and Bass, 1995] Kazman, R. and Bass, L., 1995. Software Architectures for Human-Computer Interaction: Analysis and Construction. <http://www.cgl.uwaterloo.ca/~rnkazman/SA-bib.html>, <http://citeseer.ist.psu.edu/36927.html>.
- [Keene, 1989] Keene, S. E., 1989. Object-oriented Programming in Common Lisp, Addison-Wesley.
- [Kiczales, 1994] Kiczales, G., 1994. Why are Black Boxes so Hard to Reuse?—Towards a New Model of Abstraction in the Engineering of Software. In: Invited talk at OOPSLA'94 and ICSE-17. <http://www2.parc.com/csl/groups/sda/projects/oi/towards-talk/transcript.html>.
- [Kiczales *et al.*, 1991] Kiczales, G., Rivières, J. d., and Bobrow, D. G., 1991. The Art of the Metaobject Protocol, MIT Press.
- [Kiczales *et al.*, 1997a] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J., and Irwin, J., 1997a. Aspect-Oriented Programming. In: Aksit, M. and Matsuoka, S. (eds.), Proceedings of ECOOP '97—Object-Oriented Programming 11th European Conference, vol. 1241 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 220–242. <http://www.cs.ubc.ca/~gregor/kiczales-ECOOP1997-AOP.pdf>.
- [Kiczales *et al.*, 1997b] Kiczales, G., Lamping, J., Lopes, C. V., Maeda, C., Mendhekar, A., and Murphy, G., 1997b. Open Implementation Design Guidelines. In: Proceedings of the 19th International Conference on Software Engineering (ICSE'97), ACM Press, New York, NY, pp. 481–490. <http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-ICSE97/for-web.pdf>.
- [Kindberg *et al.*, 2000] Kindberg, T. *et al.*, 2000. People, Places, Things: Web Presence for the Real World. In: Proceedings of the 3rd Annual Wireless and Mobile Computer Systems and Applications, p. 19. <http://www.cooltown.com/>, <http://www.hpl.hp.com/techreports/2000/HPL-2000-16.html>.
- [Klas *et al.*, 1989] Klas, W., Neuhold, E. J., and Schrefl, M., 1989. Tailoring Object-Oriented Data Models through Metaclasses, Arbeitspapiere der GMD 404, Gesellschaft für Mathematik und Datenverarbeitung mbH (GMD).
- [Klemmer *et al.*, 2001] Klemmer, S., Newman, M. W., Farrell, R., Bilezikian, M., and Landay, J. A., 2001. The Designers' Outpost: A Tangible Interface for Collaborative Web Site Design. In: Proceedings of 14th Annual ACM Symposium on User Interface and Software Technology (UIST'01), vol. 3, no. 2 in CHI Letters, ACM Press, New York, NY, pp. 1–10.
- [Konomi *et al.*, 1999] Konomi, S., Müller-Tomfelde, C., and Streitz, N. A., 1999. Passage: Physical Transportation of Digital Information in Cooperative Buildings. In: Cooperative Buildings – Integrating Information, Organizations, and Architecture. Proceedings of the Second International Workshop on Cooperative Buildings (CoBuild'99), vol. 1670 of *Lecture Notes in Computer Science*, Springer, Heidelberg, New York, pp. 45–54.
- [Kon *et al.*, 2002] Kon, F., Costa, F., Blair, G., and Campbell, R. H., 2002. The case for reflective middleware, Communications of the ACM 45 (6), 33–38. <http://doi.acm.org/10.1145/508448.508470>.

- [Kramer, 1994] Kramer, A., 1994. Translucent Patches – Dissolving Windows. In: Proceedings of the 7th annual ACM symposium on User interface software and technology (UIST '94), ACM Press, New York, NY, pp. 121–130.
- [Krasner and Pope, 1988a] Krasner, G. E. and Pope, S. T., 1988a. A Cookbook for Using the Model-View-Controller User Interface Paradigma in Smalltalk-80, *Journal of Object Oriented Programming (JOOP)* 1 (3), 26–49.
- [Krasner and Pope, 1988b] Krasner, G. E. and Pope, S. T., *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*. Parc Place Systems Inc., Mountain View, 1988b.
- [Krebs *et al.*, 2000] Krebs, A. M., Dorohonceanu, B., and Marsic, I., 2000. Collaboration Using Heterogeneous Devices – from 3D Workstations to PDA's. In: Proceedings of the 4th IASTED International Conference Internet and Multimedia Systems and Applications (IMSA'00), pp. 309–313.
- [Kruchten, 1995] Kruchten, P. B., 1995. The 4+1 View Model of Architecture, *IEEE Software* pp. 42–50.
- [Krueger, 1992] Krueger, C. W., 1992. Software reuse, *ACM Computing Surveys (CSUR)* 24 (2), 131–183. <http://doi.acm.org/10.1145/130844.130856>.
- [Kruger *et al.*, 2003] Kruger, R., Carpendale, M., Scott, S., and Greenberg, S., 2003. How People Use Orientation on Tables: Comprehension, Coordination and Communication. In: Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work, [acm, pp. 359–378.
- [Kurtenbach and Buxton, 1994] Kurtenbach, G. and Buxton, W., 1994. User learning and performance with marking menus. In: Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'94), ACM Press, pp. 258–264. <http://doi.acm.org/10.1145/191666.191759>.
- [Landay and Myers, 1993] Landay, J. A. and Myers, B. A., 1993. Extending an existing user interface toolkit to support gesture recognition. In: INTERACT '93 and CHI '93 conference companion on Human factors in computing systems, ACM Press, New York, NY, pp. 91–92.
- [Langheinrich, 2001] Langheinrich, M., 2001. Privacy by Design – Principles of Privacy -Aware Ubiquitous Systems. In: Abowd, G. D., Brummitt, B., and Shafer, S. (eds.), Proceedings of UbiComp'01: Ubiquitous Computing, vol. 2201 of *Lecture Notes in Computer Science*, Springer, Heidelberg, New York, pp. 273–291.
- [Leroy, 2001] Leroy, X., Objective Caml Homepage, <http://pauillac.inria.fr/ocaml/> (2001).
- [Lieberman, 1986] Lieberman, H., 1986. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In: Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA'86), ACM Press, New York, NY, pp. 214–223.
- [Linton *et al.*, 1989] Linton, M. A., Vlissides, J. M., and Calder, P. R., 1989. Composing User Interfaces with InterViews, *IEEE Computer* 22 (2), 8–22. <http://citeseer.nj.nec.com/linton89composing.html>.
- [Luo *et al.*, 1993] Luo, P., Szekely, P., and Neches, R., 1993. Management of Interface Design in HUMANOID. In: Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'93), ACM Press, New York, NY, pp. 107–114.
- [Lyytinen and Yoo, 2002] Lyytinen, K. and Yoo, Y., 2002. Introduction to the Special Issues on Issues and challenges in ubiquitous computing, *Communications of the ACM* 45 (12), 62–65. <http://doi.acm.org/10.1145/585597.585616>.
- [Magerkurth and Prante, 2001a] Magerkurth, C. and Prante, T., 2001a. 'Metaplan' für die Westentasche: Mobile Computerunterstützung für Kreativitätssitzungen. In: Proceedings of Mensch & Computer 2001 (MC '01), Teubner Verlag, pp. 163–171, (in German). <http://ipsi.fraunhofer.de/ambiente/publications/>.
- [Magerkurth and Prante, 2001b] Magerkurth, C. and Prante, T., 2001b. Towards a unifying Approach to Mobile Computing, *SIGGROUP Bulletin* 22 (1), 16–18.
- [Mankoff and Abowd, 1998] Mankoff, J. and Abowd, G. D., 1998. Cirrin: A word-level unistroke keyboard for pen input. In: Proceedings of the 11th annual ACM symposium on User interface software and technology (UIST'98), ACM Press, New York, NY, pp. 213–214. <http://www.cc.gatech.edu/fce/pendragon/publications/cirrin98.pdf>.
- [Mankoff *et al.*, 2000a] Mankoff, J., Hudson, S. E., and Abowd, G. D., 2000a. Interaction techniques for ambiguity resolutions in recognition-based interfaces. In: Proc. of UIST'00, vol. 2, no. 2 in CHI Letters, ACM Press, New York, NY, pp. 11–20.
- [Mankoff *et al.*, 2000b] Mankoff, J., Hudson, S. E., and Abowd, G. D., 2000b. Providing Integrated Toolkit-Level Support for Ambiguity in recognition-Based Interfaces. In: Proceedings of the CHI 2000 conference on Human factors in computing systems (CHI'00), vol. 2, no. 1 in CHI Letters, ACM Press, New York, NY, pp. 368–375.
- [Marshall and Shipman, 1995] Marshall, C. C. and Shipman, F. M. I., 1995. Spatial hypertext: designing for change, *Communications of the ACM* 38 (8), 88–97. <http://doi.acm.org/10.1145/208344.208350>.
- [Marshall *et al.*, 1994] Marshall, C. C., III, F. M. S., and Coombs, J. H., 1994. VIKI: Spatial Hypertext Supporting Emergent Structure. In: Proc. ECHT94, pp. 13–23.

## References

- [Marsic, 2001] Marsic, I., 2001. An Architecture for Heterogeneous Groupware Applications. In: Proceedings of the 23rd International Conference on Software Engineering, IEEE Computer Society Press, pp. 475–484. <http://citeseer.nj.nec.com/marsic01architecture.html>.
- [Martin *et al.*, 2002] Martin, D. A., Morrison, G., Sanoy, C., and McCharles, R., 2002. Simultaneous Multiple-Input Touch Display. In: UbiComp'02 Workshop on "Collaboration with Interactive Walls and Tables". <http://ipsi.fraunhofer.de/ambiente/collabtablewallws/>.
- [Massachusetts Institute of Technology, 2003] MIT Project Oxygen: Pervasive Human-Centered Computing, Massachusetts Institute of Technology, <http://oxygen.lcs.mit.edu/> (2003).
- [Meyer, 1988] Meyer, B., 1988. Object-Oriented Software Construction, Prentice-Hall.
- [Meyer and Bederson, 1998] Meyer, J. and Bederson, B. B., 1998. Does a Sketchy Appearance Influence Drawing Behavior?, HCIL Tech. Rep. 98-12, University of Maryland, Human-Computer Interaction Lab. <http://www.cs.umd.edu/hcil>.
- [Microsoft Research, 2001] EasyLiving Homepage, Microsoft Research, <http://www.research.microsoft.com/easyliving> (2001).
- [Mills and Scholtz, 2001] Mills, K. L. and Scholtz, J., 2001. Situated computing: The Next Frontier for HCI Research. In: Carroll, J. M. (ed.), Human-Computer Interaction in the New Millennium, Addison Wesley, pp. 537–552.
- [Moran and Dourish, 2001] Moran, T. P. and Dourish, P., 2001. Introduction to This Special Issue on Context-Aware Computing, Human-Computer Interaction 16 (2–4), 87–95.
- [Moran *et al.*, 1995] Moran, T. P., Chiu, P., Melle, W. v., and Kurtenbach, G., 1995. Implicit Structures for Pen-Based Systems Within a Freeform Interaction Paradigm. In: Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'95), ACM Press, New York, NY, pp. 487–494.
- [Moran *et al.*, 1997] Moran, T. P., Chiu, P., and Melle, W. v., 1997. Pen-Based Interaction Techniques For Organizing Material on an Electronic Whiteboard. In: Proc. of UIST'97, ACM Press, New York, NY, pp. 45–54.
- [Moran *et al.*, 1998a] Moran, T. P., Melle, W. v., and Chiu, P., 1998a. Spatial Interpretation of Domain Objects Integrated into a Freeform Electronic Whiteboard. In: Proc. of UIST'98, ACM Press, New York, NY, pp. 175–184.
- [Moran *et al.*, 1998b] Moran, T. P., Melle, W. v., and Chiu, P., 1998b. Tailorable Domain Objects as Meeting Tools for an Electronic Whiteboard. In: Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work (CSCW'98), ACM Press, New York, NY, pp. 295–304.
- [Morris *et al.*, 1986] Morris, J. H., Satyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S., and Smith, F. D., 1986. Andrew: a distributed personal computing environment, Communications of the ACM 29 (3), 184–201. <http://doi.acm.org/10.1145/5666.5671>.
- [Müller-Tomfelde and Steiner, 2001] Müller-Tomfelde, C. and Steiner, S., 2001. Audio Enhanced Collaboration at an Electronic White Board. In: Proceedings of 7th International Conference on Auditory Display (ICAD'01), pp. 267–271. <http://ipsi.fraunhofer.de/ambiente/publications/>.
- [Müller-Tomfelde, 2003] Müller-Tomfelde, C., 2003. Sounds@Work—Akustische Repräsentationen für die Mensch-Computer Interaktion in kooperativen und hybriden Arbeitsumgebungen, Ph.D. thesis, Technische Universität Darmstadt, Fachbereich für Elektrotechnik und Informationstechnik, (in German). <http://elib.tu-darmstadt.de/diss/000313/>.
- [Myers, 1990] Myers, B. A., 1990. A New Model for Handling Input, ACM Transactions on Information Systems 8 (3), 289–320.
- [Myers, 1991] Myers, B. A., 1991. ACRONYMS: Acronym Creating Rules On Naming Your Machines and Systems, Tech. Rep. CMU-CS-91-6969, Carnegie Mellon University, College of Computer Science. <http://www-2.cs.cmu.edu/~bam/acronyms.html>.
- [Myers, 1999] Myers, B. A., 1999. An Implementation Architecture to Support Single-Display Groupware, CMU Tech. Rep. CMU-CS-99-139, CMU-HCII-99-101. <http://www.cs.cmu.edu/~pebbles/papers/pebblesarchtr.pdf>.
- [Myers, 2001] Myers, B. A., 2001. Using Handhelds and PCs Together, Communications of the ACM 44 (11), 34–41.
- [Myers, 2003] Myers, B. A., 2003. Graphical User Interface Programming. In: Tucker, A. B. (ed.), CRC Handbook of Computer Science and Engineering (to appear), CRC Press, Inc., Boca Raton, FL, 2nd edition. <http://www.cs.cmu.edu/~bam/pub/uimsCRCrevised.pdf>.
- [Myers and Kosbie, 1996] Myers, B. A. and Kosbie, D. S., 1996. Reusable Hierarchical Command Objects. In: Conference proceedings on Human factors in computing systems (CHI'96), ACM Press, New York, NY. <http://www.cs.cmu.edu/~amulet/papers/commandsCHI.html>.
- [Myers *et al.*, 1992] Myers, B. A., Giuse, D. A., and Zanden, B. V., 1992. Declarative Programming in a Prototype-Instance System: Object-Oriented Programming without Writing Methods. In: Conference proceedings on Object-oriented programming systems, languages, and applications (OOPSLA'92), ACM Press, New York, NY, pp. 184–200.



- [Myers *et al.*, 1996] Myers, B. A., Miller, R. C., McDaniel, R., and Ferreny, A., 1996. Easily Adding Animations to Interfaces Using Constraints. In: Proceedings of the 9th annual ACM symposium on User interface software and technology (UIST'96), ACM Press, New York, NY. <http://www.cs.cmu.edu/~amulet/>.
- [Myers *et al.*, 1997] Myers, B. A. *et al.*, 1997. The Amulet Environment: New Models for Effective User Interface Software Development, *IEEE Transactions on Software Engineering* 23 (6), 347–365. <http://www.cs.cmu.edu/~amulet/>.
- [Myers *et al.*, 1998a] Myers, B. A., McDaniel, R. G., and Miller, R. C., 1998a. The Amulet Prototype-Instance Framework. In: Fayad, M. E. and Johnson, R. E. (eds.), *Domain-specific application frameworks: frameworks experience by industry*, vol. 3, John Wiley & Sons, New York, NY, USA. <http://www.cs.cmu.edu/~amulet/>.
- [Myers *et al.*, 1998b] Myers, B. A., Stiel, H., and Gargiulo, R., 1998b. Collaboration Using Multiple PDAs Connected to a PC. In: Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work (CSCW'98), ACM Press, New York, NY, pp. 285–294. <http://www.cs.cmu.edu/~pebbles>.
- [Myers *et al.*, 2000] Myers, B. A., Hudson, S. E., and Pausch, R., 2000. Past, Present, and Future of User Interface Software Tools, *ACM Transactions on Computer-Human Interaction* 7 (1), 3–28.
- [Myers *et al.*, 2001] Myers, B. A., Peck, C. H., Nichols, J., Kong, D., and Miller, R., 2001. Interacting at a Distance Using Semantic Snarfing. In: Abowd, G. D., Brummitt, B., and Shafer, S. (eds.), *Proceedings of UbiComp'01: Ubiquitous Computing*, vol. 2201 of *Lecture Notes in Computer Science*, Springer, Heidelberg, New York, pp. 305–314. <http://www.cs.cmu.edu/~pebbles>.
- [Myers *et al.*, 2002] Myers, B. A., Malkin, R., Bett, M., Waibel, A., Bostwick, B., Miller, R. C., Yang, J., Denecke, M., Seemann, E., Zhu, J., Peck, C. H., Kong, D., Nichols, J., and Scherlis, B., 2002. Flexi-modal and Multi-Machine User Interfaces. In: *IEEE Fourth International Conference on Multimodal Interfaces*, IEEE CS Press, pp. 343–348. <http://www.cs.cmu.edu/~cpof/papers/cpoficmi02.pdf>.
- [Mynatt, 1999a] Mynatt, E. D., 1999a. Flatland: New Dimensions in Office Whiteboards. In: *Proceeding of the CHI 99 conference on Human factors in computing systems (CHI'99)*, ACM Press, New York, NY, pp. 346–353.
- [Mynatt, 1999b] Mynatt, E. D., 1999b. The Writing on the Wall. In: Sasse, A. and Johnson, C. (eds.), *Proceedings of Human-Computer Interaction (INTERACT'99)*, IOS Press.
- [Mynatt *et al.*, 1998] Mynatt, E. D., Back, M., Want, R., Baer, M., and Ellis, J. B., 1998. Designing Audio Aura. In: *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'98)*, ACM Press, New York, NY, pp. 566–573.
- [MySQL AB, 1995-2004] MySQL Homepage, MySQL AB, <http://www.mysql.com/products/mysql/> (1995-2004).
- [Nichols *et al.*, 2002] Nichols, J., Myers, B. A., Higgins, M., Hughes, J., Harris, T. K., Rosenfeld, R., and Pignol, M., 2002. Generating remote control interfaces for complex appliances. In: *Proceedings of the 15th annual ACM symposium on User interface software and technology*, ACM Press, pp. 161–170. <http://doi.acm.org/10.1145/571985.572008>.
- [Niemela and Vaskivuo, 2004] Niemela, E. and Vaskivuo, T., 2004. Agile Middleware of Pervasive Computing Environments, *PerWare 2004 Middleware Support for Pervasive Computing Workshop* held at the 2nd IEEE Conference on Pervasive Computing and Communications (PerCom'04). <http://csdl.computer.org/comp/proceedings/percomw/2004/2106/00/21060192abs.htm>.
- [Nigay and Coutaz, 1991] Nigay, L. and Coutaz, J., 1991. Building User Interfaces: Organizing Software Agents. In: *Esprit'91 Conference Proceedings*, ACM Press, New York, NY, pp. 707–719. <http://citeseer.nj.nec.com/nigay91building.html>, <http://iihm.imag.fr/publs/1991/>.
- [Nigay and Coutaz, 1995] Nigay, L. and Coutaz, J., 1995. A generic platform for addressing the multimodal challenge. In: *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'95)*, ACM Press/Addison-Wesley Publishing Co., pp. 98–105.
- [Nomura *et al.*, 1998] Nomura, T., Hayashi, K., Hazama, T., and Gudmundson, S., 1998. Interlocus: Workspace Configuration Mechanisms for Activity Awareness. In: *Proceedings of the ACM 1998 Conference on Computer Supported Cooperative Work (CSCW'98)*, ACM Press, New York, NY, pp. 19–28.
- [Nowack, 1999] Nowack, P., 1999. Structures and Interactions—Characterizing Object-Oriented Software Architecture, Ph.D. thesis, Faculty of Software Engineering and Technology, University of Southern Denmark. <http://www.mip.sdu.dk/sweat>.
- [Nunamaker *et al.*, 1991] Nunamaker, J. F., Dennis, A. R., Valacich, J. S., Vogel, D. R., and George, J. F., 1991. Electronic Meeting Systems to Support Group Work, *Communications of the ACM* 34 (7), 40–61.
- [Nunamaker *et al.*, 1995] Nunamaker, J. F., Briggs, R. O., and Mittleman, D. D., 1995. Electronic Meeting Systems: Ten Years of Lessons Learned. In: Coleman, D. and Khanna, R. (eds.), *Groupware: Technology and Applications*, Prentice-Hall, Inc., ch. 6, pp. 149–193.
- [Object Management Group] *Common Object Request Broker Architecture (CORBA) Specification*. Object Management Group, . [http://www.omg.org/technology/documents/formal/corba\\_iiop.htm](http://www.omg.org/technology/documents/formal/corba_iiop.htm).
- [Object Management Group, Inc., 2003] *OMG-Unified Modeling Language, v1.5*. Object Management Group, Inc., 2003. <http://www.omg.org/technology/documents/formal/uml.htm>.

## References

- [O'Hara *et al.*, 2002] O'Hara, K., Churchill, E., Perry, M., Russell, D. M., and Streitz, N. A., 2002. Public, Community and Situated Displays: Design, use and interaction around shared information displays. <http://www.appliancestudio.com/cscw/cscwdisplayworkshopcall.htm>.
- [Olsen, 1998] Olsen, Jr., D. R., 1998. Interacting in chaos. In: Proceedings of the 3rd international conference on Intelligent user interfaces, ACM Press, p. 97. <http://doi.acm.org/10.1145/268389.268407>.
- [Olsen *et al.*, 2000a] Olsen, Jr., D. R., Moyes, W. A., Jefferies, S. S., and Nielsen, S. T., XWeb: An Architecture for Cross-modal Collaboration, [http://icie.cs.byu.edu/Papers/\(2000a\)](http://icie.cs.byu.edu/Papers/(2000a)).
- [Olsen *et al.*, 2000b] Olsen, Jr., D. R., Jefferies, S., Nielsen, T., Moyes, W., and Fredrickson, P., 2000b. Cross-modal interaction using XWeb. In: Proceedings of the 13th annual ACM symposium on User interface software and technology, ACM Press, pp. 191–200. <http://doi.acm.org/10.1145/354401.354764>.
- [Olsen *et al.*, 2001] Olsen, Jr., D. R., Nielsen, S. T., and Parslow, D., 2001. Join and Capture: A Model for Nomadic Interaction. In: Proceedings of 14th Annual ACM Symposium on User Interface and Software Technology (UIST'01), vol. 3, no. 2 in CHI Letters, ACM Press, New York, NY, pp. 131–140.
- [Opdyke and Johnson, 1990] Opdyke, W. F. and Johnson, R. E., 1990. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In: SOOPPA Conference Proceedings, ACM Press, New York, NY, pp. 145–161.
- [OpenOffice.org, 2004] OpenOffice Homepage, OpenOffice.org, <http://www.openoffice.org/> (2004).
- [Oviatt *et al.*, 1997] Oviatt, S., DeAngeli, A., and Kuhn, K., 1997. Integration and Synchronization of Input Modes during Multimodal Human-Computer Interaction. In: Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'97), ACM Press, New York, NY, pp. 415–422.
- [Oviatt *et al.*, 2000] Oviatt, S., Cohen, P., Wu, L., Vergo, J., Duncan, L., Suhm, B., Bers, J., Holzman, T., Winograd, T., and Jam, 2000. Designing the User Interface for Multimodal Speech and Pen-Based Gesture Applications: State-of-the-Art Systems and Future Research Directions. In: Carroll, J. A. (ed.), Human-Computer Interaction in the New Millennium, Addison Wesley, pp. 421–456.
- [Paepcke, 1993] Paepcke, A. (ed.), 1993. Object-Oriented Programming: The CLOS Perspective, MIT Press.
- [Palm, Inc., 2003] Palm Homepage, Palm, Inc., <http://www.palm.com/> (2003).
- [Pangaro *et al.*, 2002] Pangaro, G., Maynes-Aminzade, D., and Ishii, H., 2002. The actuated workbench: computer-controlled actuation in tabletop tangible interfaces. In: Proceedings of the 15th annual ACM symposium on User interface software and technology, ACM Press, pp. 181–190. <http://doi.acm.org/10.1145/571985.572011>.
- [ParcPlace-Digitalk, Inc., 1995] *VisualWorks User's Guide*. ParcPlace-Digitalk, Inc., 999 East Arques Avenue, Sunnyvale, CA, Revision 2.0 (Software Release 2.5), 1995.
- [Parnas, 1972] Parnas, D. L., 1972. On the criteria to be used in decomposing systems into modules, Communications of the ACM 15 (12), 1053–1058. <http://doi.acm.org/10.1145/361598.361623>.
- [Patten *et al.*, 2001] Patten, J., Ishii, H., Hines, J., and Pangaro, G., 2001. Sensetable: A Wireless Object Tracking Platform for Tangible User Interfaces. In: Proceedings of CHI 2001, ACM Press, New York, NY.
- [Patterson, 1991] Patterson, J. F., 1991. Comparing the Programming Demands of Single-User and Multi-User Applications. In: Proceedings of ACM SIGGRAPH Symposium on User Interface Software and Technology (UIST'91), ACM Press, New York, NY, pp. 87–94.
- [Patterson, 1995] Patterson, J. F., 1995. A taxonomy of architectures for synchronous groupware applications, workshop on Software architectures for cooperative systems CSCW'94, ACM SIGOIS Bulletin Special Issue "Papers of the CSCW'94 workshops" 15 (3).
- [Patterson *et al.*, 1990] Patterson, J. F., Hill, R. D., Rohall, S. L., and Meeks, W. S., 1990. Rendezvous: An Architecture for Synchronous Multi-User Applications. In: Proceedings of the ACM 1990 Conference on Computer Supported Cooperative Work (CSCW'90), ACM Press, New York, NY, pp. 317–328.
- [Pedersen *et al.*, 1993] Pedersen, E. R., McCall, K., Moran, T. P., and Halasz, F. G., 1993. Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings. In: Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'93), ACM Press, New York, NY, pp. 391–398.
- [Perlin, 1998] Perlin, K., 1998. Quikwriting: Continuous Stylus-based Text Entry. In: Proceedings of the 11th annual ACM symposium on User interface software and technology (UIST'98), ACM Press, New York, NY, pp. 215–216.
- [Perry and Wolf, 1992] Perry, D. E. and Wolf, A. L., 1992. Foundations for the Study of Software Architecture, ACM SIGSOFT Software Engineering Notes (4), 40–52.
- [Pfaff, 1985] Pfaff, G. E. (ed.), 1985. Proceedings of the Workshop on User Interface Management Systems held in Seeheim, FRG, Nov 1-3, 1983, Springer, Heidelberg, New York.
- [Phillips, 1999] Phillips, W. G., 1999. Architectures for Synchronous Groupware, Tech. Rep. 1999-425, Queen's University, Kingston, Ontario K7L 3N6. <http://phillips.rmc.ca/greg/pub/>.
- [Pier and Landay, 1992] Pier, K. and Landay, J. A., 1992. Issues for Location-independent Interfaces, Tech. Rep. ISTL92-4, Xerox PARC, Palo Alto, CA, USA. <http://www.cs.berkeley.edu/~landay/research/publications/LII.ps>, <http://citeseer.nj.nec.com/pier92issues.html>.

- [Pierce and Mahaney, 2004] Pierce, J. S. and Mahaney, H., 2004. Opportunistic Annexing for Handheld Devices: Opportunities and Challenges. In: Paper presented at the Human Computer Interaction Consortium 2004 Winter Workshop (HCIC'04). <http://www.cc.gatech.edu/projects/PIE/pubs/>.
- [Pingali et al., 2003] Pingali, G., Pinhanez, C., Levas, A., Kjeldsen, R., Podlaseck, M., Chen, H., and Sukaviriya, N., 2003. Steerable Interfaces for Pervasive Computing Spaces. In: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom'03), IEEE Computer Society. [http://percom.org/percom\\_2003/](http://percom.org/percom_2003/).
- [Ponnekanti et al., 2001] Ponnekanti, S. R., Lee, B., Fox, A., Hanrahan, P., and Winograd, T., 2001. ICrafter: A Service Framework for Ubiquitous Computing Environments. In: Abowd, G. D., Brummitt, B., and Shafer, S. (eds.), Proceedings of UbiComp'01: Ubiquitous Computing, vol. 2201 of *Lecture Notes in Computer Science*, Springer, Heidelberg, New York, pp. 56–75. [http://graphics.stanford.edu/papers/icrafter\\_ubicomp01/](http://graphics.stanford.edu/papers/icrafter_ubicomp01/).
- [Ponnekanti et al., 2003] Ponnekanti, S. et al., 2003. Portability, Extensibility and Robustness in iROS. In: 1st IEEE International Conference on Pervasive Computing and Communications (PerCom 2003), IEEE Computer Society, pp. 11–19. [http://percom.org/percom\\_2003/](http://percom.org/percom_2003/).
- [Popovici et al., 2003] Popovici, A., Alonso, G., and Gross, T., 2003. Spontaneous Container Services. In: Cardelli, L. (ed.), 17th European Conference on Object-Oriented Programming (ECOOP'2003), vol. 2743 of *Lecture Notes in Computer Science*, Springer, Heidelberg, New York, pp. 29–53. <http://www.ecoop.org>, <http://www.informatik.uni-trier.de/~ley/db/conf/ecoop/ecoop2003.html>.
- [Potel, 2000] Potel, M., MVP: *Model-Viewer-Presenter*. Taligent, Inc, 2000. <http://www.ibm.com/developerworks/java/library/j-mvp.html>.
- [Prante, 1999] Prante, T., 1999. Eine neue stiftzentrierte Benutzungsoberfläche zur Unterstützung kreativer Teamarbeit in Roomware-Umgebungen, Diplomarbeit, GMD-IPSI, Darmstadt Technical University, Department of Computer Science, (in German).
- [Prante, 2001] Prante, T., 2001. Designing for Usable Disappearance—Mediating Coherence, Scope, and Orientation. In: Workshop Proceedings "Distributed and Disappearing User Interfaces in Ubiquitous Computing", ACM CHI'01. <http://www.teco.edu/chi2001ws/proceedings.html>.
- [Prante et al., 2002] Prante, T., Magerkurth, C., and Streitz, N. A., 2002. Developing CSCW Tools for Idea Finding—Empirical Results and Implications for Design. In: Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work (CSCW'02), ACM Press, New York, NY, pp. 106–115. <http://ipsi.fraunhofer.de/ambiente/publications/>.
- [Pree, 1999] Pree, W., 1999. Hot-Spot-Driven Development. In: Fayad, M. E., Schmidt, D. C., and Johnson, R. E. (eds.), Building Application Frameworks: Object-Oriented Foundations of Framework Design, John Wiley & Sons, New York, NY, USA, ch. 16, pp. 379–393.
- [Rekimoto, 1997] Rekimoto, J., 1997. Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments. In: Proceedings of the 10th annual ACM symposium on User interface software and technology (UIST'97), ACM Press, New York, NY, pp. 31–39.
- [Rekimoto, 1998a] Rekimoto, J., 1998a. Multiple-Computer User Interfaces: A cooperative environment consisting of multiple digital devices. In: Streitz, N., Konomi, S., and Burkhardt, H. (eds.), Cooperative Buildings – Integrating Information, Organization and Architecture. First International Workshop on Cooperative Buildings (CoBuild'98), vol. 1370 of *Lecture Notes in Computer Science*, Springer, Heidelberg, New York, pp. 33–40.
- [Rekimoto, 1998b] Rekimoto, J., 1998b. A Multiple Device Approach for Supporting Whiteboard-based Interactions. In: Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'98), ACM Press, New York, NY, pp. 344–351.
- [Rekimoto, 2002] Rekimoto, J., 2002. SmartSkin: an infrastructure for freehand manipulation on interactive surfaces. In: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM Press, pp. 113–120. <http://doi.acm.org/10.1145/503376.503397>.
- [Rekimoto and Matsushita, 1997] Rekimoto, J. and Matsushita, N., 1997. Perceptual Surfaces: Towards a Human and Object Sensitive Interactive Display. In: Workshop on Perceptual User Interfaces (PUI'97).
- [Rekimoto and Saitoh, 1999] Rekimoto, J. and Saitoh, M., 1999. Augmented Surfaces: A Spatially Continuous Work Space for Hybrid Computing Environments. In: Proceeding of the CHI 99 conference on Human factors in computing systems: the CHI is the limit (CHI'99), ACM Press, New York, NY, pp. 378–385.
- [Robertson and Mackinlay, 1993] Robertson, G. G. and Mackinlay, J. D., 1993. The Document Lens. In: Proceedings of the sixth annual ACM symposium on User interface software and technology, ACM Press, New York, NY, pp. 101–108. <http://doi.acm.org/10.1145/168642.168652>.
- [Roberts et al., 1997] Roberts, D., Brant, J., and Johnson, R., 1997. A Refactoring Tool for Smalltalk. In: Pattern Languages of Program Design, vol. 3 of *Software Patterns*, Addison Wesley, ch. 25. <http://www.cs.uiuc.edu/~brant/Refactory/>.
- [Román, 2003] Román, M., 2003. An Application Framework for Active Space Applications, Ph.D. thesis, University of Illinois at Urbana-Champaign. <http://choices.cs.uiuc.edu/gaia/html/publications.htm>.

## References

- [Román and Campbell, 2001] Román, M. and Campbell, R. H., 2001. A Model for Ubiquitous Applications, Tech. Rep. UIUCDCS-R-2001-2223, UILU-ENG-2001-1730, University of Illinois at Urbana-Champaign, Department of Computer Science. <http://choices.cs.uiuc.edu/gaia/papers/appmodel-tech01.pdf>.
- [Román *et al.*, 2001a] Román, M., Kon, F., and Campbell, R. H., 2001a. Reflective Middleware: From Your desk to Your Hand, IEEE Distributed Systems Online 2 (5). [http://dsonline.computer.org/0105/features/rom0105\\_print.htm](http://dsonline.computer.org/0105/features/rom0105_print.htm).
- [Román *et al.*, 2001b] Román, M. *et al.*, 2001b. GaiaOS: An Infrastructure for Active Spaces, Tech. Rep. UIUCDCS-R-2001-2224, UILU-ENG-2001-1731, University of Illinois at Urbana-Champaign, Department of Computer Science, 1304 West Springfield Avenue, Urbana, IL, 61801-2987 USA. <http://choices.cs.uiuc.edu/gaia/>.
- [Román *et al.*, 2002] Román, M., Hess, C. K., Cerqueira, R., Ranganathan, A., Campbell, R. H., and Nahrstedt, K., 2002. Gaia: A Middleware Infrastructure to Enable Active Spaces, IEEE Pervasive Computing pp. 74–83. <http://choices.cs.uiuc.edu/gaia/html/publications.htm>.
- [Roseman and Greenberg, 1992] Roseman, M. and Greenberg, S., 1992. GROU PKIT: a groupware toolkit for building real-time conferencing applications. In: Proceedings of the conference on Computer-supported cooperative work, ACM Press, pp. 43–50. <http://doi.acm.org/10.1145/143457.143460>.
- [Roseman and Greenberg, 1996a] Roseman, M. and Greenberg, S., 1996a. TeamRooms: Network Places for Collaboration. In: Proceedings of the ACM 1996 Conference on Computer Supported Cooperative Work (CSCW'96), ACM Press, New York, NY, pp. 325–333.
- [Roseman and Greenberg, 1996b] Roseman, M. and Greenberg, S., 1996b. Building Real Time Groupware with GroupKit, A Groupware Toolkit, ACM Transactions on Computer-Human Interaction 3 (1), 66–106.
- [Roth, 2000] Roth, J., 2000. 'DreamTeam': A Platform for Synchronous Collaborative applications, AI & Society 14 (1), 98–119.
- [Roth, 2002] Roth, J., 2002. Seven Challenges for Developers of Mobile Groupware. In: Workshop "Mobile Ad Hoc Collaboration", CHI 2002, Minneapolis.
- [Roth, 2003] Roth, J., 2003. Accessing Location Data in Mobile Environments—the Nimbus Location Model. In: Mobile HCI 03 Workshop on Mobile and Ubiquitous Information Access, vol. 2954 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 256–270.
- [Roth and Unger, 2000] Roth, J. and Unger, C., 2000. Using Handheld Devices in Synchronous Collaborative Scenarios. In: Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC'00), vol. 1927 of *LNCS*, Springer Verlag, pp. 187–199. <http://citeseer.nj.nec.com/article/roth00using.html>.
- [Rubine, 1991] Rubine, D., 1991. Specifying Gestures by Example. In: Proceedings of the 18th annual conference on Computer graphics and interactive techniques, ACM Press, pp. 329–337. <http://doi.acm.org/10.1145/122718.122753>.
- [Russell and Gossweiler, 2001] Russell, D. M. and Gossweiler, R., 2001. On the Design of Personal & Communal Large Information Scale Appliances. In: Abowd, G. D., Brummitt, B., and Shafer, S. (eds.), Proceedings of UbiComp'01: Ubiquitous Computing, vol. 2201 of *Lecture Notes in Computer Science*, Springer, Heidelberg, New York, pp. 354–361.
- [Russell and Weiser, 1998] Russell, D. M. and Weiser, M., 1998. The Future of Integrated Design of Ubiquitous Computing in Combined Real & Virtual Worlds. In: CHI'98, Late-Breaking Results: "The Real and the Virtual: Integrating Architectural and Information Spaces (Suite)", ACM Press, New York, NY, pp. 275–276.
- [Rutgers University, 2002] Manifold Homepage, Rutgers University, <http://www.caip.rutgers.edu/disciple/> (2002).
- [Salber *et al.*, 1999] Salber, D., Dey, A. K., and Abowd, G. D., 1999. The Context Toolkit: Aiding the Development of Context-Enabled Applications. In: Proceeding of the CHI 99 conference on Human factors in computing systems (CHI99), ACM Press, New York, NY, pp. 434–441.
- [Schilit, 1995] Schilit, W. N., 1995. A System Architecture for Context-Aware Mobile Computing, Ph.D. thesis, Columbia University. <http://www.fxpai.xerox.com/people/schilit/schilit-thesis.pdf>.
- [Schmid, 1997] Schmid, H. A., 1997. Systematic Framework Design by Generalization—How to deduce a hot spot implementation from its specification, Communications of the ACM 40 (10), 48–51.
- [Schmid, 1999] Schmid, H. A., 1999. Framework Design by Systematic Generalization. In: Fayad, M. E., Schmidt, D. C., and Johnson, R. E. (eds.), Building Application Frameworks: Object-Oriented Foundations of Framework Design, John Wiley & Sons, New York, NY, USA, ch. 15, pp. 353–378.
- [Schmidt, 2000] Schmidt, A., 2000. Implicit Human Computer Interaction Through Context, Personal Technologies 4 (2+3), 191–199. <http://www.teco.edu/>.
- [Schmidt *et al.*, 1999] Schmidt, A., Beigl, M., and Gellersen, H., 1999. There is more to Context than Location, Computer & Graphics 23 (6), 893–902. <http://www.elsevier.com>.
- [Schuckmann *et al.*, 1996] Schuckmann, C., Kirchner, L., Schümmer, J., and Haake, J. M., 1996. Designing Object-oriented Synchronous Groupware with COAST. In: Proceedings of the ACM 1996 Conference on Com-

- puter Supported Cooperative Work (CSCW'96), ACM Press, New York, NY, pp. 30–38.  
<http://doi.acm.org/10.1145/240080.240186>.
- [Schuckmann *et al.*, 1999] Schuckmann, C., Schümmer, J., and Seitz, P., 1999. Modeling Collaboration using Shared Objects. In: Proceedings of International ACM SIGGROUP Conference on Supporting Group Work (GROUP'99), ACM Press, New York, NY, pp. 189–198. <http://www.opencoast.org>.
- [Schümmer *et al.*, 2000] Schümmer, J., Schümmer, T., and Schuckmann, C., 2000. COAST – Ein Anwendungsframework für synchrone Groupware. In: Conference Proceedings for the "Net.ObjectDays 2000". <http://www.opencoast.org>.
- [Scott *et al.*, 2003] Scott, S. D., Grant, K. D., and Mandryk, R. L., 2003. System Guidelines for Co-located, Collaborative Work on a Tabletop Display. In: Proceedings of the 18th European Conference on Computer Supported Cooperative Work (ECSCW'03), Kluwer Academic Publishers, Amsterdam, NL.
- [Seitz, 1997] Seitz, P., 1997. Entwurf einer objektorientierten Experiment- und Modellbeschreibungssprache zur Validierung dynamischer Verhaltensmodelle technischer Systeme, Diplomarbeit, TH Darmstadt, Institut für Systemarchitektur, Fachgebiet Praktische Informatik, (in German). <http://www.tandlers.de/peter/Prima/>.
- [Seitz, 1999] Seitz, P., 1999. Connecting Roomware: The Software Infrastructure of i-LAND. In: Boaster paper presented at 10. Human-Computer Interaction Consirtium Winter Conference (HCIC'99). <http://ipsi.fraunhofer.de/ambiente/publications/>.
- [Shafer, 2001] Shafer, S. A. N., 2001. Ubiquitous Computing and the EasyLiving Project. In: 40th Anniversary Symposium, Osaka Electro-Communications University. <http://www.research.microsoft.com/easyliving/>.
- [Shafer *et al.*, 1998] Shafer, S., Krumm, J., Brumitt, B., Meyers, B., Czerwinski, M., and Robbins, D., 1998. The New EasyLiving Project at Microsoft Research. In: Proc. of Joint DARPA / NIST Workshop on Smart Spaces, pp. 127–130. <http://research.microsoft.com/easyliving/Documents/1998%2007%20DARPA%20workshop.pdf>.
- [Shafer *et al.*, 2000] Shafer, S., Brumitt, B., and Meyers, B., 2000. The EasyLiving Intelligent Environment System. In: CHI Workshop on Research Directions in Situated Computing, ACM Press, New York, NY. <http://www.research.microsoft.com/easyliving>.
- [Shafer *et al.*, 2001] Shafer, S. A. N., Brumitt, B., and Cadiz, J., 2001. Interaction Issues in Context-Aware Intelligent Environments, *Human-Computer Interaction* 16 (2–4), 363–378.
- [Shaw *et al.*, 1995] Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., and Zelesnik, G., 1995. Abstractions for Software Architecture and Tools to Support Them, *IEEE Transactions on Software Engineering* 21 (4), 314–335. [http://www-2.cs.cmu.edu/~Vit/paper\\_abstracts/UniCon.html](http://www-2.cs.cmu.edu/~Vit/paper_abstracts/UniCon.html).
- [Shen *et al.*, 2002] Shen, C., Vernier, F., Lesh, N., and Forlines, C., 2002. Around the Table.
- [Shipman and Marshall, 1999] Shipman, F. and Marshall, C., 1999. Formality Considered Harmful: Experiences, Emerging Themes, and Directions on the Use of Formal Representations in Interactive Systems, *Computer Supported Cooperative Work (CSCW)* 8 (4), 333–352.
- [Shoemaker and Inkpen, 2001] Shoemaker, G. B. D. and Inkpen, K. M., 2001. Single display privacyware: augmenting public displays with private information. In: Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'01), ACM Press, New York, NY, pp. 522–529.
- [Shrobe *et al.*, 2002] Shrobe, H., Quigley, K., Peters, S., Kochman, R., Kleek, M. V., Look, G., and Gajos, K., 2002. Building Software Infrastructure for Human-Centric Pervasive Computing, MIT Artificial Intelligence Laboratory, pp. 412–413. <http://www.ai.mit.edu/projects/aire/papers.shtml>.
- [SMART Technologies Inc., 2002] SMART Technologies Homepage, SMART Technologies Inc., <http://www.smarttech.com> (2002).
- [Software AG, 2004] Adabas D Homepage, Software AG, <http://www.softwareag.com/adabas/> (2004).
- [Sousa and Garlan, 2002] Sousa, J. and Garlan, D., 2002. Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. In: *Software Architecture: System Design, Development, and Maintenance (Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture)*, pp. 29–43. <http://www.cs.cmu.edu/~aura/>.
- [Stanford University, 2000] Stanford Interactive Workspaces Project, Stanford University, <http://graphics.stanford.edu/projects/iwork> (2000).
- [Steele, 1990] Steele, G. L., 1990. *Common Lisp: The Language*, Digital Press, Bedford, 2. edition.
- [Stefik *et al.*, 1987a] Stefik, M., Foster, G., Bobrow, D. G., Kahn, K., Lanning, S., and Suchman, L., 1987a. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings, *Communications of the ACM* 30 (1), 32–47.
- [Stefik *et al.*, 1987b] Stefik, M., Bobrow, D. G., Foster, G., Lanning, S., and Tatar, D., 1987b. WYSIWIS Revised: Early Experiences with Multi-User Interfaces, *ACM Transactions on Information Systems* 2 (5), 147–167.
- [Steiner, 1999] Steiner, S., 1999. Audiounterstützung für Gruppenarbeitssituationen an der DynaWall, Studienarbeit, Technische Universität Darmstadt, Fachbereich für Elektrotechnik und Informationstechnik, (in German).

## References

- [Stewart *et al.*, 1998] Stewart, J., Raybourn, E. M., Bederson, B., and Druin, A., 1998. When Two Hands Are Better Than One: Enhancing Collaboration Using Single Display Groupware. In: ACM SIGCHI '98 late-breaking results, ACM Press, New York, NY.
- [Stewart *et al.*, 1999] Stewart, J., Bederson, B. B., and Druin, A., 1999. Single Display Groupware: A Model for Co-present Collaboration. In: Proceeding of the CHI 99 conference on Human factors in computing systems (CHI'99), ACM Press, New York, NY, pp. 286–293. <http://www.cs.umd.edu/hcil>.
- [Streitz, 2001] Streitz, N. A., 2001. Mental vs. Physical Disappearance: The Challenge of Interacting with Disappearing Computers. In: Workshop Proceedings "Distributed and Disappearing User Interfaces in Ubiquitous Computing", ACM CHI'01. <http://www.teco.edu/chi2001ws/proceedings.html>.
- [Streitz *et al.*, 1994] Streitz, N. A., Geißler, J., Haake, J. M., and Hol, J., 1994. DOLPHIN: Integrated Meeting Support across Local and Remote Desktop Environments and LiveBoards. In: Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work (CSCW'94), ACM Press, New York, NY, pp. 345–358. <http://ipsi.fraunhofer.de/ambiente/publications/>.
- [Streitz *et al.*, 1997] Streitz, N. A., Rexroth, P., and Holmer, T., 1997. Does 'roomware' matter? Investigating the role of personal and public information devices and their combination in meeting room collaboration. In: Proceedings of the European Conference on Computer-Supported Cooperative Work (E-CSCW'97), Kluwer Academic Publishers, Amsterdam, NL, pp. 297–312. <http://ipsi.fraunhofer.de/ambiente/publications/>.
- [Streitz *et al.*, 1998] Streitz, N., Rexroth, P., and Holmer, T., 1998. Anforderungen an interaktive Kooperationslandschaften für kreatives Arbeiten und erste Realisierungen. In: Tagungsband der D-CSCW'98, B.G.Teubner, Stuttgart, Leipzig, pp. 237–250, (in German).
- [Streitz *et al.*, 1999] Streitz, N. A., Geißler, J., Holmer, T., Konomi, S., Müller-Tomfelde, C., Reischl, W., Rexroth, P., Seitz, P., and Steinmetz, R., 1999. i-LAND: An interactive Landscape for Creativity and Innovation. In: Proceeding of the CHI 99 conference on Human factors in computing systems (CHI'99), ACM Press, New York, NY, pp. 120–127. <http://ipsi.fraunhofer.de/ambiente/publications/>.
- [Streitz *et al.*, 2001] Streitz, N. A., Tandler, P., Müller-Tomfelde, C., and Konomi, S., 2001. Roomware: Towards the next generation of human-computer interaction based on an integrated design of real and virtual worlds. In: Carroll, J. A. (ed.), Human-Computer Interaction in the New Millennium, Addison Wesley, pp. 553–578. <http://ipsi.fraunhofer.de/ambiente/publications/>.
- [Streitz *et al.*, 2002] Streitz, N. A., Prante, T., Müller-Tomfelde, C., Tandler, P., and Magerkurth, C., 2002. Roomware: The Second Generation. In: Video Proceedings and Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI'02), ACM Press, New York, NY, pp. 506–507. <http://ipsi.fraunhofer.de/ambiente/publications/>.
- [Succi *et al.*, 1999] Succi, G., Predonzani, P., Valerio, A., and Vernazza, T., 1999. Frameworks and Domain Models: Two Sides of the Same Coin. In: Fayad, M. E., Schmidt, D. C., and Johnson, R. E. (eds.), Building Application Frameworks: Object-Oriented Foundations of Framework Design, vol. 1, John Wiley & Sons, New York, NY, USA, pp. 211–214.
- [Sun Microsystems, 1999] *Jini™ Technology Architectural Overview*. Sun Microsystems, 1999. <http://www.sun.com/software/jini/whitepapers/architecture.html>.
- [Sun Microsystems, 2000] *JavaSpaces Technology*, Sun Microsystems, <http://www.javasoft.com/products/javaspaces/> (2000).
- [Sun Microsystems, 2001] *Jini™ Technology Core Platform Specification*. Sun Microsystems, version 1.2, 2001. <http://www.sun.com/jini/specs/>.
- [Sun Microsystems, 2002] *JavaSpaces™ Service Specification*. Sun Microsystems, version 1.2.1, 2002. <http://www.sun.com/jini/specs/>.
- [Sun Microsystems, 2004] *StarOffice Homepage*, Sun Microsystems, <http://www.sun.com/software/star/staroffice/> (2004).
- [Sussman *et al.*, 2001] Sussman, J., Banavar, G., and Bergman, L., 2001. Generalization: A Key Concept in the Creation of Platform Independent User Interfaces. In: UbiTools'01–Workshop on Application Models and Programming Tools for Ubiquitous Computing (held in conjunction with the UbiComp'01). <http://choices.cs.uiuc.edu/UbiTools01/>.
- [Swaminathan and Sato, 1997] Swaminathan, K. and Sato, S., 1997. Interaction Design for Large Displays, interactions pp. 15–24.
- [Szekely, 1990] Szekely, P., 1990. Template-based mapping of application data interactive displays. In: Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology, ACM Press, pp. 1–9. <http://doi.acm.org/10.1145/97924.97925>.
- [Szekely and Myers, 1988] Szekely, P. and Myers, B., 1988. A user interface toolkit based on graphical objects and constraints. In: Conference proceedings on Object-oriented programming systems, languages and applications, ACM Press, pp. 36–45. <http://doi.acm.org/10.1145/62083.62088>.
- [Szekely *et al.*, 1992] Szekely, P., Luo, P., and Neches, R., 1992. Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. In: Proceedings of the ACM Conference on Hu-

- man Factors in Computing Systems (CHI'92), ACM Press, pp. 507–515.  
<http://doi.acm.org/10.1145/142750.142912>.
- [Szekely *et al.*, 1993] Szekely, P., Luo, P., and Neches, R., 1993. Beyond interface builders: Model-based interface tools. In: Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'93), ACM Press, New York, NY, pp. 383–390.
- [Tandler, 2001a] Tandler, P., 2001a. Modeling Groupware Supporting Synchronous Collaboration with Heterogeneous Single- and Multi-User Devices. In: Bogers, M. R. S., Haake, J. M., and Hoppe, H. U. (eds.), Proceedings of 7th International Workshop on Groupware (CRIWG'01), IEEE CS Press, pp. 6–15.
- [Tandler, 2001b] Tandler, P., 2001b. Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices. In: Abowd, G. D., Brummitt, B., and Shafer, S. (eds.), Proceedings of UbiComp'01: Ubiquitous Computing, vol. 2201 of *Lecture Notes in Computer Science*, Springer, Heidelberg, New York, pp. 96–115.
- [Tandler, 2001c] Tandler, P., 2001c. The BEACH Application Model and Software Framework for Synchronous Collaboration in Ubiquitous Computing Environments. In: UbiTools'01–Workshop on Application Models and Programming Tools for Ubiquitous Computing (held in conjunction with the UbiComp'01).  
<http://choices.cs.uiuc.edu/UbiTools01/>, <http://ipsi.fraunhofer.de/ambiente/publications/>.
- [Tandler, 2003] Tandler, P., 2003. Working at the Intersection of UbiComp, HCI, CSCW, and Software Technology. In: Johanson, B., Borchers, J., Schiele, B., Tandler, P., and Edwards, K. (eds.), At the Crossroads: The Interaction of HCI and Systems Issues in UbiComp, UbiComp '03 workshop. <http://ubihcisys.stanford.edu/>.
- [Tandler, 2004] Tandler, P., 2004. The BEACH Application Model and Software Framework for Synchronous Collaboration in Ubiquitous Computing Environments, *Journal of Systems & Software (JSS) Special issue on Ubiquitous Computing* 69 (3), 267–296. <http://ipsi.fraunhofer.de/ambiente/publications/>,  
<http://authors.elsevier.com/sd/article/S0164121203000554>.
- [Tandler *et al.*, 2001] Tandler, P., Prante, T., Müller-Tomfelde, C., Streitz, N., and Steinmetz, R., 2001. Connectables: Dynamic Coupling of Displays for the Flexible Creation of Shared Workspaces. In: Proceedings of 14th Annual ACM Symposium on User Interface and Software Technology (UIST'01), vol. 3, no. 2 in CHI Letters, ACM Press, New York, NY, pp. 11–20. <http://ipsi.fraunhofer.de/ambiente/publications/>.
- [Tandler *et al.*, 2002a] Tandler, P., Magerkurth, C., Carpendale, S., and Inkpen, K., UbiComp'02 Workshop on "Collaboration with Interactive Walls and Tables", Göteborg, Sweden,  
[http://ipsi.fraunhofer.de/ambiente/collabtablewalls/\(2002a\)](http://ipsi.fraunhofer.de/ambiente/collabtablewalls/(2002a)).
- [Tandler *et al.*, 2002b] Tandler, P., Streitz, N., and Prante, T., 2002b. Roomware—Moving Toward Ubiquitous Computers, *IEEE Micro* 22 (6), 36–47. <http://ipsi.fraunhofer.de/ambiente/publications/>.
- [Tarpin-Bernard *et al.*, 1998] Tarpin-Bernard, F., David, B., and Primet, P., 1998. Frameworks and patterns for synchronous groupware: AMF-C approach. In: IFIP Working Conference on Engineering for HCI: EHCI'98, pp. 225–242. <http://citeseer.nj.nec.com/439075.html>.
- [Tarr *et al.*, 1999] Tarr, P. L., Ossher, H., Harrison, W. H., and Jr., S. M. S., 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proceedings of the 21st International Conference on Software Engineering (ICSE'99), ACM Press, New York, NY, pp. 107–119. [citeseer.nj.nec.com/tarr99degrees.html](http://citeseer.nj.nec.com/tarr99degrees.html).
- [Tatar *et al.*, 1991] Tatar, D. G., Foster, G., and Bobrow, D. G., 1991. Design for Conversation: Lessons from Cognition, *International Journal of Man-Machine Studies* 34 (2), 185–209.
- [Taylor *et al.*, 1995] Taylor, R. N., Nies, K. A., Bolcer, G. A., MacFarlane, C. A., Anderson, K. M., and Johnson, G. F., 1995. Chiron-1: a software architecture for user interface development, maintenance, and run-time support, *ACM Transactions on Computer-Human Interaction* 2 (2), 105–144.  
<http://doi.acm.org/10.1145/210181.210182>.
- [Taylor *et al.*, 1996] Taylor, R. N. et al., 1996. A Component- and Message-Based Architectural Style for GUI Software, *IEEE Transactions on Software Engineering* 22 (6), 390–406.
- [ter Hofte, 1998] ter Hofte, G. H., 1998. Working Apart Together – Foundations for Component Groupware, Ph.D. thesis, Telematica Instituut, NL, P.O. Box 589, 7500 AN Enschede, The Netherlands.  
<https://doc.telin.nl/dscgi/ds.py/Get/File-7623/wath.pdf>.
- [Thevenin and Coutaz, 1999] Thevenin, D. and Coutaz, J., 1999. Plasticity of User Interfaces: Framework and Research Agenda. In: Proceedings of Human-Computer Interaction (INTERACT'99), IOS Press, pp. 110–117.
- [Ullmer and Ishii, 2000] Ullmer, B. and Ishii, H., 2000. Emerging Frameworks for Tangible User Interfaces, *IBM Systems Journal* (3&4), 915–931.
- [Ullmer *et al.*, 1998] Ullmer, B., Ishii, H., and Glas, D., 1998. mediaBlocks: Physical Containers, Transports, and Controls for Online Media. In: Computer Graphics Proceedings (SIGGRAPH'98), ACM Press, New York, NY, pp. 379–386. <http://doi.acm.org/10.1145/280814.280940>.
- [Ungar and Smith, 1987] Ungar, D. and Smith, R. B., 1987. Self: The Power of Simplicity. In: Meyrowitz, N. (ed.), OOPSLA '87 Conference Proceedings, ACM Press, pp. 227–242. <http://self.sml.com/papers/self-power.html>,  
<http://self.sml.com/papers/selfPower.ps.Z>.

## References

- [University of Illinois at Urbana-Champaign, 2002] Gaia Project Homepage, University of Illinois at Urbana-Champaign, <http://choices.cs.uiuc.edu/gaia/> (2002).
- [University of Washington, 2003] one.world Homepage, University of Washington, <http://one.cs.washington.edu> (2003).
- [Urnes and Graham, 1999] Urnes, T. and Graham, T. N., 1999. Flexibly Mapping Synchronous Groupware Architectures to Distributed Implementations. In: *Proceedings of Design, Specification and Verification of Interactive Systems (DSV-IS'99)*, Springer, Heidelberg, New York, pp. 133–147. <http://dundee.cs.queensu.ca/~graham/stl/pubs>.
- [Wacom Technology Co., 2003] Wacom Homepage, Wacom Technology Co., <http://www.wacom.com/> (2003).
- [Wall *et al.*, 2002] Wall, L. *et al.*, *Perl Reference Manual*. 2002. <http://www.perldoc.com/perl5.8.0/>.
- [Walrath and Campione, 1999] Walrath, K. and Campione, M., 1999. The JFC Swing Tutorial: A Guide to Constructing GUIs, Addison Wesley. <http://java.sun.com/docs/books/tutorial/books/swing/>.
- [Wang *et al.*, 1999] Wang, W., Dorohonceanu, B., and Marsic, I., 1999. Design of the DISCIPLE Synchronous Collaboration Framework. In: *Proceedings of the 3rd IASTED International Conference on Internet, Multimedia Systems and Applications (IMSA'99)*, pp. 316–324. <http://www.caip.rutgers.edu/disciple/>.
- [Ward *et al.*, 2002] Ward, S., Terman, C., and Saif, U., Goal-Oriented System Semantics, MIT Laboratory for Computer Science, <http://o2s.lcs.mit.edu/> (2002).
- [Weiser, 1991] Weiser, M., 1991. The Computer for the 21st Century, *Scientific American* 265 (3), 94–104. <http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html>.
- [Weiser, 1993] Weiser, M., 1993. Some Computer Science Issues In Ubiquitous Computing, *Communications of the ACM* 36 (7), 75–84.
- [Weiser, 1996] Weiser, M., Ubiquitous Computing, <http://www.ubiq.com/hypertext/weiser/UbiHome.html> (1996).
- [Wilkhahn, 2002] Wilkhahn Homepage, Wilkhahn, <http://www.wilkhahn.de> (2002).
- [Winograd, 2001a] Winograd, T., 2001a. Interaction Spaces for Twenty-First-Century Computing. In: Carroll, J. M. (ed.), *Human-Computer Interaction in the New Millennium*, Addison Wesley, pp. 259–276. <http://hci.stanford.edu/~winograd/papers/21st/>.
- [Winograd, 2001b] Winograd, T., 2001b. Architectures for Context, *Human-Computer Interaction* 16 (2–4), 401–419.
- [Winograd and Guimbretière, 1999] Winograd, T. and Guimbretière, F., 1999. Visual Instruments for an Interactive Mural. In: *Proceeding of the CHI 99 conference on Human factors in computing systems (CHI'99) extended abstracts*, ACM Press, New York, NY, pp. 234–235. <http://graphics.stanford.edu/projects/iwork/papers/chi99>.
- [World Wide Web Consortium (W3C), 2003] Scalable Vector Graphics (SVG), XML Graphics for the Web, World Wide Web Consortium (W3C), <http://www.w3.org/Graphics/SVG/> (2003).
- [Wyckoff *et al.*, 1998] Wyckoff, P., McLaughry, S. W., Lehman, T. J., and Ford, D. A., 1998. T Spaces, *IBM Systems Journal* 37 (3), 454–474. <http://www.research.ibm.com/journal/sj/373/wyckoff.html>.
- [Yokote, 1992] Yokote, Y., 1992. The Apertos Reflective Operating System: The Concept and Its Implementation. In: *Conference proceedings on Object-oriented programming systems, languages, and applications (OOPSLA'92)*, ACM Press, New York, NY, pp. 414–434. <http://www.csl.sony.co.jp/person/ykt/ykt-oopsla92.html>.
- [Yokote *et al.*, 1994] Yokote, Y., Kiczales, G., and Lamping, J., 1994. Separation of concerns and operating systems for highly heterogeneous distributed computing. In: *Proceedings of the 6th workshop on ACM SIGOPS European workshop*, ACM Press, pp. 39–44. <http://doi.acm.org/10.1145/504390.504401>.
- [Zanella and Greenberg, 2001] Zanella, A. and Greenberg, S., 2001. Reducing Interference in Single Display Groupware through Transparency. In: Prinz, W., Jarke, M., Rogers, Y., Schmidt, K., and Wulf, V. (eds.), *Proceedings of the Seventh European Conference on Computer Supported Cooperative Work*, Kluwer Academic Publishers, Amsterdam, NL. <http://www.cpsc.ucalgary.ca/group/lab/papers/>.
- [Zweben *et al.*, 1995] Zweben, S. H., Edwards, S. H., Weide, B. W., and Hollingsworth, J. E., 1995. The Effects of Layering and Encapsulation on Software Development Cost and Quality, *IEEE Transactions on Software Engineering* 21 (3), 200–208.