Software architecture for social robots

Software Architecture for Social Robots

Martin Saerbeck

Software Architecture for Social Robots

Martin Saerbeck

The work described in this thesis has been carried out at the Philips Research Laboratories Eindhoven, the Netherlands, as part of the Philips Research Program.

Typeset with LaTeX 2$_\varepsilon$
Cover design by Christoph Bartneck

Proefontwerp Technische Universiteit Eindhoven.
ISBN: 978-90-5335-234-2
Trefw.: Social Robots, Interaction Design, Software Architecture

Software Architecture for Social Robots

PROEFONTWERP

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven,
op gezag van de rector magnificus, prof.dr.ir. C.J. van Duijn,
voor een commissie aangewezen door het College voor Promoties
in het openbaar te verdedigen op
donderdag 10 december 2009 om 14.00 uur

door

Martin Saerbeck

geboren te Hamm, Duitsland

De documentatie van het proefontwerp is goedgekeurd door de promotor:

prof.dr.ir. L.M.G. Feijs

Copromotoren:
dr. C. Bartneck
en
dr.ir. M.D. Janse

Acknowledgements

This study was supported by:

- Philips Research Laboratories
- Eindhoven University of Technology

I would like to thank the many people who contributed to realize this PhD. First of all, I would like to thank dr.ir. Albert van Breemen, who has invited me to join Philips Research and the iCat project. During the first two years, Albert has been my supervisor at Philips and guided me in the transition from an university to an industrial research environment. At this point, I also would like to mention the other members of the iCat team. Bernt Meerbeek and Peter Bingley supported me with their rich experience and practical advice in conducting experiments with iCat. Many of these experiments would not have been possible without the software engineering expertise of Dennis Taapken and Peter Jakobs, who have been the driving forces for the development of the OPPR framework. I also will miss the discussions with dr. Leszek Holenderski, who never got tired in sharing his deep insights and pragmatic solutions on numerous complex problems.
My special thanks go to the members of my reading committee, with name prof.dr. E.J. Krahmer, prof.dr. Matthias Rauterberg, dr.ir. B.J.A. Kröse and dr.ir. A. van Breemen, for reviewing my work and for their critical and constructive comments. In particular, I would like to express my sincere appreciation to my promoter prof.dr.ir. Loe Feijs, who guided me through the process of this thesis and supported me with numerous scientific advices and constructive feedback. Furthermore, my thanks extend to my co-promoters and daily supervisors dr. Christoph Bartneck and dr.ir. Maddy Janse, who were always available and supported me with their rich expertise and experience. Their valuable advices helped to keep my research on track.

I also would like to thank the members of the Connected Consumer Solutions group at Philips Research and the members of the Design Intelligence group at TU/e, for offering a welcoming and productive working environment and for the many invaluable comments on my research.
In particular, I would like to thank Matthias Krause and Nele Van den Ende, with whom I shared an office at Philips Research and who faced the same challenges in pursuing their PhD. Simply being able to share my worries with them has been more than once sufficient to solve them. The same holds for Sibrecht Bouwstra, Omar Mubin, Bram van der Vlist, Gerrit Niezen and Alex Juarez, with whom I shared an office on the campus of TU/e.
I also would like to thank the many other friends and colleagues who gave me a good balance between work and private life, especially Fabio Sebastiano, who endured the many German peculiarities during the past four years.
At last, I would like to thank those closest to me, my family, Angelika, Dieter and Thomas for their encouragements during frustrating times and for their confidence in me. Most importantly, I would like to thank Hendrikje, the love of my life, for her endless patience and unconditional support through the long periods of hard work.

Martin Saerbeck
December 2009, Eindhoven

Contents

List of Figures

List of Tables

Part I

Design and application of social robots

Chapter 1

Introduction

Digital technologies have become a prerequisite to participate in our digital society [68]. In addition to offering entertainment, digital technology facilitates relations between people, defines communication, provides access to private and public services and offers to participate in on-line communities and political discussions. While these ongoing developments offer substantial opportunities, they also pose increasing demands on users and application designers. The European commission estimates that between 30% - 40% of the population do not receive benefits of the information society also due to increased demands on 'digital literacy' of the users [68].

In particular, the increase in complexity and autonomy of systems and services steepens the learning curve. The increase in complexity can be attributed to devices that are increasingly 1) multi-purpose, 2) involve multi-user and 3) incorporate multiple devices. For example, contemporary mobile phones incorporate far more features besides the original functionality of phoning. In consequence, it becomes difficult for a user to assess which functions are supported by a device and how to access them. If multiple users are involved, such as in on-line communities or multi-user games this difficulty increases. In multi-user environments, it is often difficult to determine how to interact with other users and to identify which rules apply or how private information are treated. In the same manner, usage becomes more challenging if device functionality is distributed across device boundaries such as in ambient intelligent environments [249]. In the best case, interactions between devices are transparent to the user. However, practical experiences have shown that users often fail to understand the consequences of their actions.

Similarly to an increase in complexity, an increase in device autonomy elevates the demand on the user to instruct a device. In consequence, the application designer has to make a trade-off between the level of control offered to a user and ease-of-use. The more possibilities a user has to influence the device's behavior the more difficult it becomes to operate the device.

One of the fundamental problems of traditional technology driven design is that it demands users to develop an understanding of the underlying technology, instead of offering the user concepts that he is already familiar with [18, 218]. To this end, social interfaces have been proposed as a promising interface paradigm to achieve easy and natural interaction as they utilize people's natural abilities to interact with other social communication partners [72, 47, 59, 28]. People already possess social interaction capabilities, while technical interaction capabilities have to be learned explicitly. Social interaction techniques provide an adequate solution to optimally reuse existing capabilities [213, 148].

Additionally, it has been argued that people can not *not* interact socially and that therefore devices have to be socially designed to avoid misunderstandings and conflicting signals [244, 248]. Reeves and Nass have shown that people are naturally biased to treat and respond to media in the same way as they react to other humans [198]. Furthermore, people apply social rules even if they are consciously aware of dealing with a machine. A common explanation of this phenomenon is that anthropomorphism and social reasoning helps us to explain observations from our environment [59, 114, 152].

However, social interaction techniques might also raises wrong expectations which can leads to frustration, if these expectations can not be met [59]. Additionally, social expressions might simply be distracting from the real task. Picard has identified several challenges that have to be overcome before social and emotional computing techniques can be applied to user interfaces, including perception, selection of modalities, expression of emotions and ethical and social concerns [191].

In fact, social interactive interfaces only give the impression of being social and having emotions. However, as impressively demonstrated by the movie industry, convincing fabricated emotional expressions can evoke intense emotional experiences [231]. The basic idea underlying social user interfaces is to augment technical functionality in terms of human-like modalities. For example, a confirmation can be implemented in technical terms using a light signal or sound, or by using human-like modalities such as

speech or the indication of a nod. In order to realize such functionality, devices need to be able to express themselves and react to stimuli according to rules of human-human interaction.

Research on social interaction technology has mainly focused on virtual anthropomorphic characters, either screen based or with a robotic embodiment [72, 89, 193, 242, 9, 117, 120]. However, the application of social interaction is not limited to anthropomorphic interfaces. Also very abstract interfaces such as abstract movement patterns of light can be used for social interaction [172]. With modern robot technology also hardware implementations of expressive interfaces become possible.

A major challenge for the application of social interactive interfaces is how the available modalities are used within an application. In comparison to traditional robotic applications, the performance of an application depends on the richness of the behaviors, rather than on execution speed or precision. Furthermore, the concrete interaction depends to a large extend on the employed hardware and the application context, which poses additional challenges on the design process and the capabilities of designers. The topic of this thesis is to develop a design framework that supports the design process of social interactive interfaces.

1.1 Robot application design process

This technological design concerns the design process of social interactive robotic interfaces. The focus is set on the application design task in terms of software not on hardware, in particular on designing an application for a social robotic interface. An abstract design scenario is depicted in Fig. 1.1. In this example an application developer creates an application that utilizes a social interactive robotic interface. The user interacts with the interface using his natural interaction capabilities. In this context, natural interaction refers to interaction that the user is naturally capable of, without additional learning effort. The interface is a robotic platform for which the application designer creates an application. The goal of the design process is to use the robot's interaction capabilities to present a coherent interface to the user.

The major obstacle to realize this scenario is that currently no unified design framework exists that can bring the various roles together that are involved in the design process. Design challenges that are posed to the designer include developing the application theme, interfacing with the hardware,

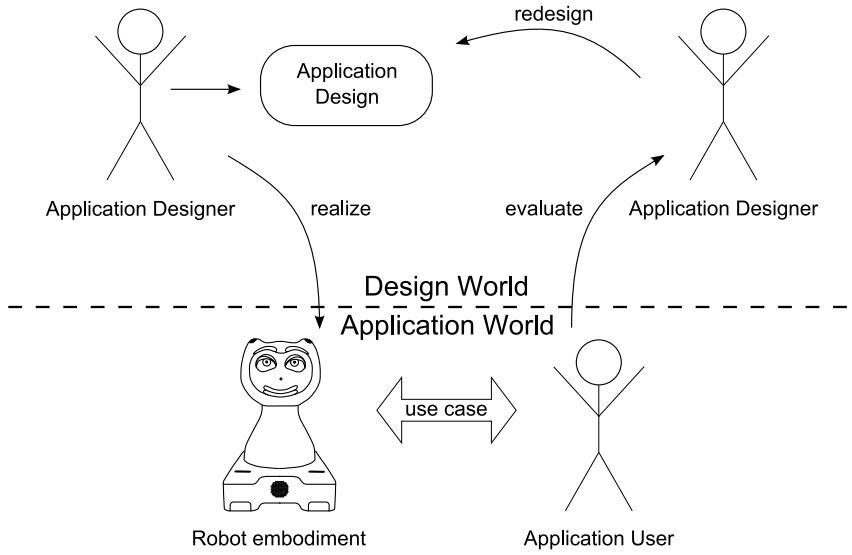


Figure 1.1: Design scenario: An application designer creates an application for a robotic user interface.

deciding on the interaction modalities and designing appropriate device behaviors. During this process, the designer has to keep several constraints in mind such as the expectations of the user, social interaction rules, expression and perception of emotion as well as reacting appropriately to the user's input.

The basic concept of a supporting design framework can be illustrated by analogy to available media design tools. Design suites like the Adobe Master® Collection[1] define a set of tools that support a designer to develop content for digital applications. For example, Adobe Photoshop® provides a specialized set of tools for drawing and editing images. The basic concept is a two dimensional canvas that can be filled with colored pixels. On top of this basic structure, plenty of pixel processing methods are defined, that alter the image in one way or the other. The resulting images can in return be the basic building blocks for a web application. For example, they can be used for customizing the appearance of a button. Adobe Fireworks® provides basic editing features and concepts for creating websites. All of these tools define a specific set operators that are relevant for a specific problem at a given level of abstraction. Additionally, they provide different views on the same design problem.

[1] http://www.adobe.com/products/creativesuite/mastercollection

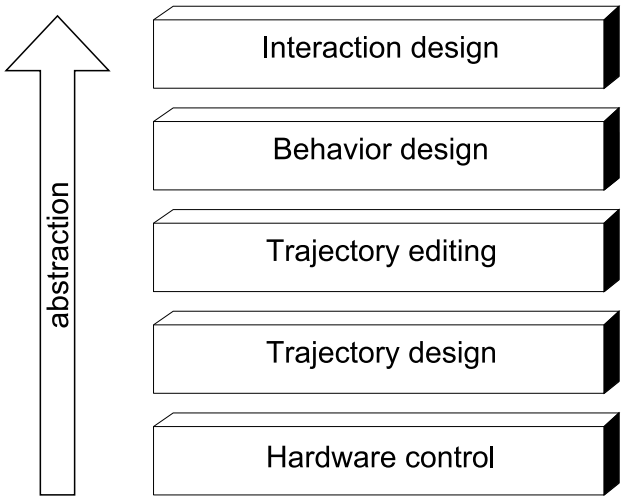


Figure 1.2: Levels of abstraction in the design of a social robotic interface.

The problem is that no such consistent set of tools is existing for the development of social interactive applications for robotic embodiments. An illustrative hierarchy of levels of abstraction for robotic behavior design is depicted in Fig. 1.2. The different levels of the hierarchy are discussed in the following.
On the lowest level, the designer has direct control over the hardware, including access to the motor control. Working on this level requires extensive technical skills from the designer. In analogy to the media design, it compares to controlling the display driver of the monitor.
On the next level of abstraction, the designer has a higher level of control over the actuators by defining trajectories over time. For example, the designer can define the trajectory of a robotic arm over time. This equates to drawing tool that allows the designer to color individual pixels.
On trajectory editing level, the designer modifies or combines trajectories using higher level operators such as blending or filtering[30, 130]. This level already provides tools to generate very expressive behaviors as commonly applied in designing traditional animations for movies [253, 231, 200, 131]. In media design, this is realized by specialized brushes or filters that allow to modify an existing image. With editing methods, the designer can create very expressive pieces of behavior. However, these behaviors are very specific to a particular situation. For example, an expressive grasping action fails, if the object is placed slightly different than at the time when the behavior was defined. High level control is necessary to direct and modify

expressive behaviors. In comparison, automatic layout mechanisms allow that screen widgets change layout and behavior.
Ultimately, on the topmost level, the designer reaches the level of a movie script that describes roles and interactions on the levels of personality and general interaction rules.
These different levels of abstraction illustrate the complexity of creating interactive applications. However, currently most of the applications for robot applications are created on a hardware control or trajectory design level.
In order to increase this level of abstraction, this technological design aims to create a supporting framework and required tools that enable high level application development and therefore enable social interaction.

1.2 Research questions

This technological design proposes a design suite for interactive robot applications. In particular, a general design architecture is developed that identifies the basic concepts for a software architecture, which integrates different design tools. In summary, this thesis poses the following four research questions:

① **Design challenges:** What are the design challenges in the process to develop an application for a social robotic interface?

② **Tools:** How can the design process be supported by design tools?

③ **Requirements:** What are the requirements for a software architecture that unifies the design process?

④ **Architecture:** What architecture fulfills the requirements for a robot application design framework?

1.3 Guide through the thesis

This thesis consists of three major parts. In the first part the design challenges of the application design task are investigated. The second part derives requirements and develops a unified software architecture and tools to support the design task which is evaluated and validated in case studies in the third part.
To this end, this thesis first discusses in Chapter 2 current social interactive interfaces and their application domains. In Chapter 3 various design

challenges of the development process are analyzed. Chapter 4 introduces animation technology as an enabling technology for expressive interfaces. Chapter 5 analyzes the software design domain and identifies the major requirements for a software architecture. Based on this analysis, an overall high level architecture is developed that introduces two separate environments: a 'Development-Environment' and an 'Execution-Environment'. The architecture of the Development-Environment is developed in Chapter 6 and the architecture of the Execution-Environment is developed in Chapter 7. The following chapters evaluate and apply the architecture. First of all, Chapter 8 evaluates the functional and non-functional attributes of the architecture. A particular challenge for the evaluation of a design framework is to assess the quality of the designed artifacts, because these depend on the capabilities of a designer as well as on the quality of the provided tools. Therefore, the architecture is evaluated in case studies in which the framework has been used to develop applications using social interaction artifacts. One of which applies the framework for a research project to investigate how robot motion is perceived in terms of affective content. The results are reported in Chapter 9. Finally, this documentation concludes with a general discussion on the architecture in Chapter 10.

Chapter 2

Expressive robotic interfaces

Social interfaces aim to utilize peoples' natural interaction capabilities [72, 89]. In this context, this technological design focuses on social robotic interfaces, in which the robot serves as the interface for an application. Bartneck and Forlizzi define a social robot as:

> A social robot is an autonomous or semi-autonomous robot that interacts and communicates with humans by following the behavioral norms expected by the people with whom the robot is intended to interact. (Bartneck and Forlizzi [10] p. 592)

In the literature, social robotic interfaces have been analyzed from a variety of disciplines, including computer science, robotics, psychology and social sciences. These disciplines define different terminology for the analyzed constructs. In the following the key terminology that is used throughout this thesis is introduced.

2.1 Terminology

Natural interaction Throughout this work the term 'natural interaction' refers to interaction that uses modalities that are understandable for a user without prior training. Natural interaction depends on the 'perceived affordance' of an object [176]. Therefore, the definition for natural interaction depends on the target group. Cultural background, educational level, age group or gender can have an influence on what types of interactions are 'natural' for a particular audience.

Application An application describes a set of services that are offered by a device to a user. This definition is most closely related to the usage

of the word in the field of computer science, in which the term refers to a computer program that enables users to use a computer for a specific task. A concrete application defines the interaction context in which the user interacts with the device.

Social robotic interface The term 'social robotic interface' refers to a robot that serves as an interface in a given application. For communicating with a user, the robot uses human-like interaction modalities including speech, gestures and emotional expressions [213, 9].

Believability 'Believability' denotes a degree of convincingness and consistency in the device behavior [178]. The term believability was first introduced by Bates in 1994 in the context of designing intelligent, engaging and life-like characters [17]. In his definition, the ability to express emotions takes a central role for being life-like. Mateas added to this definition the concept of 'drama' and 'story' of a character [158]. Dautenhahn stresses that for being believable, a character does not necessarily have to exhibit intelligent or realistic behavior [45]. Instead, following principles of animation, the audience must be convinced of dealing with a life-like character with own drives and needs [231, 113, 112].

Animacy The term 'animacy' describes the perception of life-like characteristics. It has been shown, that people develop very early a capability to distinguish animate from inanimate objects [194]. One explanation suggests that people are able to detect energy violations of self propelled objects [215]. Whenever people perceive a change of the current motion that is not caused by external events following the Newtonian laws, these objects are perceived as animated. Another model for interpretation of life-like characteristics is that social reasoning helps to make sense of an observation [59, 45]. The perception of animacy appears to be a prerequisite for successful social interaction.

Animation The term 'animation' denotes a sequence of actions of a character [231]. An animation is fixed in a sense that it cannot be dynamically modified during execution of the actions. Most commonly, animations are implemented using look-up tables, which contain the states of actuators over the duration of an animation.

Behavior A behavior is like an animation a timed set of actions, but with the difference that a behavior may define reactions to sensor inputs. Behaviors are defined by sets of equations that relate the state of actuators

to input parameters for the behavior [237]. Most commonly, behaviors are implemented using scripting technology.

Designer This technological design targets at designers of applications for social robotic interfaces. The tools and architectures that are presented aim to support the designer.

User The term 'user' refers to the users of the applications that have been created by designers with the tools developed throughout this thesis. The term is not to be confused with the role of designers, who in fact are users of these software tools. As depicted in Fig. 1.1 on page 6, the user interacts with a robotic interface, while the designer interacts with the design tools that are presented in this thesis.

2.2 Social interaction with robots

When users start to attribute human-like characteristics to a device, they naturally start to interact with it in a social way [198, 46]. In consequence, they use natural communication channels like speech, gestures or emotions [169]. The example of how people interact with pets demonstrate that users are using speech even if they know that the communication partner can not understand the semantics of the utterances. A particular strong bond is created if the user gets the impression that the communication partner feels emotions [50, 76, 190].
From a design perspective, social interaction provides design guidelines for designing interaction sequences between user and device [129]. In the literature, several high level interaction patterns have been identified that reoccur in human-human interaction [124]. For example, during first encounter, a mutual frame of reference between two communication partners is established. During an interaction, subtle cues are used for guiding an interaction in terms of turn taking or to give feedback to the communication partner. The design challenge is that social interaction defines a mutual relationship. The appropriateness of the device's behavior is crucial for establishing satisfactory social interactions [191, 87]. This does not mean, however, that the robot needs full human-like social interaction capabilities. Reeves and Nass have shown that people are naturally biased towards social interaction and even apply social rules when consciously aware of dealing with a machine [198]. Using social reasoning, people place themselves in the role of the interaction partner [45, 103]. For example, a mobile

robot can express its battery level in terms of driving speed. If the robot becomes slow, the user might reason that the robot is tired and needs to rest. During resting at the home base, the robot can recharge.

Nevertheless, most research on robotics is focused on algorithms (such as path planning or face recognition) and is hardly concerned with the effects that these algorithms have on how the robot is perceived [219]. Applying complex algorithms does not necessarily result in the perception of intelligent behavior or in producing a pleasant interaction [113]. People are very sensitive to unnatural behavior [231]. In this sense, the development of social robotic interfaces poses a greater risk, because a subtle design flaw might not only destroy the impression of a life-like character, but will leave a negative impression of the device in general [170]. Positive examples of long term engaging robots can be found in the movie business. The robots R2D2 and C3PO from Star Wars™ are engaging for hours, even though there is no a direct interaction between the audience and the film characters. These robots express emotions only based on movement and sound. Another example is the usage of remote controlled robots in attraction parks such as Disney's® Lucky the Dinosaur [57].

Social interaction technology can increase engagement of the user. The user as well as the device take active roles in the interaction. This has for example been captured by Nakatsu et al., who have presented a framework which explicitly introduces robots as potential interface for interactive media [173]. Even more importantly, it serves as a crucial enabler for a variety of application areas, including interfaces to control ambient intelligence environments [9], robots in health and elderly care [102], or in teaching and learning environments [146]. It offers possibilities to introduce technological devices to areas where it was formerly difficult to find acceptance due to technical shortcomings or imposed learning effort on the user [133, 102]. For example, social robotic interfaces could provide high level control over common digital devices also for people without technical skills [68]. They allow people to use their natural interaction capabilities instead of demanding technical skills. Additionally, people tend to be more forgiving if they have an emotional bond with the device [214]. In the above mentioned Star Wars™ movie, the users of the translation device C3PO were not frustrated about how often the system was broken. Instead, the audience felt sorry for the device and many wanted to help him through difficult situations.

One of the major challenges for social interaction technology is to maintain the illusion of dealing with a real character also over a longer period of time

[87]. On first encounter, it depends to a great extend on the embodiment and first movements if a device appears life-like. It is rather easy to establish the impression of a life-like character at the beginning of an interaction [235]. Later on during the interaction, it also depends on functionality, expressiveness and how the character reacts to stimuli from the environment. For example, repetitive behaviors are attributed to machines rather than life-like characters. Therefore, it is not sufficient to simply repeat expressive animations.
The above discussion shows that social interaction technology introduces a new set of social requirements for the application design task. Currently, the designer is left without a conclusive framework for designing interactive behavior of social interactive interfaces. The next section discusses several potential hardware platforms that have been conceived for direct interaction with the user and would benefit from social interaction technology.

2.3 Personal robots

Multiple expressive robots have been developed. In this section a few of these robots are selected to demonstrate generic concepts. The most defining and constraining factor for a robot is obviously its physical embodiment. In the literature, screen based simulations of physical robots have been proposed as one possible replacement of physical hardware that enable similar types of interactions, but remove hardware maintenance throughout the development [162, 72]. However, it is still not clear how the embodiment influences the interaction, for example in terms of acceptance as a social interaction partner or the perception of emotional expressions [13, 91, 52, 88]. For example Kiesler et al. investigated the effect of embodiment by comparing the reaction to a screen based character, a video transmission of a real robot and a collocated physical robot. They found that the embodiment had an influence on how much private information participants shared [193].
Nevertheless, from an interface perspective, virtual representations of physical robots have the same number of degrees of freedom and are able to perform similar actions. They can therefore be treated similar in terms of software design. A general overview and classification scheme of common robot hardware in terms of available actuator types and control models is presented in [219]. The most common actuator for controlling motion are servo motors, as employed by many of the following hardware platforms.

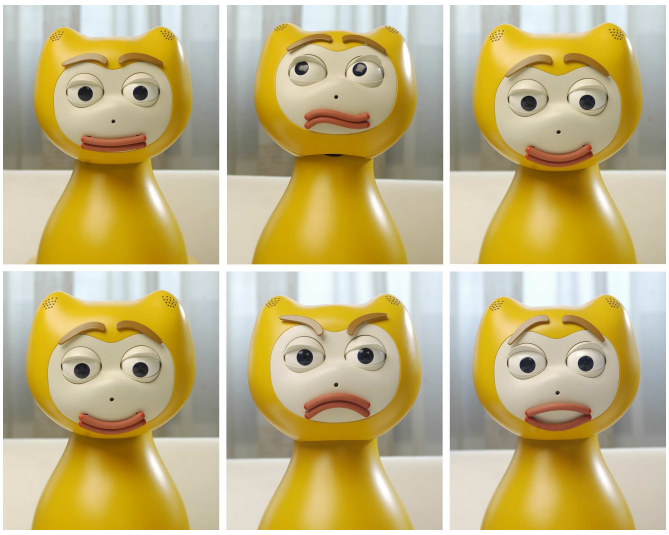


Figure 2.1: The Philips iCat robotic research platform is able to express basic facial expressions

Philips iCat The 'interactive Cat' (iCat) robot is a stationary robotic research platform to be placed on a desk. It is developed by Philips Research to research human-machine interaction modalities [242, 238]. The robot is depicted in Fig. 2.1. The iCat robot has the physical shape of a cat with a mechanical rendered face and is approximately 40 cm high. With 13 degrees of freedom to animate parts of the head it is able to express basic facial expression and emotions (see Fig. 2.1). Additionally, iCat has RGB LEDs located in the ears and paws and a speaker in the base. Therefore, iCat is able to express itself in the modalities light, sound and motion, which allows for a rich user interaction. Light can be a very expressive medium that appears natural for interaction, even though it is not one of peoples' natural modalities. For example, iCat is able to show a red light in the ears in case an error has occurred. This message can be enhanced if at the same time iCat displays a sad face, looks down, shakes the head and plays a sad audio fragment. For sensing the environment, iCat has a camera located in the nose, two microphones located in the paws and four touch sensors at the ears and paws. Furthermore, it contains an infrared distance sensor in the left paw to sense an interaction partner.

The main purpose of iCat is to serve as a research vehicle for social interaction technology. Through its human-like expressions it is able to trigger peoples' social interaction capabilities with the goal to bridge between the

Figure 2.2: iRobot robotic vacuum cleaner Roomba (Version SE 5210)

digital world of devices and the user. A variety of applications has been developed with iCat, including a game-buddy [143], TV assistant [161] and waiter application [208].

Roomba Roomba is a domestic vacuum cleaning robot developed by iRobot. It has a circular geometric shape with a radius of approximately 15cm and is about 7cm high (see Fig. 2.2). Roomba has a velocity controlled differential drive system consisting out of two motors. A detailed overview of the drive system of Roomba can be found in Chapter 7.4.

In its basic form, Roomba is designed as a random walker that offers three modes of application: 1) Normal cleaning mode, 2) Spot cleaning and 3) Large area cleaning. The differential drive system allows for great driving flexibility, including turning on the spot. Roomba does not have special degrees of freedom to express emotions. However, Forlizzi found that users perceive rich emotions and personalities in the robot [74].

Its architecture is based on Brook's subsumption architecture [29]. Therefore, Roomba does not possess global path planning, but seemingly organized behaviors emerge from a set of simple behaviors.

PARO PARO is a personal robot, with the physical shape of a seal, which is developed for therapeutic applications [217]. The robot is depicted in Fig. 2.3. Due to its zoomorphic features, PARO has calming and relaxing influence on patients. Its life-like embodiment, including the fur, supports the perception of dealing with a life-like character [245]. Participants build up an emotional relationship to the robot, treating it like an animal. PARO

Figure 2.3: PARO robotic seal for therapeutic applications (Source: http://www.aist.go.jp)

does not use speech, but interacts by means of sound and movement patterns. People can engage with the robot by physical interaction such as caressing it.

Care-O-Bot 3 Care-O-Bot® 3 is the third version of the domestic service robot developed by Fraunhofer institute [100]. The robot is depicted in Fig. 2.4.
Care-O-Bot 3 is intended to support independent living at home by providing physical services such as fetch-and-carry tasks as well as walking assistant. Furthermore, it serves as central control over devices in the home and provides connectivity to reach relatives and health-care services. It has an omni directional drive system that allows the robot to navigate in space constraint personal homes. The main sensors are a 3D depth-image stereo camera and a laser scanner.
Care-O-Bot 3 engages in rich interactions and collaborations with the user. It is equipped with a mechanical arm with seven degrees of freedom that allows to hand objects over to the user. Furthermore, it contains a touch screen display for menu based interaction. Care-O-Bot 3 is also able to react to simple gestures and movement sequences. However, it does not have special degrees of freedom to display emotions. Nevertheless, it resembles basic human features with a head and arm.

Geminoid Geminoid is a realistic copy of a human [175]. Its main purpose it to serve as a general human-machine interface. It was argued that due to its human-like shape, humans can interact with the robot the same

Figure 2.4: Fraunhofer Care-O-Bot® 3 (Source: [92])

Figure 2.5: Geminoid robot. (Source: [13])

Figure 2.6: Lego®Minstroms v.2 (Source: http://www.lego.com)

way they interact with other humans [114]. To this end, the robot has similar degrees of freedom as a human. However, it does not have the ability to walk and is therefore bound to a chair. The robot is depicted in Fig. 2.5. Geminoid is mainly remote controlled for interaction in Wizard-of-Oz type of experiments. It does not possess autonomy like Roomba or Care-O-Bot 3, but it aims to offer an interface that is entirely familiar to a human user. A user may use the exact same interaction techniques as when dealing with other people.

Lego® Mindstorms The Lego® Mindstorms kit[1] is an educational robotic development kit that consists of multiple sensors and actuators that can be connected to a central unit. Robots can be built in the typical Lego way, by assembling Lego blocks. An example for the types of robots that can be created with the kit is depicted in Fig. 2.6.
The final robot can be programmed using a specialized Lego Mindstorms software that is derived from LabView. LabView offers a graphical editor that allows to connect multiple predefined building blocks following a program flow metaphor. That is, the building blocks are arranged along lines of execution. A line of execution defines a sequence of actions. Multiple lines can be executed in parallel, which makes it especially easy also to define parallel paths of execution. The Mindsotrms kit is positioned as an educational kit that helps to learn about robots, develop creativity, or just to play and have fun.

[1] http://mindstorms.lego.com

Chapter 3

Design challenges

The development of social robotic interfaces poses new design challenges to the developer. In this chapter several of these challenges are analyzed in detail and presented at three different levels of abstraction. The topmost level concerns the overall application design. The next lower level is concerned with the interface options for interaction design. The lowest level discusses the concrete behavior design of socially interactive interfaces.
The design challenges that deal with the design of appearance for the robot are not taken into consideration. Appearance has received plenty of attention in literature [14, 13, 91, 52, 125, 129, 243, 21, 151, 88].
For the development process, a designer may choose from different design approaches. Usually, an iterative and user-centered design approach is chosen [18, 127]. A particular design approach for the development of a social robotic interface is to design along a personality. For example, the designer might adopt an iterative design approach as described by Meerbeek et al. [162]. A strength of this approach is that it combines best practices from technology driven, artistic and user-centered design.

3.1 Application design challenge

This section analyzes the design process for an application using a robotic user interface and argues that a balanced design is a crucial factor for the success of a particular application. It is shown that an application can fail, because of an unbalanced design, which consequently nullifies the efforts made to apply an expressive character in the interface. The study was presented at the International Conference of Social Robotics (ICSR09) [209].

Abstract – People are striving for easy, natural interfaces. Robotic user interfaces aim at providing this kind of interface by using human like interaction modalities. However, many applications fail, not because of fundamental problems of addressing social interaction but due to an unbalanced design. In this paper we derive a balancing framework for designing robotic user interfaces that balances four key dimensions: user, application, interface and technology. We investigate applicability of the framework by means of two experiments. The first experiment demonstrates that violations to the balancing framework can negate the efforts to improve an interface with natural interaction modalities. In the second experiment we present a real world application that adheres to the balancing concepts. Our results show that a balanced design is a key factor for the success or failure of a given robotic interface.

3.1.1 Introduction

The observation by Ben Shneiderman [218] that computer users waste an average of 5.1 hours per week trying to use computers has a serious implication for the design of user interfaces. Low prices for hard- and software will make technology available to a larger number of people, while at the same time new technological achievements introduce new functionality that make the devices more and more complex. In order to enable people to use all the functionality there are two main approaches: educate the users in operating the devices or make the devices easier to handle. Shneiderman calls it "bridging the gap between what users know and what they need to know". Most of the time the first approach is taken, resulting in big booklets accompanying the various devices, but people are striving for easier, more natural interfaces. Robotic User Interfaces (RUI) are aiming in providing exactly that kind of interface [72].
In this study we analyze the design process of such RUIs and argue that a balanced design is a crucial factor for the success of a particular application. We translate a design framework originating from the theory of interactive systems to the design of RUIs. The goal is to investigate the applicability of a such a framework and to use it as a success predictor of a certain application during the design process. Besides the fact that we are designing an interactive interface that meets functional, physiognomic and cultural requirements we strive to create a believable, social accepted communication partner. Therefore, we need a research environment that allows us to

test interaction paradigms in a real world setting. Designers of RUIs deal with very complex evolutionary evolved social interaction abilities that are not yet fully understood. Subtle flaws in the design might negate all the efforts to make the interface more usable. Incorporating social interaction modalities in an interface requires a high amount of balancing intuition of the designer. We demonstrate the importance of a balanced design with an experimental setup that violates these principles.

3.1.2 Balancing framework

The field of interactive systems suggests that the key dimensions to balance a RUI design are: user, technology, interface and application [18]. In the following these four dimensions are explained in detail:

User The purpose of using a robotic user interfaces is to allow everyone to use it without additional learning effort. This does not mean that there is one universal RUI but that the elements for an RUI have to be carefully chosen to accomplish its service. The design has to take into account cultural differences, social constraints as well as short term moods of the user.

Application Application areas for robotic user interfaces are manifold. Ultimately, robots could serve as interface between a user and any electronic device, bridging the gap between the physical and the digital world. For example, application areas also includes entertainment, in which the user is interacting with the device just for the fun of interacting with it.

Interface Designing a robotic user interface means to choose the modalities and define interaction methods. The most common input modalities are vision, speech, keyboard, touch sensors and switches. For giving feedback to the user the robot can communicate by means of light, sound and motion. With facial expressions and body postures the robot is able to convey emotional messages. In order to define the interface, the modality as well as the protocol of messages have to be specified.

Technology Most of the constraints for the design of a RUI are technological. Processing power and memory become less of a factor because of technological achievements, while at the same time dropping in

price. Additionally, wireless communication enables remote processing and access to worldwide knowledge. The same trend as for processing power can be observed in the price for sensors and actuators. Currently, the more limiting factors are available techniques for data interpretation and generating appropriate behavior. Artificial Intelligence algorithms for interpreting speech and vision data are still not able to extract a sufficient percentage of information encoded in the signals to ensure a seamless interaction. Also new techniques have to be developed to enable robotic characters to become a fully accepted social communication partner.

All four dimensions have to be in balance to create an effective RUI. Very often their constraints compete with each other. In the following section we explain the balancing process.

3.1.3 Balance the dimensions for an RUI

In our analysis of balancing the RUI we start with the user. The user has to be interested in the task proposed by the robot. This is a prerequisite that always has to be met and already puts the first constraint on the application. The designer has to find a match between the application and the user. This relation seems very obvious, but the designer has to keep it in mind and account for it during the design and implementation process. Adapting the application to the user might not always be possible, because it is constrained by technological limitations. Very often additional research is needed to find new solutions that enable the desired applications and functions. In consequence, it is only to a certain extent possible to adapt the application to the user.
As soon as the initial application is defined we incorporate the interface. As concluded by Dautenhahn [46] there is no use of anthropomorphizing an interface if the aim of the application is to perform repetitive actions, which do not need a deep emotional content such as, for example, operating a washing machine. In this case efficiency and swiftness are more important features. That means the application requirements need to reflect the degree of emotional expressivity that is needed to involve the user.
By incorporating anthropomorphic artifacts in the user interface, the designer has to define needed functionality of the RUI and match them with shape and behavior of the robotic embodiment. The results of Goets [88] shows that people judge the capabilities of a robot by shape and behavior and are willing to interact if they match each other. If shape and behavior

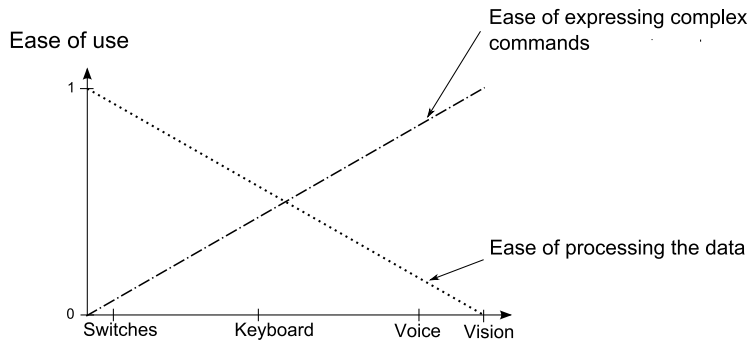


Figure 3.1: Antagonism between rich input and computer understanding

raise expectations that cannot be met it will lead to frustration of the users, who might get the impression that it is not able to perform the task.

For designing the interface the designer has to assess how much effort is imposed on the user. For example, spoken language (speech) is a streaming medium and poses a higher load on people's short-term and working memory than non-streaming media like images. Shneiderman explains that spoken language and problem solving share the same "resources" in the brain, and therefore the short-term and working memory load is much higher when a speech interface is chosen [218]. The interface should not put an additional burden on the user. Considering these facts, the interface might give new constraints to the application that have to be taken into account for application design.

At last we balance the system with the available technology. Because we have already specified demands of the application and the interface we can specify the requirements for the technology very precise. Of course it can happen that the requirements cannot be met. For example, if the user demands a natural input for easily expressing his emotions that is beyond the capabilities of the current technology. The richer the input, the less reliable are current technologies to interpret the data. This antagonism is qualitatively illustrated in Fig. 3.1 for a chitchat application. The X-axis represents a scale of input from technical to human-like modalities. On the Y-axis 2 different data are displayed, related to ease of use. Both are theoretical ratios: the first one represents the ease of expressing complex commands while the second one represents the ease of computationally processing the commands. A switch is a very simple input device but is also limited in its expressiveness.

In the balancing process we propagate constraints backwards and reiterate the last step if we encounter a constraint mismatch. If in the end it turns out that with the given constraints the user is not interested in the application any more, we have to revise it.
In the following we present two application design scenarios: The first violates the balancing framework and the second adheres to it. We argue that the balancing framework can be used as a predictor for the success of an application.

3.1.4 Case studies

In our study we focus on the iCat research platform, developed by Philips research [189] and made available to universities as a common research platform. Several studies have shown that iCat is able to express recognizable emotional expressions (e.g., [95]), while avoiding the uncanny valley [170] by its intentionally comic style appearance. We fix the shape of the interface, but vary the technology, the application and the type of interaction.
The first design scenario is a simple conversation bot application. We used straightforward design decisions to create the application and observed the effects of adding facial emotional expressions as feedback to the user in the interface. We expected the emotional feedback to increase the believability of the chatter bot as a character and hence improve the overall appreciation of the interface. As a second experiment we developed an application adhering to the balancing concepts. With the second experiment we demonstrate a successful balanced application incorporating a RUI. This is not trivial, because as soon as we address natural and social interaction capabilities of the user we have to satisfy user expectation for social interaction.

Conversation bot

With the conversation bot experiment we explore the design of an application for an RUI. The application provides us with an easy example scenario, which is already biased towards human-like interaction, hence for using an RUI. The hypothesis is that people will prefer an interface that incorporates more human-like artifacts in the interaction over machine-like artifacts. We test this hypothesis by comparing two different interface designs for the application. The in the following presented studies are mainly designed to test the balancing framework. Therefore, the number of participants have been limited to be sufficient to assess the application design rather than providing data for a full statistic analysis.

Scenario & technology In the chatter bot scenario iCat takes the role of a chat friend entertaining and serving the user. iCat listens to the comments of the user and shows emotional involvement with facial expressions. iCat has two fields of expertise, music and robots, which define the domain for the application. The user may ask for information such as newest publications from his favorite band and talks about his music taste. Therefore, one potential application domain of the chatter bot is to provide a natural interface for specifying preferences. Even if the main initiative resides on the side of the user, an active listener alone can already improve the efficiency of the application [134]. In the experiment iCat also shows its own musical taste that supports the impression of dealing with a believable character. The user would also be able to switch between topics, which is currently one of the major challenges for conversation bot technologies, but common in human style conversations.

Given the available technology, the best input medium for our test is a keyboard for the user. People are used to chatting systems, especially young people who grew up with computers and used chat systems in Internet cafes, on websites or with separate messaging tools. Therefore, one criterion for the selection of our subjects is that they are familiar with chatting technology.

Study design For the experiment of testing the design of the conversation bot application we chose for a wizard of Oz setup. The parameter that we manipulated during the experiment is the emotional feedback of iCat and the wizard of Oz design insured rational answers and appropriate emotional feedback while still presenting the original input interface to the user. We created two designs for the application: an experimental design in which iCat showed facial expressions and a control condition in which iCat had a neutral face. More emotional feedback should give the user the feeling that the conversation bot understands and cares about him. Every subject was shown both conditions in a randomly assigned order. Due to the exploratory nature of our study the sample size was limited to sixteen, all highly educated people, interns of Philips Research, in their twenties. They were selected for their frequent use of instant messaging systems, so that they fulfilled our requirement of being familiar with chatting technology using keyboard and computer screen. For the final test setup we chose for practical reasons a virtual embodiment of iCat. We are aware that there is a difference in the perception of a physical and a virtual embodiment [193], but we considered the differences not to influence our results if

Table 3.1: User satisfaction on conversation bot (regardless of the session)

Usefulness

	Not at all	Slightly	Moderately	Quite	Extremely	Average
Useful	0	5	5	5	1	3.125
Practical	1	3	6	4	2	3.5625
Functional	2	5	4	4	1	2.8125
Helpful	3	2	7	2	2	2.875
Efficient	1	6	4	4	1	2.875

Pleasure

	Not at all	Slightly	Moderately	Quite	Extremely	Average
Exciting	0	3	3	5	5	3.75
Fun	0	0	2	6	8	4.375
Amusing	0	0	3	7	6	4.1875
Thrilling	1	5	7	2	1	2.8125
Cheerful	1	2	1	11	1	3.5625

The average value is based on a 1 (not at all) to 5 (extremely) notation

we use the same embodiment in both conditions. Additionally, Bartneck et al. found that in terms of capabilities to express emotions, physical robots and virtual robots appear to be similar [15] that

User test results

We measured the perceived usefulness and pleasure the participants experienced while interacting with iCat. The results of the questions relating to those measures are shown in table 3.1.
The given positive feedback has to be taken with care. Due to the short interaction time no statement on the long term satisfaction can be made. Positive reactions motivated by surprise stem from novelty, which is lost in long term interactions. The usefulness was rated less high. During the semi structured interview participants expressed their doubts on the usefulness of the application. We have to keep in mind that all participants were technically skilled and interested in technology with proper background knowledge on current conversation bot technology. Despite their doubts people liked the idea of having a chat buddy to talk about music. In summary it can be stated that the additional emotional feedback did not improve the

Figure 3.2: Robotic waiter scenario in a restaurant environment

interaction. Less attention was given to iCat's facial expressions, because the user was using a keyboard. While using speech as input could improve this situation, the technology is not mature enough for realizing the conversation bot application. We can thus conclude that the conversation bot application is not well balanced due to mismatches between combination of modalities and available technologies. For example, there is a mismatch between the choice for using natural language and offering human-like emotions, but restricting the input to a keyboard.

Waiter application

With a second experiment we addressed the question whether it is possible at all to design a well balanced RUI given the current state of technology. Therefore, we designed another application as a use case to study the balancing framework. A video prototype of this application was first presented at the HRI conference 2007 [208].

Scenario To find a balance between the four dimensions - user, application, technology and interface - one needs to find a scenario that satisfies all of them. We chose to elaborate a waiter application because of three application conditions: limited interaction time, controllable environment, and narrow domain.

The scenario for the waiter application takes place in a restaurant environment. iCat is located on a table in the restaurant as illustrated in Fig. 3.2. When customers enter the restaurant iCat will greet them and offer them a free seat at her table. iCat will take their orders and submit them to an interface in the kitchen. The touch sensors in iCat's paws were used for simple yes and no answers. This input method proved to be very robust and also kept the user from anthropomorphizing too much. In our setting iCat would perform the following tasks: invite customers to the table, offer and explain dishes from the menu card, take orders, entertain customers while waiting, serve as an interface for controlling the environment such as lighting or music.

Balancing the application design We considered a RUI to be an appropriate interface, because welcoming and accommodating the customer with individual care is an emotional task rather than a functional, neutral one. A robotic user interface is in position to provide such emotional feedback. The application scenario further supports the use of social interaction technology. First, it is very difficult to maintain the impression of dealing with a social intelligent character over a longer period of time. Existing methods fail after a certain period, due to inefficiencies of the technology to create believable non-repetitive behavior. After that the user starts to recognize machinelike behavior. The chosen scenario lasts only for the time needed to complete dinner. Therefore, it has much less requirements to the interaction consistency than in applications in which the conversation bot that is continuously observed by the user over long periods of time. Second, the environment of a restaurant is much more controllable then a personal home. The position of iCat is fixed and customers usually sit around the table. Also the lighting conditions are predictable to a certain extent. Additionally, various sensors can be placed in the environment that enhances the perception of iCat. One might think of additional microphones, cameras or sensors to detect if a person is sitting on a chair. Third, the domain for the conversation is narrow. Explanations of particular dishes can be predefined as well as some typical sentences common in this type of scenario such as "I hope you liked the soup!". Of course, we have to keep in mind not to raise too many expectations by the behavior.

One of the main tasks for the system is to take orders from the customers. As stated above, just relying on a speech recognition system is currently not sufficient, especially in a noisy environment such as a restaurant. Our conversation bot experiment showed that offering a keyboard for natural

language input would be also the wrong choice. Instead, the restaurant can offer an "electronic menu card", so that the user just needs to point at a dish he wants to order or get additional information. In a situation like this it is realistic to assume that customers want to get an explanation of a specific dish. In this case, they point at the menu card and ask the waiter for his explanation.
Still we need a technique for choosing iCat's actions. Our requirements are that the system should be able to select discrete action depending on the state in a continuous domain. Therefore, we chose for Extended Behavior Networks ([155], [56]).

User test results To evaluate our design we performed an experts test with two restaurant managers. Before starting the test we asked them what they expect from the application. It turned out that their main concern was the functionality of the system, whether it is able to take orders. Their second concern was if it will bring added value to the restaurant.
After these initial questions we let them interact with our prototype from a user's point of view. They knew already the setup from the video prototype that we took earlier in one of their restaurants. In this version, we also included a first prototype of an electronic menu card from which they were able to select dishes. For reasons of simplicity and available resources we restricted the menu card to simple pointing gestures and used a bar-code reader to point at a specifically prepared menu card. An order was finalized and confirmed using the touch sensors in iCat's paws.
We conducted a semi-structured interview, focusing on three points: (1) Applicability of the application, (2) Interface design and (3) Personality of the waiter. Overall, they were positive about the interaction and the efficiency of the robot. At first we analyzed the application and its functionality. They were surprised by the speed with which the customers are able to get information and order a dish. They predicted it will be a great benefit for the restaurant. The process of collecting and ordering dishes worked seamlessly, though the bar-code reader seemed a little old-fashioned. They would prefer to have a touch screen for their restaurants.
Next we talked about their impressions on the interface. At the beginning the experts had some doubts interacting with a robot, but while testing, they developed a strong drive towards iCat, which they expressed by appreciating the actions of iCat like the accomplishments from a person. They only would exchange the embodiment to fit to the concept of the restaurant. The only thing they rated negative was that iCat did not support

their choice of dishes, a behavior that we did not include in our prototype. Especially at the beginning iCat should be much more proactive, offering help on how to use the service and propose dishes or drinks.
The last focus point was iCat's personality. The most important thing to mention is that they accepted iCat as a communication partner. They expected iCat to provoke conversation between customers. Both managers attributed iCat to have a social character, but they diverged in their opinion which characteristic would fit better in their restaurants. The first wanted iCat to provide additional services, e.g., to read out on-line news services or weather information while the other preferred a more passive iCat not too overwhelming or disturbing for the customer.

3.1.5 Discussion: Balancing framework

The results of our experiments emphasize the importance of employing a balanced design for utilizing a social robotic interface. Some major violations to the balancing framework kept the conversation bot application from succeeding. First, the choice for a keyboard input was purely technology driven, violating the demands of the interface and the user. It only adhered to the demands of the application to provide the possibility of unconstrained textual input. Second, the emotional feedback given to the user was not perceived, because of the faulty configuration of the interface. There is a mismatch in using natural language as means for communication but offering a keyboard for input. One might argue that adding a method to make the user look more at iCat would increase the perception if iCat's emotion such as disabling the keyboard while iCat is answering, but from the feedback we got we conclude that such coercive methods will only lead to user frustration. Third, the emotional feedback given by facial expressions did not match the plain voice of the text-to-speech system. The lessons learned from the experiment are that the demands of a natural language user interface by means of speech recognitions are out of scope of the current technology. A combination of natural language and emotional feedback with keyboard input had a negative effect on the interaction. A second design experiment of a waiter application was carried out to see whether it is possible at all to design a balanced RUI application. We were mainly interested in the value of the general application concept and therefore validated the scenario with feedback from experts in the field. The feedback that we got on the waiter application suggests that it is in general possible to construct a balanced application with current available technology.

3.1.6 Conclusion: Balancing framework

In this paper we argued that human-like interaction modalities will increase the acceptance of an interface if applied in a balanced design. We found that a balanced design is a key factor for the success or failure of an RUI. The four key dimensions to take into account are: user, application, interface and technology. The difficulties in finding a balanced design stem from the fact that RUIs address social interaction capabilities that are not yet fully understood. In order to validate the framework, we developed two scenarios: a conversation bot and a waiter application. We consider the conversation bot example to have failed because of various violations to the balancing framework and not due to principle problems of imitating social interaction. With the waiter application we presented a balanced design incorporating a RUI and validated the application with experts in the field. The results of our experiments indicated that RUIs could provide means to increase the acceptance of devices and allow them to be used by a broader range of people.

3.2 Interaction design challenge

This section analyzes the interaction design challenge. This analysis has been presented at the First International Conference on Adaptive and Self-adaptive Systems and Applications (ADAPTIVE 2009) [210].

Abstract – A fundamental issue in designing a user interface for an autonomous device is the level of end-user control. Typically, the designer of the user interface has to make a trade-off between ease of use and full control over the device.

In this study we analyzed different approaches adopted in interface design, and identified two basic categories of control: configuration and programming. Currently, most approaches adopt configuration, due to its simplicity, but we observe a shift towards programming interfaces due to the limitations of configuration. We identify several essential differences between these two approaches towards interface design, and suggest three criteria that can guide a designer in choosing a particular category of control: (1) modified versus added control, (2) unconditional versus conditional control and (3) level of device autonomy. Our criteria help to decide when configuration is not sufficient, but programming should be used instead.

Technology advances allow us to automate systems with less need for human interaction. Typical applications of autonomous systems can be found in home automation domain, e.g., in energy management, climate control, or atmosphere creation. Home robotics, for example for autonomous vacuum cleaning is another fast growing domain[74].

One of the core needs for the users of such systems is maintaining the feeling of staying in control [225]. Designers of user interfaces for autonomous systems face an increasing challenge to find the appropriate balance between ease of use and satisfying the user's need to feel in control of the system. Studies have been conducted to investigate to what extent certain system functionality should be automated for various application scenarios [181, 204, 42]. However, only little attention was paid to the question of which interface concepts are appropriate to allow different levels of control over the autonomy of devices. In this study we analyze different interface concepts applied in user interface design and offer a classification of interaction concepts suited for controlling the device's autonomy.

Most of the design guidelines for designing interactive autonomous systems originate from the field of human-computer interaction [18]. Typically, a user-centered design approach is taken, in which the user is an integral part of the design process. The most commonly cited principles are still the three design principles conceived by Gould and Lewis in 1985 [90]. These principles, originating for the development of computer human interfaces have been evolved, critized and adapted along with the development of new technologies and application domains [37]. Höök [107] argues for the domain of embodied conversational agents that traditional user studies try to generalize interaction patterns by considering a standard user, while such an average user in fact does not exist. She proposes to focus on individual interactions, including feedback from the user in very early stages of development. Interestingly, she identifies two problems in the design process that "... concern the timing of events and the level of control handed to the end-user" ([107], page 136). She points out that these problems need close attention in the design process but does not offer any design decisions related to the level of control.

This study is written for designers of interactive autonomous systems and offers an analysis of the core differences between configuration and end-user programming in interface design. A system designer may select from a variety of interface artifacts to construct the interface, ranging from a single button to fully open application programming interfaces, with graphical interfaces in the middle. Given the diversity of possibilities, it is not trivial

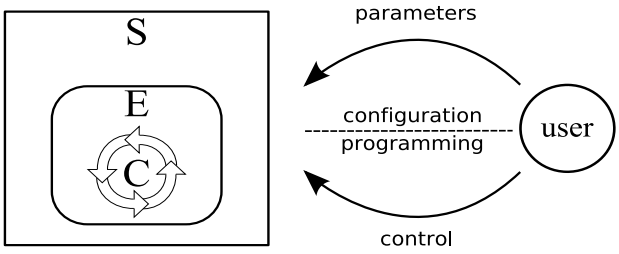


Figure 3.3: A user is interacting with an autonomous system S that contains an execution engine E running a control algorithm C. The user influences the behavior either by specifying parameters or adding new control to the system.

to select the right interface artifact for the right level of control. There are aesthetic aspects of the decision, e.g., using a button, value slider or turning knob, as well as conceptual aspects, which interface artifact matches what level of control.
The conceptual differences between configuration and programming will be analyzed and the consequences for interface design decisions will be deduced in the next section.

3.2.1 Configuration versus Programming

Design decisions for enabling optimal user interaction and control with technological interfaces can be clustered in two basic classes: configuration and programming. A system designer can therefore base his decision on which interface artifacts to use on the class of control of the user interface. The difference between those classes might seem obvious at first glance, but the consequences of the choice between the classes are often neglected. Furthermore, both concepts are confused with each other when using natural language. Therefore, the designer needs a clear understanding of both classes in order to justify his design decisions.

Conceptual differences

For analyzing the differences between the classes the scenario depicted in Fig. 3.3 is considered. A user is interacting with an autonomous system *S*. The system contains an internal execution engine *E* running a control algorithm *C*. The control algorithm determines the behavior and the level of autonomy of the system.

The user influences the behavior of the system through its interface. If the interface only allows the user to modify parameters of the provided control algorithm, the user is configuring the system. As soon as he adds or modifies the control itself we talk about programming. Choosing a radio channel or setting the volume of a TV set are examples of configuration, because such actions only affect the parameters of current control. Also more complex interactions such as setting a list of favorite channels or arranging a VCR to record a movie change only parameters, and belong therefore to the class of configuration, even though the latter is often referred to as "programming" the VCR rather than configuring it. However, if the user owns a modern media center for which he adds a new control flow for a specific action (for example, by redefining an action such as recording), he is programming the system. To be programmable the system must have an execution engine that can be extended with additional control.

Perceptual differences

From a user perspective, there is a perceptual difference between the configuration and programming classes of control. People appear to use different metaphors for dealing with configuration and programming interfaces. For example, people easily understand the meaning of a light switch, or how to select the volume of a TV set with a turning knob, by observing the consequences of changing its state. The state holds a certain configuration of the system. In terms of interface design that means that whenever a state needs to be represented, methods of configuration are sufficient.
In contrast, programming an autonomous system requires to describe the dynamic behavior of system, rather than its state. For example, a graphical user interface representing blocks that control the flow of data [119] could be used as a metaphor to represent dynamic behavior. In this metaphor, the flow of data can be conceptualized with the flow of water. Another metaphor for programming is the use of an animated character. Humans are naturally biased to interpret autonomous behavior as actions of animate beings and to interpret those actions according to social rules [198]. Adding a new behavior could be described as *teaching* it a new behavior. Researchers are only beginning to investigate the full potential of dealing with interactive autonomous systems [232] and to investigate new metaphors for end-user programming.
For creating an intuitive and easy to use interface, the designer has to find an appropriate metaphor depending on which class of control he chooses. We identified three major criteria that can help to choose the right class

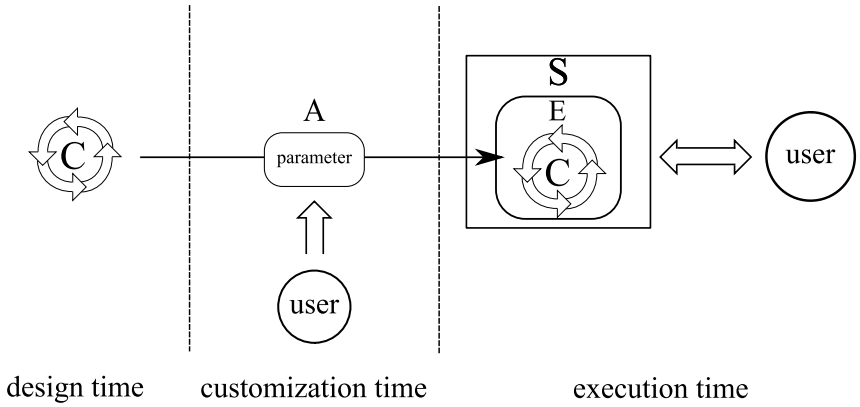


Figure 3.4: The user is interacting with the system S during execution phase. The control algorithm C was modified during customization phase and is executed by an execution engine E.

of control: (1) modifying versus adding control algorithms, (2) conditions, (3) level of autonomy. We devote a separate section to each of these.

3.2.2 Modifying versus adding control algorithms

We can distinguish three phases in the lifetime cycle of an autonomous system: *design phase*, *customization phase* and *execution phase*. A user interacts with a system during the *execution phase*. The term *execution phase* emphasizes the fact that the interaction is governed, from the system's side, by executing some control algorithm. Typically, the algorithm is pre-built into a device by the device's producer, in the design phase. If the algorithm is parametric (like showing a video on a display, with particular brightness, contrast, and audio volume) the user can influence the algorithm by setting the parameters, in the customization phase. This kind of customization is what we mean by *configuration*. The above scenario is shown schematically in Fig. 3.4. S denotes the system, *C* denotes the control algorithm, and the double arrow depicts the interaction.
Another way to customize the predefined control is to extend the existing control with a new one as shown in Fig. 3.5. In this scenario, the control algorithm that governs the interaction during the execution phase may come from two sources. It can be either specified during the design phase by the designer (as depicted by arrow A) or during the customization phase (as depicted by arrow B). Of course, the ability to add new control demands end-user programming.

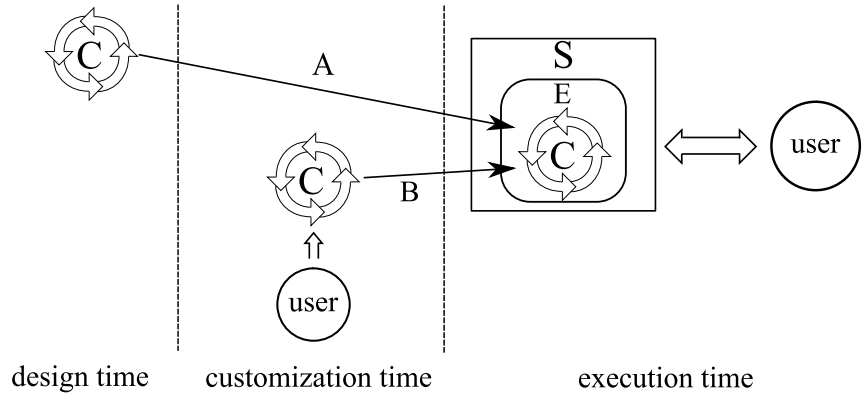


Figure 3.5: The user is interacting with the system S during execution phase. The control algorithm C was added during customization phase and is executed by execution engine E.

The distinction between the customization and execution phase is only a conceptual one. In the actual device, execution of primary functions and customization may be blended, so that the user perceives customization as one of the primary functions of the device.
In summary, end-user programming gives a user the ability of adding new control algorithms to a device while configuration offers only the ability of modifying existing control algorithms pre-built into the device by its producer.

3.2.3 Conditions

To instruct an autonomous systems the user has to specify *what* function to perform, *how* to perform it and *when* to perform it. For example, a TV set can show TV channels or DVD movies, or serve as a monitor to a home PC. In this case, the content describes *what* function to perform. In addition, a user can set various parameters (like brightness, contrast and volume) that instruct on *how* various video contents should be shown.
The difficulty for the interface designer arises as soon as the user needs to specify *if* or *when* a function should be executed. Taking the example of a security system, suppose the user wants to instruct the system that the image of a security camera should always be displayed on the screen that is closest to his current position, e.g., PC, TV or a photo frame. In this scenario the user introduces conditional execution, i.e., display the content if the screen is close by and switched on.

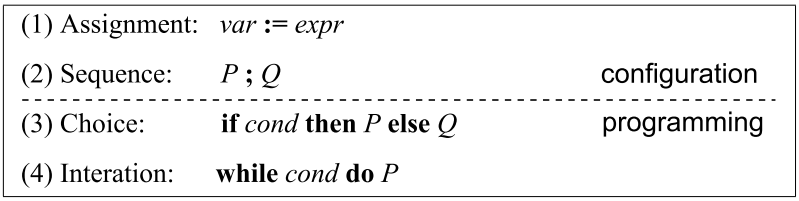


Figure 3.6: Basic constructs needed for configuration and programming. Configuration can be reduced to assignments (possibly put in a sequence), while all four are needed for programming.

While the *what* and *how* can be handled by configuration, e.g., by offering the user to choose from a finite set of parameters, conditions require programming. This can conceptually be explained by analyzing the task from a computer science perspective. In computer science, it is well known that, on some level of abstraction, sequential programming can be reduced to four basic constructs: assignment, sequence, choice and iteration (see Fig. 3.6). In this context, configuration can be reduced to assignments and their sequences, since a configuration action (say, setting the volume of a TV set) corresponds to an assignment (of some value to a variable in a control program), and several configuration actions can be performed by a user in sequence. Full programming, on the other hand, involves conditions that govern choice and iteration. From this, it is clear that configuration is just a special case of programming, albeit a much simpler one.
The need to specify conditions is the main source of difficulties in making end-user programming easy. While existing interaction techniques support well the process of configuring a system, e.g., through buttons, sliders or turning knobs, we simply do not have obvious metaphors for representing complex conditions. The usual way of expressing conditions, as known in computer science, by applying Boolean operators to atomic propositions is too complex for an end-user with no prior experience in programming. With every new condition the number of possible paths of execution increases exponentially which easily becomes too difficult for a user to maintain.

3.2.4 Autonomy

The last criterion to decide on the class of control is the level of device autonomy. As Kawamura observed [127], there is a relationship between the level of autonomy and the way a device should be instructed by a user. For example, in the extreme case of a fully autonomous robotic device

equipped with a sophisticated control algorithm that enables the device to automatically make decisions, there is no need for user instructions at all, because the robot does everything by itself. On the other extreme, a device equipped with a primitive control would demand almost constant instructions from a user. Whether such instructions should be given in the form of configuration or programming is not obvious. Even less obvious are the levels of control that lie between the two extremes.

System autonomy has been defined and measured by a variety of constructs such as the system's intelligence, capabilities, level of interaction or degree of self-government [171, 98]. Joslyn, for example, describes autonomy in terms of isolation from the environment [121]. In this study we follow Gunderson's definition of autonomy as "...the ability of a system to make choices and enforce its decisions." [98].

We analyzed three different autonomy scales known from past studies: (1) system automation [216], (2) user interaction [101] and (3) origin of control [171].

The first autonomy scale, system automation, concentrates on functional aspects of a device. It is based on the assumption that the main purpose of an autonomous device is to free the user from tedious or repetitive tasks. The level of autonomy is measured by the length of time intervals the system can be neglected by the user, while performing its main functionality. Several scales that capture this notion of autonomy have been developed. The most commonly used one is a ten point scale listed in Fig. 3.7. The first variant of this scale was already developed by Sheridan in 1974 [216]. On this scale, the levels of autonomy are classified according to the time a user spends interacting with a device. The less time needed, the more autonomous the device is. This classification scheme is useful, for example for analyzing to what extent a complex system, such as air-traffic control or unmanned systems, should be automated in order to measure the impact on the user's performance [204, 123].

However, this ten point scale is not useful for an interaction designer as a criterion for deciding if a device should be instructed by configuration or by programming. When analyzed from a user interface perspective, it appears that on the levels 1 to 3 configuration is sufficient, because the mentioned interaction scenarios can be narrowed down to choosing from a finite list of options. For example, on the first level the user exerts direct control over the device by triggering its basic behaviors. These choices are on the following layers consecutively narrowed down by increasing the level of abstraction and decreasing the number of options. On the levels 4 to 10,

1	the robot offers no assistance; the human must take all decisions and actions
2	the robot offers a complete set of decision/action alternatives
3	the robot narrows the selection down to a few options
4	the robot suggests one alternative
5	the robot executes that suggestion if the human approves
6	the robot allows the human a restricted time to veto before automatic execution
7	the robot executes automatically, then necessarily informs the human
8	the robot informs the human only if asked
9	the robot informs the human only if it decides to
10	the robot decides everything, acts autonomously and ignores the human

Figure 3.7: Ten point system automation autonomy scale of an autonomous robot (Sheridan 1974 [216]).

there is no need for instructing the device at all, at least regarding its main functionality, because the device does not offer the possibility to modify the behavior. Hence, the conclusion would be that programming is not necessary to instruct a device, no matter what its autonomy level is.

The second autonomy scale, user interaction, concentrates on the interaction abilities of a device. It is based on the assumption that interaction with the user is one of the core functionalities of the device. The user perceives the device as an artificial agent with whom he is cooperating. The focus on a team like collaboration leads to a different scale of autonomy which emphasizes the interaction aspect [101]. The scale used for this mixed-initiative approach contains 5 levels, with direct control at the lowest level and peer-to-peer collaboration at the highest level of autonomy as listed in Fig. 3.8. This kind of autonomy has little in common with the concept of autonomy based on functionality, i.e., system automation. It should be treated as orthogonal to the previous scale. For example, a robot that is

1 direct control (also called *teleoperation*)
2 mediated control (also called *mediated teleoperation*)
3 supervised control
4 collaborative control
5 peer-to-peer collaboration

Figure 3.8: Interaction based five point scale of the level of autonomy of an autonomous robot (Goodrich and Schultz 2007 [89]).

capable of navigating autonomously and that communicates its position to collaborate with people, would score high on the five point scale. However, it would score low on the first scale, because it would interact with other team members, and thus requires time for interaction.

Analyzing the scale form a user interaction point of view, a relationship between level of autonomy and the potential for programming as depicted in Fig. 3.9 was identified. On the extremes of the scale, i.e., direct control and peer-to-peer collaboration, configuration can be applied to exhibit control over the functionality of the device, while for supervised control programming needs to be applied.

For robots that are on the first level of autonomy, the user only chooses from the set of basic low-level functionality such as setting speeds to the motors. While this kind of interaction clearly can be addressed with configuration, the user also just configures the system behavior on the highest level of autonomy. This becomes more obvious if considering the reasoning engine of the robot that is necessary to interpret the commands of the user. The user is not programming a control in the sense of specifying a sequence of actions, but only configures the reasoning engine to create the sequence of commands that is necessary to accomplish the given task for him. This subtle difference has an important impact on the interface design. Instead of requiring the user to think about control structures the designer can present a list of high level commands such as "*check the room*" or "*report your status*". Towards the middle of the scale, the potential of incorporating programming techniques increases. On level two, mainly the abstraction of the set of commands increases (e.g., driving straight), but there is already some potential for programming, for example by offering the user to define new commands that consist out of a sequence of more basic functions.

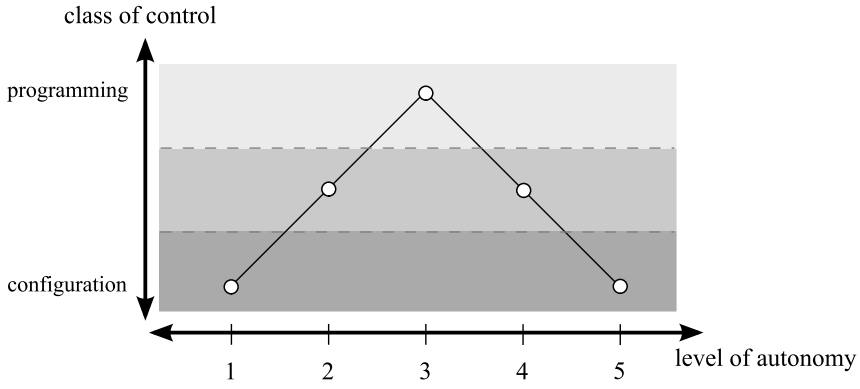


Figure 3.9: Class of control needed for different levels of autonomy on an interaction based scale.

Another good example for such interfaces is programmable remote controls. On level three the user is still in full control over the behavior, but does not spend continuous attention to the robot as in the direct control situation. Instead he specifies higher level plans to the system. On this intermediate level of autonomy, the devices still lacks a sophisticated reasoning engine, requiring the user to give a detailed specification. End-user programming techniques offer exactly this class of control. On level four, the control over the behavior is equally shared between the user and the robot. The robot is capable of making decisions of its own, but still requires guidance of the user. Finally on level five, the control resides solely on the side of the device.

The autonomy scale, origin of control, is proposed by Musse et al. [171]. They measure the level of autonomy by the amount of control that needs to be provided by the user. Their scale considers three levels of autonomy, 1)guided, 2)programmed and 3) autonomous as stated in Fig. 3.10. On the guided level, the control resides fully on the side of the user who is triggering the behaviors of the device. On the intermediate, programmed, level the device offers means to describe behavior, rather than directly executing it. Lastly, devices in the autonomous category are capable of acting independently. All necessary control is implemented on the device itself.

Among the three scales, we consider the three point scale the best criterion to discriminate between configuration and programming, because it offers the most concise description of end-user interaction. It explicitly measures the level of autonomy in terms of origin of control. Important from a de-

1 guided
2 programmed
3 autonomous

Figure 3.10: Origin of Control based three point scale of the level of autonomy of an autonomous robot.

sign perspective is that the relationship between the level of autonomy and class of control for the interface follows the same qualitative shape as found for the five point scale. Devices in the guided category do not have any sophisticated decision making capabilities, but the user is fully in charge over the behavior of the device. As outlined before, this type of interaction can be addressed using configuration techniques. In the programmed category, devices exhibit the potential of making autonomous decisions, but need to be instructed to do so, implying the need for describing behavior. On the highest level of autonomy all necessary control resides on the side of the device. Hence, there is no need for the user to add new control to the device implementation, but configuration is sufficient to give the user control over the autonomous behavior of the device.

In summary, as soon as the designer has classified the requirements for a device according to its level of autonomy, he can deduce a need for configuration or programming.

3.2.5 Execution versus description

An important difference between configuration and programming is the time in which both activities take place. As mentioned in the previous section, the purpose of programming is to *describe* the future execution of control while the purpose of configuration is to modify the current *execution* of control, by setting some parameters that influence the control on-the-fly. In other words, configuration is performed when control is being executed while programming is done before the control starts executing.

Of course, this difference is only important for a designer, and not for an end-user. For the end-user, controlling a device, either by configuring or programming it, is part of what she perceives as simply using it. However, for the designer the difference is important since it strongly influences the software architecture of an execution engine. An architecture that enables

the ability to configure a system is much simpler. If arranged as a reactive system (i.e., structured as a set of components with a master control module and drivers for sensors and actuators) it typically consists of one execution engine that handles both configuration and control. The configuration interface is handled as part of control, where some sensors are used for triggering configuration actions (e.g., up and down buttons on a TV set remote controller are used to change sound volume) and some actuators are used to give feedback (e.g., the current volume level). The architecture that enables programming, however, must contain two additional components: one for specifying a program and the other for executing it.

3.2.6 Conclusions: Interaction design

Autonomous systems typically contain various control algorithms built into a system by the system's producer. We propose a simple conceptual framework to support the design of user interfaces for controlling autonomous systems. This framework, is based on the fundamental assumption that the purpose of such an interface is to allow a user to either *modify* the pre-built control algorithms or *add* some new ones (or combine both activities). We term the first activity *configuration* and the second one *programming*.
Essential differences between configuration and programming were identified, and their impact on designing the interface for controlling autonomous systems was studied. In essence, configuration only sets parameters of a pre-built control algorithm, while end-user programming describes the process of adding new system behavior and therefore offers a more general class of control. Behavior then depends on run-time conditions.
Designing an interface for configuration is much easier than designing for programming. Most interaction technologies, be it speech, gestures, tangible or graphical interfaces, can be used for configuration. For programming, however, the choice is much more limited; i.e., usually to textual notations, or some symbolic notations disguised in graphical representations. In addition, programming is usually perceived as difficult by casual users. For these reasons, a designer should favor configuration over programming, and revert to programming only if necessary.
To guide the designer in making this choice, we propose to use three criteria. First, should a user be allowed to only *modify* the pre-built control algorithms or also *add* some new ones? Second, is it enough to instruct a device by simple unconditional actions performed by the user, or the actions must be governed by user defined conditions? Third, does the device exhibit a level of autonomy proper for configuration, or proper for programming?

3.3 Behavior design challenge

Having discussed the design challenges at application and interaction design level, a third study was conducted to investigate the design challenges at behavior design level. This study has been presented at the Robot and Human Interactive Communication conference (RO-MAN) [212]

Abstract – Robots use their movement to interact with the user. These movements are a crucial part in designing the interfaces of personal robots. They have a major impact on how a robot is perceived by the user who is interacting with it. Subtle flaws in the movement can elicit a repelling feeling and therefore negatively affect the interaction. On the other hand, carefully designed little quirks can give the robot a perceived personality and make it more interesting for the user. A possible relationship between appearance and motion on how a device is perceived is described by the 'Uncanny Valley' conceived by Mori in 1970. It is widely used as a design guideline for creating humanoid interface robots. It does not, however, describe the impact that the quality of movements has on the perception and the believability of a character. Conclusive frameworks for the animation of robots are rare and are often adapted from cartoon or computer animation. This section analyzes the behavior design challenge and presents refined design guidelines for the motion of robotic characters with respect to the design requirements naturalness, adequateness and development over time. Furthermore, we created a set of tools that eases the process of designing the movement of the robot by improving on common animation techniques such as key-framed or scripted animations. The tools preserve the freedom of the animator to define specific expressive motions while at the same time offering the full power of scripting technology.

3.3.1 Introduction

Currently, we observe a trend in which robots leave their socially isolated industrial workplaces and enter our every-day lives. It is predicted that they take the same development as personal computers and will be common in every household [81]. Personal robots are a subset of these robots that directly interact with humans. They have the potential to become powerful

tools for a variety of application areas, as they serve as an interface for ambient environments, entertainment such as gaming and storytelling, and healthcare applications. The latter one is of increasing importance since our society faces an increasing number of elderly people that are in need for care. In these application areas, users require natural and easy means of interaction with such robots. The advantage that personal robots have over other interfaces is that they can engage in a social and natural interaction with the human user. In [198] Reeves and Nass showed that people apply social rules to media and especially computers. They even apply these rules when they are consciously aware that they are dealing with a machine. Social interactive interfaces exploit this phenomenon to maintain a social relationship in order to add an additional source of information for the communication. In [27] Breazeal studied whether comparable effects can be achieved with robotic embodiments, since a physical robot gives only a crude approximation of human attributes. She applied human-like artifacts to the robot design and found that a robot can evoke the same social responses. She also found that social expressions of a robot have a positive influence on the interaction, as participants were more engaged in the interaction. With their physical presence personal robots are well suited to bridge the digital and the real world. Hence, for future interface devices social interaction is an important design factor. How a robot is perceived depends on multiple factors, such as its appearance, behavior, sound, smell and what its materials feel like. In this section we will focus on the design of one of these factors for a robotic character, i.e., motion. Motion takes a crucial part in the design for a social interactive interface, but its communicative power is often neglected in favor for appearance. This neglect of motion is problematic, because humans are naturally biased to perceive biologic motion. People are able to attribute agency, drives and intentions only based on motion cues [104]. The importance of motion has also been demonstrated by Ishiguro, who exposed participants for two seconds to a realistic android [114]. In one case the participants saw just the static android and in the second case the robot performed small movements. In the second case significantly less people became aware of dealing with a machine. These results showed that next to appearance motion has to be addressed explicitly in the design of a personal robot. Some authors go even further and attribute a greater importance to motion than to appearance. Maddock et al. for example argues that a photorealistic appearance is of less importance in his discussion of methods for realistic animation of animated faces than the behavior [154]. Photo realistic appearance seems to

be easier to reach for artificial characters than realistic motion. This holds true also for virtual characters despite the fact that they have all the freedom to perform any movement without physical constraints. Their robotic counterparts, on the other hand, are much more limited. The challenge that designers of personal robots face is to create high quality movements that convey familiar artifacts and support the interaction.

3.3.2 Designing personal robot behavior

Design space of behavior

Creating socially accepted personal robots is a design challenge not fully understood yet. What is known is that the appearance and behavior of a personal robot play a crucial role in how it is perceived. Because humans are an example of a socially behaving actor, one approach to design a socially accepted robot is to adopt human characteristics. Mori, however, formulated a theory that there is no linear relationship between the similarity to a human and the perception of familiarity of the robot, known as the "Uncanny Valley" [170, 152] (see Fig. 3.11). This theory describes the repelling feeling that an almost human-like robot evokes. The lesson is that subtle flaws in the design of the personal robot can have a strong negative effect on the interaction. Therefore, the application designer faces an additional design challenge for carefully designing the motion. The Uncanny Valley theory serves as a design guideline for creating robots and is mostly applied to the appearance of a robot. However, in the original version of the Uncanny Valley theory Mori also described the effect that motion has on the perception of the character. He predicts that motion will exaggerate the peaks. The importance of motion was later also addressed by Gee in his review on the Uncanny Valley theory [83]. He concludes that the uncanny valley theory needs some more refined redefinition that includes the influence of all factors and their interdependence. One extension to the original graph is presented by Ishiguro [114]. He added 'similarity of behavior' as a new axis orthogonal to 'similarity of appearance' to account for human-like behavior. He argues that already the shape of the robot raises certain expectations. If the expectations are met, the robot appears more familiar to the user. However, how the robot is perceived depends on multiple factors including appearance, motion, speech, social and religious context. What is not addressed so far is the effect that the quality of motion has on the perception of the character. A sack of flour animated in a right way can convey familiar attributes like emotional states and intentions [231]. Since

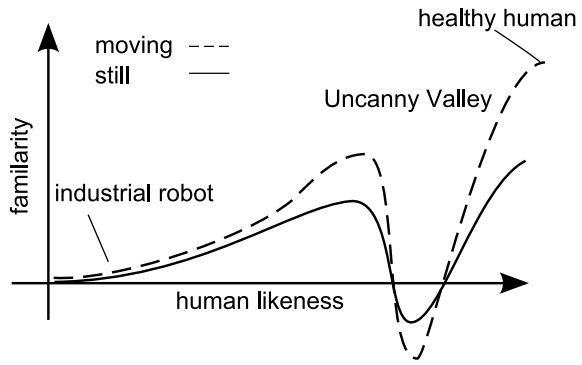


Figure 3.11: Simplified version of uncanny valley graph conceived by Mori 1970 [83]

a sack of flour has not a human-like shape it would be placed very low on the familiarity axis of the Uncanny Valley graph. On the other hand, a photo realistic humanoid can give a repelling feeling if the motions are not designed carefully.

Conceptual framework

To develop our motion design tools we present a conceptual framework of the design case. In our framework we consider a designer who wants to design the behavior of a personal robot. The design case consists of a personal robot situated in some context and being observed by an observer. The designer's task is to make the robot act such that it conveys a well-defined message to the observer. The personal robot communicates a message to the observer through physical channels that can be sensed by the observer. For instance, a robot communicates by moving, making sound, speech, lights, odor, temperature, vibrations or any other physical channel. The interpreted meaning of the robot's act depends on the context and the observers' profile. A particular gesture can have different meanings in different contexts or in different cultures. Notice that even when the designer does not intentionally put a message into the act of a personal robot, an observer always interprets the act and thus receives a particular meaningful message. Therefore it is always important to study what messages a robot communicates when it is acting. We classify robot behavior into two classes. First, a robot might show control behaviors. A control behavior is generated to realize system goals, such as stabilization,

localization, navigation etc. The second class of behavior is communicative behaviors. A robot generates communicative behaviors when it wants to communicate intentionally a message to some user, such as greeting a user. As will be discussed further on, each of these two classes of behaviors requires a different tool to design them efficiently. This paper focuses on the development of tools for designing communicative motions. In the next section we will discuss and motivate which requirements motions need to fulfill in order to be qualified as good communicative motions.

3.3.3 Motion design requirements

In mobile robotics one design goal is to let the robot localize itself and navigate through some environment. Different movement strategies, though all labeled as 'artificial intelligence', cause a different perception of its intelligence [27, 113]. Therefore, a variety of methods have been developed to make the motion of an embodiment explicit and convey an intended message. These methods can be classified into three classes:

Trajectory Design Methods These design methods directly generate the individual trajectories of the motion variables of a robot. Keyframed animations [231], motion capture [221] and scripted animations [185] fall into this category. Motion capture and scripted animations within this context take a purely generative approach and do not perform further processing to existing motion data.

Motion Editing Methods These methods are based on editing the whole sequence of given motion data. Different editing techniques are developed, such as motion signal processing [30], retargeting of motion to new embodiments [85] and constrained based motion editing [86]. Also blending of existinganimations to create new ones belongs to this category [198].

High Level Behavior Design Methods These methods are based on generating motions by specifying the robot act in terms of abstract parameters. For example emotional reasoning methods that calculate blending parameters to existing animations [179] or behavioral control [236] that triggers animations can be accounted to this class of methods. Independent of the design method used by the designer the resulting motion should adhere to certain quality requirements in order to create believable robot motions. We have identified three of such quality variables, naturalness, adequateness and development over time. These are discussed below.

Naturalness

Naturalness of motion describes the movement models for the actuators. It covers the design of detailed trajectories as well as the overall coherence of motion for the character. In this context 'natural' does not mean as similar to human motion as possible, but the motion should be perceived as natural for the character to be designed. A robotic looking character should describe more robotic like trajectories, for example by choosing motion trajectories with equal velocities [238]. Mostly, Trajectory Design Methods are used to conform to this design requirement. Keyframing methods serve a good representation for creating expressive motions for all kinds of embodiments, but make it difficult to model physical correct motions. Physical modeling techniques, on the other hand, model a movement as a force affecting a physical system. This creates natural motion trajectories, but it makes it difficult to create specific expressions. Motion capture methods cover both but are restricted to embodiments that have a similar dynamics as the human body. Later on we will present a method in which keyframed and procedural animations can be combined to take advantage of both methods. Carefully designing the animation according to this motion design requirement will enable the animator to play non interactive pieces of realistic motion. He might combine them to a short theatrical play.

Adequateness

The adequateness of motion describes the situated context and the reactivity of the character. The animator has to make sure that the natural looking motion sequences are displayed at the right moment. The 'right moment' refers to timing and context of the action. The context is determined by the communication partners that observe the robot, their social relationship and by the application scenario. Therefore this design requirement includes modeling a personality of the robotic character as well as accounting for cultural, religious or other expectations that the user might have. Describing the relationships between events and reactions or cultural norms require a different representation than describing the position of the actuators over time. For example rule based systems or scripting technology are well suited for this purpose. We will describe a method how these representations can interface with natural and expressive animations later on.

Development over time

Development over time addresses the behavior that the robot shows over time and how it changes over time. For example, a character should avoid to show the same animation twice within a short period of time. Humans are very sensitive to biological motion. A repetitive behavior is attributed to machines, but not to living creatures. No two movements of a biological creature are exactly the same. Repetitive behavior should not be mistaken with rhythmic motion. For humans it is natural to synchronize the motion for example with music, but even if the overall pattern stays the same there are always little variations. Development over time requires the designer to create a coherent history of the character, including short term states as well as long term memory. For example an expression of the character depends on its emotional state and a longer term mood. Additionally, the modeling of a history enables the character to evolve over time. The best tools at hand for this design requirement seem to be again rule based systems and scripting technology. A special database can be used to keep track of the evolution of the character, e.g., improvement by learning new skills or changing the relationship to a person. Future behavior can be modeled with evolutionary algorithms.

3.3.4 Motion design tools

Many tools exist today that relate to the motion design problem described in this paper. Designing adequate believable motion is more an art than science. Therefore, no method is available to fully automatic generate believable motion for a specific type of robotic character or personality. For this reason we focus on design tools to provide an editing framework that helps the animator to abstract and generalize carefully designed animations. Existing tools can be roughly divided into two groups. The first group contains tools from the field of traditional computer animation. These tools are developed to animate virtual gaming or cartoon characters. The second group contains robot programming tools. After having discussed the strong points and limitations of some of these tools we will present our improvement for a motion design tool for believable embodied robotic characters.

Existing tools

A rich set of computer animation tools exists, including Maya [4], Lightwave [1] and the open source tool Blender[2]. These tools are used to create animated characters and are used for productions such as the 'Shrek'[3] cartoon. Besides being tools to create motions of virtual characters these tools also support modeling character as well as rendering them. The advantage of computer animation tools is that they embed strong character animation techniques such as key framed animations, scripted animations, morphing techniques and motion capturing. The major disadvantage of computer animation tools for the design of believable motions for embodied robotic characters is that they operate on the pixel level. This allows these tools for instance to apply morphing techniques to create facial expressions, but the results depend to a large degree on the skills of the animator. Such a technique, however, cannot be applied to embodied robotic characters as they only have a limited set of actuators that are controlling parts of the robot. Furthermore, the editing techniques are limited to design time and make it difficult to change an animation at runtime to create a more interactive behavior. Besides computer animation tools also a rich set of robot programming tools exists. Matlab/Simulink[4] is a well know scientific programming tool used to developed control algorithms. By using a graphical block diagram tool an algorithm can be designed. The advantage of this tool is its graphical interface that is very intuitive and convenient to use. Also the Matlab environment provides many useful mathematical libraries such as neural networks and signal processing libraries. The disadvantage of this tool is the lack of tools for specifying precise motor trajectories which are needed to create believable motions. Lego®Mindstorms NXT[5] provides a graphical programming tools based on National Instruments' Labview. While Simulink blocks represent continuous or discrete time signal processing elements, Lego's NXT blocks represent programming elements such as performing a task or doing a 'while' loop. The advantage of this tool is its ease of use for building programs. The disadvantage is the same as that for Matlab/Simulink, namely the lack of tools for specifying precise motor trajectories. Recently Microsoft Robotics Studio[6] was introduced to the

[1] http://www.newtek.com/lightwave
[2] http://www.blender.org
[3] http://www.shrek.com/main.html
[4] http://www.mathworks.com
[5] http://mindstorms.lego.com
[6] http://msdn.microsoft.com/robotics/

robotics community. This tool aims to unify different robotic architectures by providing a general software framework based on services. In a recent update also a graphical programming tool has been added to Robotics Studio. The advantage of Robotics Studio is its graphical tool for programming control behaviors. A shortcoming is the lack of tool support for communicative behaviors. The Open Platform for Personal Robotics (OPPR) [188] is a set of tools for developing applications for believable user-interface robots developed at Philips Research. It was developed for the iCat robot – a non-mobile robot with a cat-like appearance and a cartoon face that can show facial expressions. The advantage of OPPR is its graphical motion design editor for embodied robotic characters. A designer can choose to design motions graphically or he/she can choose to script a motion. Besides the editor, OPPR also contains the 'Animation-Engine' software module, specialized module that allows users to simultaneously play and blend motions smoothly. In this way, complex behaviors are created from a library of basic behaviors. Several concepts of OPPR such as the Animation-Engine have been reused in the development of a unified software architecture to develop social robotic interfaces. A shortcoming of OPPR is its lack of high level motion design and high level sensor integration. Therefore, OPPR addresses only the first of the three design dimensions, i.e. naturalness. In the following sections we present our extensions to the OPPR framework that allows addressing the identified motion design requirements.

Functional animations

As a first extension to the OPPR framework we focus on the naturalness of motion. The animator on the one hand wants to have full control over the expressiveness of the character while on the other hand preserving a high level control over the behavior. Therefore we developed 'functional animations' that combines the expressiveness of keyframed animations with the power of parameterized control of procedural animation. Existing keyframing animations focus on an expression at a specific moment in time. Using this approach they have full control over the posture at this moment, but neglect the importance of the in-between frames [253]. Procedural animations on the other hand define a trajectory over time that generates the postures of the character for all times with equal importance. We base our system on keyframing an animation, but instead of focusing on an expression at a certain moment in time we make a keypoint responsible for a whole interval. In the following discussion the term keyframe describes the overall posture of the character at a given amount of time. Following

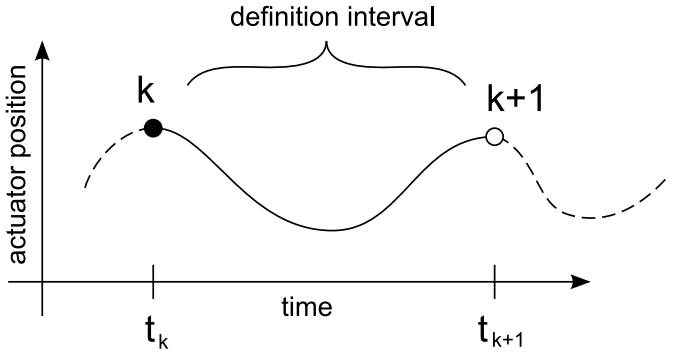


Figure 3.12: Keypoint interval definition for functional animations

the notation given in [241] the complete posture of a robot is defined by a keyframe vector $\vec{s}$ holding the positions x_i of all actuators:

$$\vec{s}(t) = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \tag{3.1}$$

The term keypoint refers to a control point that is defined in the scope of one actuator. It specifies the position x of this actuator at a specific moment in time t by its coordinate (t, x). In our system we annotate it with a function f, which defines the trajectory of the actuator after the keypoint. We refer to it as an interpolation function. A keypoint therefore does not only hold a single position, but explicitly defines the motion trajectory over a whole interval. The responsibility of the keypoint ends at the time where the next keypoint is placed (see Fig. 3.12). The figure shows the definition interval of keypoint k, which is positioned at a time t_k. The definition interval ends before the next keypoint at time t_{k+1}, so that $t_k \leq t < t_{k+1}$. The first keypoint defines $t = 0$ for the animation and the position of the last keypoint defines the length of the animation. The trajectory of an actuator is a partially defined function, containing K intervals. For every actuator a different number of keypoints may be defined. In every interval $1 \leq k \leq K$ the trajectory is given by an interpolation function f_k that is parameterized by time t and a set of parameters $\vec{\theta_k}$. The end of the animation for an actuator is defined by its last keypoint. That means

that from the last keypoint only the keypoint position is used, but not its interpolation function. The position of the actuator is given by:

$$x(t) = \begin{cases} f_1(t, \vec{\theta_1}); t \in [t_1, t_2) \\ \vdots \\ f_k(t, \vec{\theta_k}); t \in [t_k, t_{k+1}) \\ \vdots \\ f_K(t, \vec{\theta_K}); t = t_K \end{cases} \tag{3.2}$$

The whole animation is therefore defined by:

$$\vec{s}(t) = \begin{pmatrix} x_1(t) \\ \vdots \\ x_n(t) \end{pmatrix} \tag{3.3}$$

A single keyframe is generated by passing the time argument to every actuator and evaluating its interpolation function at time t. Using this approach the animation designer places keypoints for the actuators as in standard animation systems. Additionally, he chooses from an adjustable set of interpolation functions and specifies its parameters. We demonstrate this concept with a graphical editor for editing functional animations (see Fig. 3). Physical systems can be modeled by adding new interpolation types. The resulting trajectory can directly be observed in the editor which gives the animator visual feedback. He can adjust the curve by positioning the keypoints or modifying the parameters of the interpolation function. The second advantage of functional animations is that the same mechanism can be used to generate more dynamic behavior. By adjusting the control parameters at runtime a huge set of different behaviors can be generated based on a small set of predefined animations. For example while in traditional keyframe animation systems a nod to the left and a nod to the right would require two different animations, functional animations only require one. At runtime the parameter for the head angle can be set dynamically. We address this issue of setting the variables at runtime with 'global variables'.

Global variables

Global variables are defined in a global namespace and are accessible by all modules inside the OPPR architecture. Local parameters can be linked to

a global variable so that it always keeps the same value. With this mechanism a control algorithm can adopt an animation to the current situation by setting the global variable. The major advantage of this concept is that it allows the animator to attach meaning to changes of the interpolation function. While blending methods interpolate between two expressive animations they run the risk of destroying little quirks that give life to the character. The animator of those two basic animations has no control over changes made by blending. Global variables, on the other hand, give him the possibility to define variations of the animation in a controlled manner. However, by using a direct mapping it might be difficult to model high level control parameters such as interpolation function parameters. Furthermore, control parameters cannot be expressed. Therefore, we introduced the concept of parameter models.

Parameter models

A parameter model is an abstraction layer to control the parameters of an animation in a meaningful way. It formalizes the dependency between functional animation parameters and high level control parameters to constrain changes of the animation to a meaningful dimension. Mathematically, a parameter model performs a coordinate transformation between motion model space and motion control space:

$$\vec{\Theta} = m(\vec{p}) \tag{3.4}$$

The parameter p describe the position of the animation in a high level control space and are mapped to interpolation function parameters. Control parameters allow to decouple animation design from application design. This matches exactly the decoupling that we made while describing the motion design requirements. The animator focuses on the naturalness of the motion while an application designer is more concerned about the robots application and interactivity. Parameter models give the animator the full power of scripting technology to adjust the parameters of a keypoint. Constraints to changes can be implemented by carefully designing the parameter models. Sometimes it is more useful to have a different representation for constraints. The situation that we encountered most often was that we wanted to preserve the relative positions of keypoints within one actuator but also between multiple actuators. Without constraining the relative positions a movement in multiple actuators could desynchronize. We implemented a grouping mechanism to represent such relationships.

Grouping

A group is a set of keypoints that are constrained to keep a specific relative position to each other. Groups may contain single keypoints and other groups that are treated as a single entity within a group. A translation to one keypoint of the group affects the whole group. Second, a group can be annotated with a label. A label contains a symbolic description of the action for the character. This increases the readability of the animation. The animator can deduce the intended meaning without having to play the animation. This is especially helpful if multiple animators work on one animation. Another advantage of groups is that they define a clear hierarchy of local scopes. This hierarchy can also be applied to parameter models. For example two distinct groups (e.g., eye blinks for left and right eye) can be combined in one top level group. With a change of the emotional state of the character the parameters of both groups have to change simultaneously. Therefore we allowed to attach parameter models to groups. They are evaluated in the local scope of the associated group. Groups define an internal structure of the animation. Only the global variables and the parameter models from the outermost groups are accessible for outside control mechanisms. This preserves a well structured interface to control the shape of the animation.

3.3.5 Discussion: Behavior design

We motivated the importance of motion in the design of a personal robot. Designing the motion takes a crucial part for facilitating a natural interaction between humans and robots. We presented a conceptual framework for the motion design for a personal robot. In a first step we extracted three design qualities that can be addressed independently from each other and developed a toolset that addresses these design qualities. In our research we focused on a framework that supports the designer with editing animations. We proposed a framework that combines keyframed animation with procedural scripted animation. This allows the animator to create physical correct movements while preserving the expressive freedom of keyframed animations. Global variables and parameter models define an interface to alter the animation along a specific control dimension. These concepts decouple the animation design from the application design.

For the second and third design dimension ‘Adequateness of motion’ and ‘Development over time’ we rely on the existing scripting technology of

the OPPR system. Rule-based-systems and scripting technology were successfully used for representing the interactive behavior. For example the emotional state can be kept in certain state variables and in order to avoid that exactly the same animation is played twice, the parameter settings of an animation can be changed.

Chapter 4

Animation technology

In the previous chapter, the design challenges for creating social robotic interfaces were analyzed at three different levels: application level, interaction design and at behavior design level. These levels form a hierarchy of dependencies. Starting with the top layer, the success of the overall application depends to a large degree on the quality of the interaction with the device. The interaction consequently depends to a large degree on the quality of the impression that the user perceives in the behavior of the device.
For a successful social interaction, the device has to create a believable impression of a life-like character [47]. The term 'believability' has been introduced by Bates to describe the quality of an artificial character to appear live-like [17]. For example, timely expressed emotions are crucial to create a believable character. Fong et al. discriminate between two main design approaches to achieve believable social interfaces, i.e., 'biology inspired' and 'functionally designed' social interfaces [72]. In the first category, designs aim to mimic zoomorphic or anthropomorphic features by reproducing attributes found in living creatures. In the second category, the technical design is functional in a sense that it only appears from outside to be a social character. Dautenhahn supports the functional perspective to design social robots. She argues that,

> They need not necessarily have to act like biological agents, but some aspect in their behavior has to be natural, appealing, life-like. (Dautenhahn [46] p. 7)

Furthermore, Dautenhahn recognizes that research in believable agents benefits greatly from the design knowledge from the field of movie and animation technology.

4.1 Traditional animation principles

In the field of cartoon animation, animators have impressively demonstrated that carefully designed behaviors can evoke rich emotional responses [231, 253]. The basic phenomenon that people attribute desires, intentions and emotions to very abstract stimuli has been demonstrated by Heider and Simmel. They showed movement patterns of geometric objects to participants [104]. In their study, all but one participant interpreted the stimuli in social terms and explained the observed behaviors by abstract story lines, and social roles. Tremoulet has extended this findings by showing that already the movement pattern of a single dot is sufficient to give the impression of animacy [235].
However, many of the early animations lacked life-likeness [253]. There appears to be a fine line between animations that are perceived as believable and convincing and animations that are perceived as dull and unnatural. Thomas and Johnson, presented a list of 12 animation principles that guide the design of animations to give Disney® animations their uniqueness and illusion of life. In the context of designing believable social interactive interfaces, these guidelines capture important design knowledge and provide valuable input for the design process. The properties of these twelve traditional animation principles and their applicability to the animation of robotic platforms such as iCat are discussed in the following:

Squash and Stretch Cartoon characters have the ability to perform virtually any motion, because they are not constrained by a physical embodiment or physical laws. Squash and Stretch is used to keep a character in dynamic motion and to show the impact of internal and external events on the character. As a fundamental guideline, the volume of the character should stay approximately constant during a squash and stretch action in order to stay believable. An example is depicted in Fig. 4.1. Even though the depicted sack has no special attributes such as a face to express emotions, squash and stretch techniques are sufficient to convey a rich set of emotions.

For robot animation, however, this principle is of less relevance as it heavily relies on the robotic embodiment. An example embodiment that makes use of this principle is Keepon [70]. The squash and stretch principle allows Keepon with surprisingly simple movements to evoke a rich set of emotional responses. Keepon can synchronize its movements with music and perform a dance together with a user.

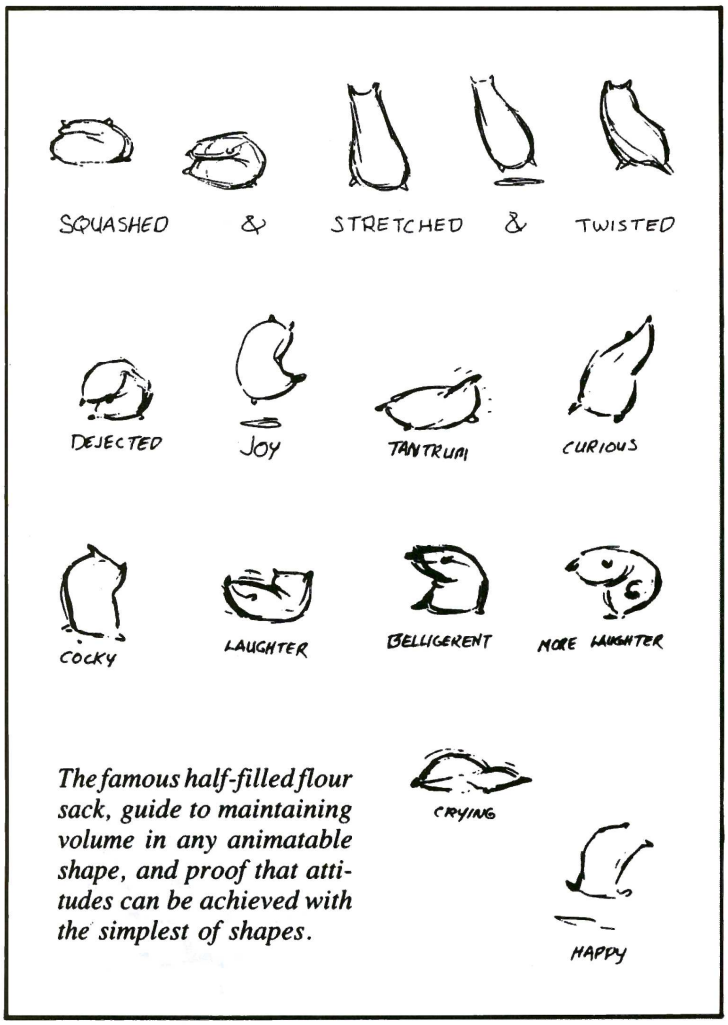


Figure 4.1: Disney's® sack of flour animated to express various emotions using the Squash and Stretch animation principle. (source: The Illusion of Life [231] p. 49)

Anticipation Anticipation is one of the most important and most often used animation techniques. Anticipation describes an action in three phases. First the audience is prepared that something is going to happen, then the action is performed and finally the action is terminated. Anticipation is used to guide the attention of the audience to focus on a particular element that might be missed otherwise. An example is depicted in Fig. 4.2, where the audience is prepared that the character will run away quickly. In the example, the character leans backwards even though this is not a gesture of person that prepares for running. Instead it resembles the preparation of shooting a bow. Anticipation helps to explain the next scene. The next scene might contain just a cloud of dust. If the audience had not been prepared, the running action might be missed, leaving the audience without explanation for the cloud.

One important point that can be derived for the design of social interfaces is that anticipation not only helps to guide the attention of the user, but also helps to build up trust. Using anticipation, the user is enabled to predict what is going to happen next. If this expectancy is met by the character, the user gets the impression to understand the device and feels in control [32].

Staging Staging guides how an action is presented so that it is well understood by the audience. Staging has its origin in theater, where actors are taught to only give one clear message at a time [228]. The audience must be guided to focus exactly where the action happens.

Staging can also be applied for the animation of robotic characters. For example, a robot might react to a certain event by displaying happiness. This expression must be clearly staged, for example by holding the expression for a certain time and making sure that the expression is visible to the user.

Straight Ahead Action and Pose to Pose This design principle supports the designer in the process of designing a complete action. Following the 'straight ahead' advice, the designer animates a sequence sequentially, starting from the first image until the last. This process helps the designer not to lose the flow of motion in the scene and gives him the freedom to decide at every frame what happens next. In the 'pose to pose' approach, the designer decides beforehand what will be contained in the sequence and which poses the character has to run through. In a first iteration, the designer might only draw the first

Figure 4.2: The animation principle 'anticipation' prepares the audience for a following action. In the example, the well-known Donald Duck by Disney prepares the audience that he will start running. (The Illusion of Life [231] p. 52)

and the last picture. In following iterations main poses are added an the remaining gaps are filled by in-between poses.

These two approaches serve as an important input to understand the work flow of a designer. If a designer is to be supported in the task to create expressive behaviors for a robot, design tools have to support this way of working.

Follow Through and Overlapping Action Characters need to stay in motion. A complete stop of a character will counteract the impression of a life-like character. The advantage of not being constrained by physical laws can also become a disadvantage, because no inertia rules need to be taken into account. This guideline also accounts for overlapping actions that are caused by the main action. For example, during a fast movement flexible body parts such as hair will perform a slightly different motion trajectory due to inertia. If such subtle cues are left out, the whole animation appears artificial.

In physical systems a moving mass cannot just be stopped in an instance. A follow through action after a fast motion is naturally required to decelerate the actuator. For concrete behavior design of robots, the consequence is that the dynamics of the hardware may

be exploited to create believable animation. Additionally, it can be derived that a character should not halt completely, even though no action is required from a functional perspective of an application. The resulting motion is only performed for the expressiveness of the device.

Slow In and Slow Out A traditional approach in animation is to start with important key poses of a character and later fill the in-between frames. However, it has been recognized that the in-between frames influence to a large degree whether a movement appears natural or machine like. A natural movement always consists of three parts, i.e., acceleration, constant movement and deceleration. This type of motion leads to an s-curve shaped trajectory if spatial progress is plotted over time, rather than a simple linear curve. Additionally, this principle allows to hold a certain position for some time, which for example supports the 'staging' or the 'follow through' design principles.

The important point that can be derived for robotic animation is that the in-between frames contain important information on the naturalness of an animation. Therefore, also the interpolations between keyframes need to be carefully designed instead of using simple linear approximation techniques.

Arcs If the spatial trajectory of a natural motion path is plotted, sharp edges or perfect straight lines hardly ever appear. Instead, almost every motion consists out of curve segments. Straight lines give a very mechanical impression and are attributed to machine behavior rather than to life-like characters. The Arcs design guideline supports the designer to place in-between frames.

This principle serves as an important input for low level behavior design of robotic interfaces. In a sense it conflicts with traditional control strategies of robots as these aim for shortest paths for efficiency. Additionally, uniform motion with as few actuations as possible insures long lifetime of the hardware. Instead, robots have to explicitly avoid straight line movements.

Secondary Action A secondary action can be used to accent a main action. As a guideline, the secondary action may never be more interesting than the main action. The main action carries the main message, but a secondary action can give a certain personality to a character. For example, for idle behavior the character mainly stands

In-betweens	Effect
0	Character is hit by tremendous force
1	Strong force (brick, frying pan)
2	Nervous tick
3	Dodging the brick/pan
4	Giving an order
5	More friendly order (come on, hurry)
6	Surprised/Happy (sees good looking girl)
7	Tries to get a better look at something
8	Searches for the peanut butter ...
9	Appraises, considering thoughtfully
10	Stretches a sore muscle

Table 4.1: Effects of animation timing in terms of number of in-betweens ([231] p. 65)

still, but on secondary action the character might still move due to breezing motion or eye blinks. This principle is strongly correlated to the principles of 'staging' and 'timing', because secondary action poses the risk of confusing the audience with too much information and is additionally tightly constrained by timing of the animation.

An example of how this principle can be applied to robot animation has been given by van Breemen [238]. He uses overlapping actions to turn a dull robotic movement into an expressive motion with personality.

Timing The timing of an animation conveys various physical attributes such as weight. The combination of timing, that it is the number of frames and the spacing, i.e. where on the canvas the action occurs, defines the dynamics of a motion. Timing can also contain information about the emotional state of a character. Table 4.1 relates timing by means of the number of in-between frames to possible effects that can be achieved.

Correct timing is an important attribute for believable animations. For example, if a robot gets hit, but the signal processing to compute an appropriate response takes too long, a completely different message might be conveyed as originally intended. If carefully used, different timings can convey important information about the state of a device.

For example, short time intervals can be used to convey that a security robot is alert, while long time intervals can be used to calm a user.

Exaggeration Exaggeration is a widely applied principle to stress a certain message. The guideline for an animator is to let a sad character appear even more sad and a happy character even more happy. For example, this can be achieved by adding a secondary action. A happy character raises head and arms next to showing a smile or additionally performing a little jump. Exaggeration may be applied as long as the resulting animation remains convincing.

For robot animation, exaggeration conflicts with the minimal movement policies of robots. From a technical perspective, a vacuum cleaning robot should perform its task as efficiently as possible and not drive additional turns. From an animation perspective, however, these extra turns can let the character appear happy and content with its task.

Solid Drawing A particular difficulty for 2D animation is that the animator has to draw the character from a certain perspective. In this process the animator has to imagine how a character appears in a 3D view. If these different views are not considered, then the different parts of the character are likely to be drawn identically. Such symmetrical postures appear very artificial and not life-like. The advice for an animator is therefore to avoid symmetrical postures. An example of this principle is depicted in Fig. 4.3

Even though for robot animation the animator has an overview of the 3D posture, he is even more in danger of creating symmetrical postures if using standard programming techniques. Unification and reuse of written code are common software design principles. Using these mechanisms, it is very easy to apply a given function not only to one, but also to a second actuator to save implementation time and increase maintainability. As a result, both actuators will perform exactly the same motion and do not give the impression of life-like behavior.

Appeal The last of the twelve principles addresses the overall design of the character. A heroic character with charisma has to convey these attributes in every motion. Commonly, these attributes are also reflected in the appearance of the character. Especially good and evil

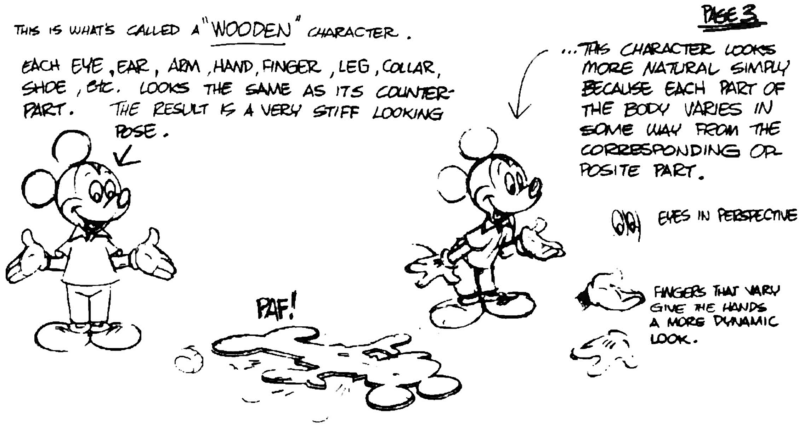


Figure 4.3: Solid Drawing animation design principle ([231] p.67)

characters, or main and minor characters of a story line are distinguishable through their design. In some sense, also evil characters have to be appealing to the audience.

For the design of applications of robotic characters, the appearance is usually fixed. For this reason, it is even more important to convey character traits and personality through motion patterns. However, in order to be believable, these must be consistent throughout the whole application. A continuously changing personality might evoke distrust, rather than the perception of dealing with a life-like character.

The presented animation principles originate from a time in which every frame was drawn by hand. Nevertheless, they still apply for modern computer tool supported animations. Many tools have been developed to support designers in their task (see Chapter 3.3.3), but automatic tools also introduce new challenges as illustrated in Fig. 4.4. In pose-to-pose animation, keyframes are animated first and in-between frames are filled in later. Computer tools can be used to automatically generate interpolations. The problem is that a simple linear interpolation can produce frames that do not necessarily make sense. The water drop splashes, even before touching the ground.
Current animation techniques can be classified in three classes [237]: 1) Prescripted animations, 2) Procedural animations and 3) Behavioral animation. Prescripted animations define look-up tables that assign a con-

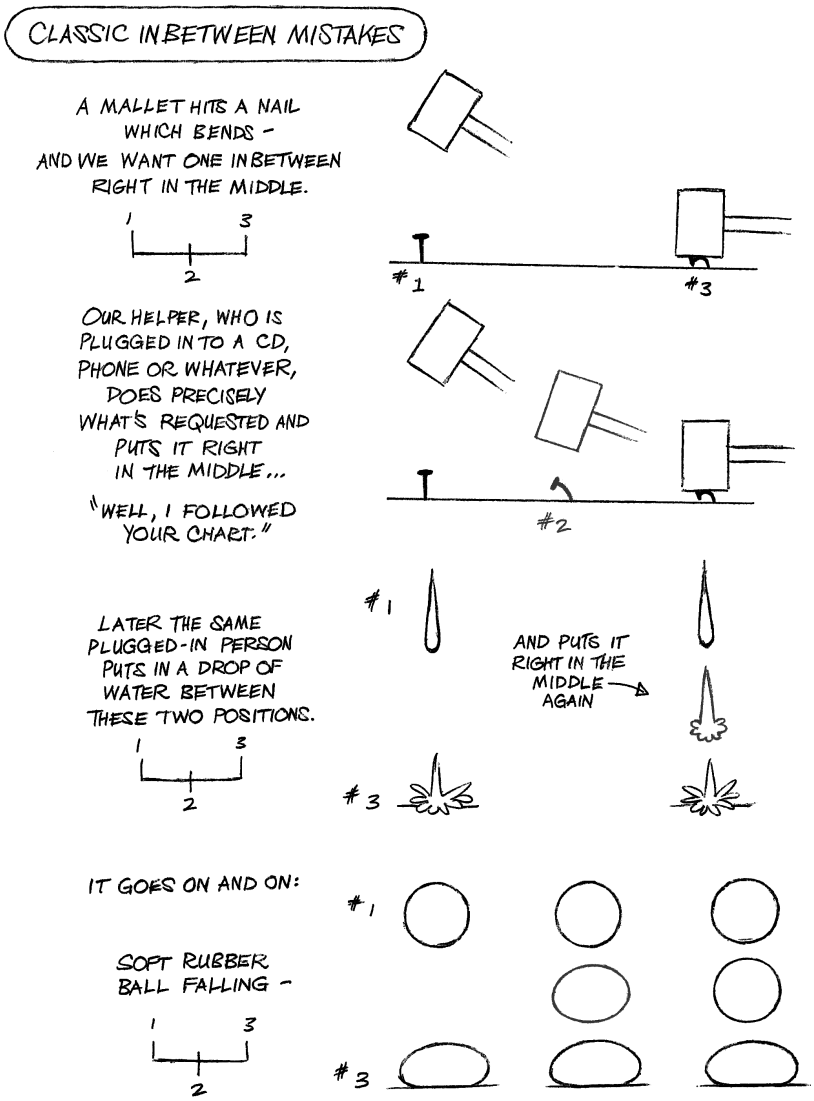


Figure 4.4: Common interpolation mistakes that are made by using linear interpolation without taking cause and effect relationships into account. ([253] p.88)

crete posture at a given time. Keyframe animations fall in this category. Procedural animations, in contrast, are evaluated by using equations. This enables them to simulate physical systems. Lastly, behaviors are also stated in equations, but the difference is that behaviors are reactive to the environment. For example during a walking behavior, the contact points between the foot and the ground are evaluated during run-time.
As one of the first development platforms the Philips Open Platform for Personal Robotics (OPPR) [239] offered the possibilities of keyframe animations as well as scripting behaviors. Equipped with this functionality, OPPR is well suited to address traditional animation principles. The existing OPPR system is discussed in the following section.

4.2 Open Platform for Personal Robotics

The Open Platform for Personal Robotics (OPPR) is a collection of software for programming personal robots [239]. In its first version it defined five software packages, namely 'Architecture', 'Workbench', 'Believability', 'Intelligence' and 'Connectivity'. However, in the latest version OPPR 2.0, only four packages are described 'Architecture', 'Animation', 'Intelligence' and 'Connectivity'. The two components 'Workbench' and 'Believability' have been combined in a single package that specifically addresses animation of robots.
The Architecture package contains a software library, 'Dynamic Module Library' (DML), for developing distributed applications. The package is a continuation of the middleware developed for the mobile robot Lino [135]. Additionally, it contains a 'Console' that serves as main entry point to configure and control the execution of a distributed application. The Animation package consists of two modules, the 'Animation-Editor' and the 'Animation-Module'.
The Animation-Editor is a graphical design tool for keyframe animations. It additionally contains a scripting editor for designing behaviors using the Lua scripting language. The Animation-Module provides and interface to the underlying robotic hardware and allows to render animations and behaviors on the robot. The underlying 'Animation-Engine' is described in [237]. At its core, it defines multiple animation channels in which animations can be loaded from an animation library. Every animation channel can contain a different animation and can be started and stopped independently from the other channels. The Animation-Engine has a global clock that defines the framerate by which commands are sent to the hardware.

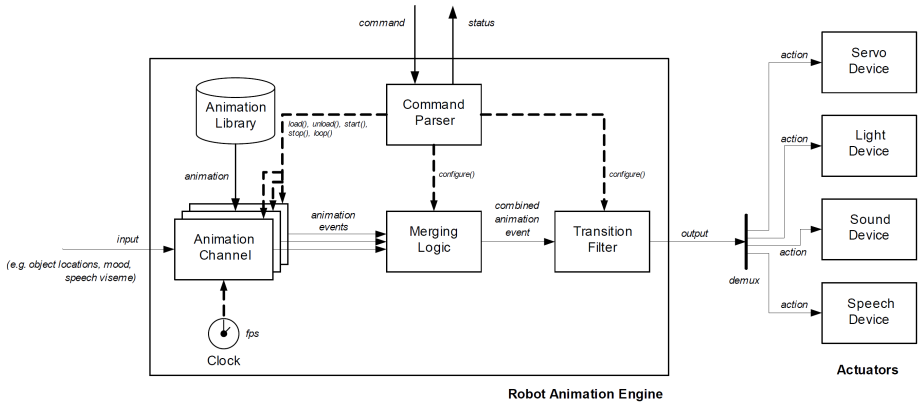


Figure 4.5: Overview of the OPPR Animation-Engine. (Source: [237])

Furthermore, the Animation-Engine' defines a merging logic that controls how multiple channels are combined. An overview of the Animation-Engine is depicted in Fig. 4.5

The Intelligence package offers a software module that allows to script behavior for the robot using the Lua scripting language. The Lua language has been extended with a set of functions that define an API to accesses the functionality of the OPPR framework, for example DML and the Animation-Module.

The Connectivity package, contains a predefined software module to access the Internet e.g. a Mail-Module that allows sending and receiving of emails. For other connectivity, the developer can rely on third party libraries or on low level socket communication.

A major addition that was introduced by the OPPR framework is the use of scripting technology to animate robots [240]. The role of scripting technology is discussed in the following section.

4.3 Scripting technology

Scripting languages were introduced as an alternative to software technologies such as software components and middleware for the development of applications. All of these concepts aim to support the development process. For example, software components define reusable and independent software modules. Modules such as image or speech recognition can typically be reused. Nowadays, multiple software packages exist that combine

these modules with a middleware API. Middlewares allow for easy development of distributed applications. In the field of robotics research, these software packages have been coined 'robot middlewares'. EPRS[1], the Open Robot Controller Architecture (ORCA) [156] and Microsoft's Robotic Studio[2] are examples of robotic middleware that provide special tools and libraries that facilitate the application development for robots. However, especially in the field of robotics, there are still plenty of unresolved issues [168]. For example, current middleware systems provide only limited support for general and frequently used services. Furthermore, there is a lack of tools for providing high level abstraction interfaces. In consequence, application developers always have to start from lower level primitives.
Scripting technology addresses the problem of abstraction and offers more flexibility to the application development process. Scripting languages are special purpose domain languages that are streamlined for rapid prototyping, iterative development and high flexibility. Therefore, scripting technology is one of the fundamental techniques that have been adopted in this technological design. In particular, the scripting language Lua[3] has been used.

Lua Lua is an open source, very light-weight imperative scripting language. It has a syntax similar to that of the programming language C [110]. The interpreter and compiler have been ported to embedded platforms such as mobile phones or the Lego®Mindstorms. The full reference interpreter with preloaded standard libraries is about 150kB in size. Lua can directly run on any platform that supports the POSIX standard. Lua is widely used in the gaming industry and for web applications, because it is small, easily embeddable to host applications and easily to extend through a simple C-API.
Lua is a dynamically typed language with proper lexical scoping. Lua functions are first class values that can be assigned and passed as arguments like any other type in Lua. The Lua compiler produces bytecode that is interpreted by a Lua virtual machine. The compilation is transparent to the language and available as a command in the language itself to evaluate and run a chunk of code. Therefore, Lua can be used to generate and execute scripts at run-time. Even though Lua is an imperative language, it can also be used following an object oriented or functional programming

[1] http://www.evolution.com
[2] http://msdn.microsoft.com/robotics/
[3] http://www.lua.org

paradigm. In particular, Lua supports object orientation with its table mechanism, which supports information hiding and inheritance. The functional programming paradigm is supported by proper tail calls for recursive functions. Unlike in imperative languages, no stack overflow is produced. In summary, scripting technology is a very powerful technique that can be applied to efficiently create expressive behaviors for robots.

Part II

A software architecture for social robots

Chapter 5

Design framework architecture

The previous chapters introduced the design domain for creating applications with a robotic interface and the related design challenges. Based on this analysis a high level architectural overview for a unified Social Robot Design (SRD) framework will be developed in this chapter. The goal of this framework is to support application developers to create applications that utilize the expressive interaction capabilities of a robotic embodiment. First, the design approach and relevant concepts from computer science and software engineering are introduced. In order to realize a unified development framework, the concepts of the design domain are modeled in a domain overview. Secondly, the main sources of requirements are identified and elaborated. The chapter concludes with a high level overview of the architecture of the SRD framework, which is further refined in Chapter 6 and Chapter 7.

5.1 Design approach

A unified design framework enables application designers with different fields of expertise to develop applications for interactive robots on a high level of abstraction. A similar interdisciplinary collaboration takes place in the gaming and movie industry [113]. Nevertheless, the development environments for robotic applications are still fragmented. One of the reasons is that they have some unique requirements related to the problem domain, which preclude simply using the same tools.

In order to provide the envisioned functionality, an appropriate software architecture needs to be developed and the main concepts and requirements need to be identified. To this end, the software architecture serves two main goals. First, it plays a crucial role in the software development process and second, it helps in analyzing the requirements, the organization of constructs and in understanding the design domain itself.

5.1.1 Design knowledge from software engineering

In the domain of software engineering it has been recognized that the design of a proper architecture is a key element for every reasonably sized software project [16]. The quality of the architecture often decides on the success or failure of the project [16, 140]. Furthermore, a software architecture is not only a plan to implement software, just like an architectural plan for the building of a house, it also holds key information about the domain that is modeled [16, 227]. The decomposition of a domain in relevant constructs breaks the domain down into small implementable pieces and reveals the structure of the domain.

5.1.2 Software architectures

The process of defining an appropriate software architecture has received great attention, not at least because of the high business relevance and economic consequences that are connected with the quality of software architectures [16]. However, until now no universal architecture that ensures the success of a software project could be found [140], even though great advances have been achieved in identifying resurfacing design patterns and domain specific template descriptions. This scattering across domains is reflected by the multitude of definitions that have been proposed for software architectures. Most commonly, software architectures have been used as a notation for specifying functionality and organization. In this sense, a software architecture gives an abstract overview of an analyzed system and guides its implementation. Bass et al. provide a working definition for software architecture:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. (Bass et al. [16] p. 3)

This working definition captures already the structural knowledge that is communicated with an architecture. However, it does not capture the reasons for design decisions and the involved design process. A more detailed definition for a software architecture is given by Booch, Rumbaugh and Jacobson in 2005, who define software architecture by

> The set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization—these elements and their interfaces, their collaborations, and their composition. Software architecture is not only concerned with the structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns. (Booch et al. [23] p. 452)

This definition captures the various dependencies of an architecture by which it mirrors its ecologic surrounding. It also highlights the value of a software architecture in terms of its communicated design knowledge.
Furthermore, it provides multiple views on a system. For example, the system can be analyzed at different levels of abstraction, including static and dynamic aspects, or in terms of business relevance. These different views provide different information on the design domain. A use case analysis, for example, might reveal the stakeholders and basic building blocks, but does not tell anything about how a specific feature is going to be realized. In the same manner, a structural organization and hierarchy of elements also need to be accompanied by a description of how these elements interact with each other.

5.1.3 Architecture notations

In software engineering multiple notations are available to denote different views of a system. These notations can be classified by the amount of formalism they define. For example, a general drawing tool can be used to draw basic shapes to represent the structure of a system, without a formal definition of the building blocks. Another possibility is to use a generic mind mapping tool to set basic concepts into context. A mind

mapping tool defines a neighborhood relationship, but it is not formally define of what kind this relationship this is. The main advantage of such approaches for system architects is that they are not constrained by the notation for his modeling of the architecture. However, this flexibility is at the cost of ill defined semantics. In contrast, a specification language has a very well defined syntax and semantics, but constraints the architect in his modeling. Most commonly, a trade-off has to be made. The choice for a certain notation is determined by the purpose for the modeling.
Multiple notation formalizations have been proposed for architecture design [16]. For this study, the Unified Modeling Language (UML)[177] has been selected as it combines a well defined semantic with the communicative power and design freedom of a graphical language. UML is the most widely used graphical modeling language for object oriented software modeling [23, 205, 140]. A definition of UML is given by Larman as:

> The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schema's, and reusable software components. (Larman [140] page. VIII)

UML has been defined by the Object Management Group (OMG)[1], an international non-profit computer industry consortium. For example, the language defines standard elements for modeling static components, applications structures, data structures as well as dynamic entities such as data flow, use cases and business processes. UML defines six types of diagrams to capture structural relationships (class diagram, component diagram, composite structure diagram, object diagram, deployment diagram, and artifact diagram) and five types of diagrams that capture dynamic aspects (use case diagram, sequence diagram, communication diagram, state diagram, and activity diagram). The most recent version of the specification is UML 2.2 [177], but a working document for version 2.3 is already available for download. With the introduction of version 2.0 the OMG divided the specification in two parts named 'UML infrastructure' and 'UML superstructure'. The UML infrastructure defines core language constructs for UML while the UML superstructure defines user level constructs. More

[1] http://www.omg.org

specifically, the infrastructure library defines a reusable meta-model and meta-language that can be used to extend UML.
UML has been successfully applied in a multitude of software projects, ranging from small scale projects with only one developer to complete enterprise solutions [140]. UML defines high level constructs that are flexible enough to not constrain the modeling process, while at the same time defining a formal semantic that can be used for automatic code generation. However, this generative power of UML and its complexity make it difficult to use [75].
At present there are several of commercial and open source software tools available that support software modeling using the UML notation. The majority of these tools provide comparable functionality with respect to basic modeling capabilities, but they differ greatly in additional features, for example with respect to automatic code generation, supported programming languages, automated documentation or support for large distributed teams. In this thesis, the modeling software 'Enterprise Architect 7.5' developed by Sparx Systems[2] was used. The software tool fully supports the UML 2.0 standard and is listed as one of the official packages on the OMG specification website [3].

5.1.4 Rational unified process

Throughout this thesis an iterative agile software engineering approach following the Rational Unified Process was followed [136, 23, 140].
An iterative design approach considers dynamic refinement of requirements during the development process instead of assuming that all requirements are fully specified before the implementation starts. In the literature, common reasons for the failure of software project have been analyzed with the conclusion that inflexibility is one of the main reasons for failure [140]. For example Rauterberg and Strohm pointed out that it is faulty to assume that the client of a software project is able to specify all requirements in the beginning [197]. They proposed an iterative software development cycle, arguing that it has the potential to significantly lower the costs of a software project and simultaneously increase the software quality.
In the domain of research, an iterative development is even more relevant, because new knowledge becomes available only throughout the process. To cope with this situation an iterative agile software design approach

[2] http://www.sparxsystems.com.au
[3] http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML(Date: July 2009)

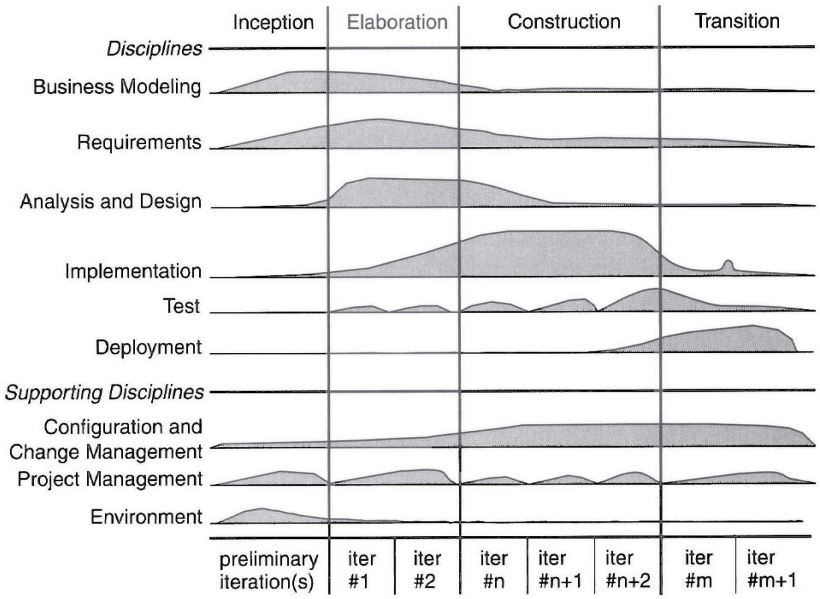


Figure 5.1: Rational Unified Process – Phases and disciplines (Source: [23] p. 34)

defines short development cycles in which only a few prioritized features are implemented. This process is formalized by the Rational Unified Process (RUP).

RUP combines best practices from multiple software design approaches including a 'waterfall process', 'evolutionary software development' and 'extreme programming'. The RUP defines four discrete phases: 'Inception', 'Elaboration', 'Construction' and 'Transition' as depicted in Fig. 5.1. In the Inception phase a common ground is established for all stakeholders of the system to get agreement on the requirements. In the elaboration phase, the problem space and key risks are analyzed and an architectural framework to address these problems is established. It is very common that the architectural description and use cases for the system are given by UML diagrams. In the construction phase these artifacts are used as input to implement and test essential parts of the system. At the end of the construction phase stands a working system that can be delivered to customers. The actual delivering process is carried out in the transition phase. Every phase contains multiple iterations that encompass activities from the following nine defined disciplines:

Business modeling Understand and establish a relationship between business goals and software engineering

Requirements Collect and analyze requirements from involved stakeholders

Analysis and design Defines relevant views on the system and an agreement on how it will be realized

Implementation Software engineering and code generation

Test Test all defined constructs, relationships and functionality

Deployment Release and installation

Configuration and management Handles requests for changes in the process and maintains the current status

Project management Organizes project execution in terms of working processes

Environment Maintain the required infrastructure

RUP is a highly customizable process and can be adapted to the specific needs of a software project. The focus of this research project is on the inception and elaboration phase, because in those two phases the relevant knowledge on the design domain is gained.

5.1.5 Requirement engineering

Requirements define a contract between the stakeholders of a software system on the functionality that the system has to provide. Next to the functional requirements there are also more generic requirements such as performance, reliability, security, compatibility or flexibility.
In the literature on software engineering, the definition of requirements has been recognized as a crucial element that determines the success or failure of a software project and consequently the product [16, 227]. However, no single methodology or process has been found that guarantees complete and correct specification of the software. Instead, various processes have been defined that support the identification of requirements and multiple notations have been developed to formally specify them. Notation methods range from natural language use case scenarios that describe the behavior

of the system in a particular situation to fully formalized requirement specification languages.
Next to the engineering problem to identify and specify the requirements of a system, there is also a more fundamental problem. As Rittlel and Webber pointed out, there exists a class of problems for which a problem specification and appropriate solution can only be given parallel to the software development process, rather than beforehand [199]. They coined this class of problems 'wicked problems'. In order to be able to fully specify all requirements beforehand, the target domain has to be fully known. In the case of a design framework for social interactive robots this knowledge is unavailable, as the design of social interaction is itself a topic of research. For this research an integrated approach has been chosen based on use case scenarios and theoretical analysis. The input for this approach has been taken from literature analysis, user stories from the OPPR community and knowledge gained during the development process.

5.2 General use case scenario

As the first design step of the Social Robot development framework a general use case scenario was developed. This overall use case of the system is depicted in Fig. 5.2. In this scenario, an application developer uses the Social Robot Development framework to create an application that encompasses a robotic user interface. The role of an application developer is a generalization of developers that approach the development from different fields of expertise. As an example, three specific roles are specified including an animation designer, an interaction designer and an application logic designer. These three domains have been derived from the background descriptions that have been given in Chapter 3. These role descriptions can be used to derive appropriate interface metaphors for the tools that are provided to them by the SRD framework. Interestingly, a 'metaphor engineering' as described by Rauterberg and Hof has not yet become a standard step in the process of user interface engineering [195]. Instead, the interfaces are evaluated with user studies.
An animation designer is skilled in designing animations. He has expertise in conveying intended messages through carefully designing expressions by means of light, sound and motion. Furthermore, he has a background in cinematography and is skilled in realizing creative ideas with standard tools from the field of animation. The animation designer has usually a visual style of working.

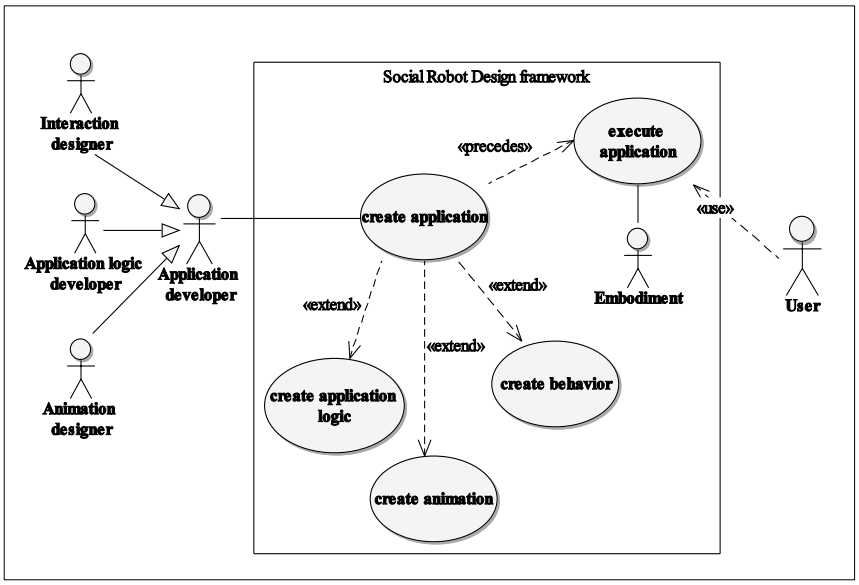


Figure 5.2: Use case of the Social Robot Development framework

An application logic developer usually takes a more formal approach. He is skilled in concepts of computer science and is able to express the intended behaviors of a robot in formal programming languages. He is familiar with common tools such as editors, compilers and debuggers. For example, the development of signal processing and artificial intelligence algorithms fall in his domain.

Lastly, the interaction designer is skilled in creating the interface and interactions for the user. He has experience in user centered design methods, knows common interface concepts and is aware of users' abilities and expectancies. He defines use case scenarios and is able to specify courses of interaction.

These interdisciplinary roles cooperate to create an application. The three corresponding high level tasks in the model are detailed extensions of the overall task to create an application. A simple application might only contain an animation, for example for entertainment in amusement parks. For more complex and interactive applications, more intelligence needs to be created, which requires different expertise.

Having described the overall use case scenario of the SRD, the following section defines key requirements that have to be meet by the framework. These requirements were derived from literature and design guidelines for

creating robot interfaces as well as from feedback that was given by the OPPR community.

5.3 Requirements

This section identifies the major requirements for development of the SRD framework. First the classes of requirements that are most relevant for the design process are selected from literature. Second, four major categories of requirements for the SRD framework that are derived from the overall use case scenario and are analyzed in greater detail.

5.3.1 Classes of requirements

In the field of software engineering, the overall set of requirements has been categorized in two major classes, 'functional requirements' and 'non-functional requirements' [227]. Functional requirements define characteristics, i.e., features and functionality, which the system should provide. In particular, they define a relationship between input and output of the system. Non-functional requirements are more general constraints on the system. While functional requirements usually specify only parts of the system's behavior and are very domain and application specific, non-functional requirements are often applicable to the whole system [227].
For the development of the SRD framework, most of the depicted non-functional software requirements play a subordinate role, because they are mainly relevant for industrial applications and not for the requirement engineering process. Furthermore, some of these non-functional requirements cannot be attributed to a single software feature, but are emergent attributes than can only be measured from the interaction of all software components. Hence, the focus has mainly been on the identification of the functional requirements that are relevant for robotic application designers. Whenever requirements of the non-functional categories are essential for the proper functioning of the software, this is made explicit. Examples of these categories are usability, efficiency and interoperability.
For the design of the SRD framework the following four major classes of requirements have been derived from the overall use case of the framework:

- Application designer requirements
- Development process requirements
- Domain requirements
- Framework requirements

Application designer requirements capture all requirements originating on the side of the application developer. Therefore, these requirements deal with the different domains of expertise of application designers and their different ways of working in terms of terminology and basic concepts. Development process requirements capture the demands on the software to create aspects of the application that are common to a development process. For example, rapid prototyping and debugging facilities fall in this category. Domain requirements are derived from the specific application domain of supporting developers of social robotic interfaces. From this specific application domain requirements on the development environment, for example with regards to the deployment of an application can be derived. Finally, there are additional requirements from the perspective of the framework itself. For example, the overall software architecture has to be flexible so that the framework can be adapted easily as soon as new knowledge on the design of social robots becomes available. In the following these four classes of requirements are analyzed in detail.

Application designer requirements

An application designer has the goal to develop an application to be used by a user. He creates a basic idea about the functionality and target group. For example, the designer might use the balanced application design approach as described in Chapter 3.1 in order to verify a main interest in the application. If the application idea is clear, the designer needs to specify in greater detail what aspects need to be realized. For this he might adopt an iterative design approach as described in [162].
As soon as the basic concepts are defined, the designer wants to realize the ideas as easily and quickly as possible. Regardless of the process that is used to specify the application, the designer needs means to test and realize the ideas. Derived from this general goal to simplify the application development process, a number of requirements on the side of the application developer can be derived.

RQ-1.1: *Increase the level of abstraction for developing applications of robots*

First of all, the framework needs to increase the level of abstraction. Low level tools such as C++ or even Assembler on embedded hardware provide only a very low level support to program the robot. Especially for developing applications which include complex interactions with the user, it is impractical to operate on motor levels. This requirement ensures that

the designer will be provided with tools that are more appropriate than low level programming tools. From this demand also important design decisions for the architecture can be derived, namely that the architecture needs to be extensible with new higher levels of abstraction as soon as the design knowledge for these levels becomes available.

RQ-1.2: *Provide tools that offer familiar concepts according to a field of expertise*

In the overall use case scenario, it was indicated that application designers may have different backgrounds and different fields of expertise. Therefore, they are familiar with different basic concepts. For example, for an animation designer the concepts of a 'keyframe' or an 'in-between' are meaningful, which is not necessarily the case for an application designer with a computer science background. In turn, the animation designer might not be familiar with the concept of classes, attributes or interface methods. For this reason, application designers with different fields of expertise will approach the task differently. To support these different views, the SRD framework needs to provide different tools that reflect different backgrounds. It is an interdisciplinary effort that requires expertise from various fields, including animation, computer science, human-machine interaction and psychology. The framework needs to be amenable to these different design backgrounds.

RQ-1.3: *Provide multiple views on the design problem*

In Chapter 3, it has been shown that multiple design challenges need to be solved in the process of creating a robotic interface application. The essence of different design challenges can best be captured when approached from different points of view. For example, to capture the expressiveness of the robotic character, the designer might think in terms of animations and symbolic postures. If the functionality of a vacuum cleaner must be described, it is more appropriate to use logic and behavioral rules. However, these two views are not independent from each other. As soon as the robot moves differently to accomplish a task, it will also convey a different message to the user. Different views on the design problem will help the designer to keep track of these dependencies. For example, the framework might define a functional view in terms of floor coverage and an emotional view in terms the user's interpretation of the motion pattern.

RQ-1.4: *Different views need to be synchronized*

Directly related to the requirement for different views is that these views need to be synchronized. This requirement seems to be rather obvious, but it has important consequences for the communication architecture of the framework. If not explicitly demanded, in an agile design approach, synchronization is omitted until it is explicitly demanded. The OPPR tools have shown that automated synchronization is an essential feature from which the development process greatly benefits. A particular example is the posture editor which synchronizes the motor positions of a keyframe editor with a virtual representation of the robot. Synchronization between different views is also one of the essential features of computer animation tools such as Maya[4] or XSI[5].

RQ-1.5: *Provide easy switching between different views*

In addition to providing different views on the design problem, it must also be easy and seamless to switch between these views. The consequence of a change in one view must be easily accessible also from another view to prevent missing important dependencies. Additionally, as experience during the development of application has shown, it is not uncommon that only small changes need to be made in one view, before returning to another view. In the existing OPPR framework for example, the designer needs to switch between text editor to code the application logic and animation editor to create expressive behaviors. Especially in the situations when spoken utterances of the robot needed to be synchronized with the animations this required quick switching between text and animation editor. This problem will become even more evident, when more views become available, for example an emotion editor or sound editor.

Development process requirements

Additionally to the requirements specific to the designer's point of view, experience showed that the framework needs to support a few general development process related functionalities. These requirements have mainly been gathered through the development of OPPR and interaction with the iCat community, which served as important input for the development of the SRD architecture.

[4]http://usa.autodesk.com
[5]http://www.softimage.com

RQ-2.1: *Allow for rapid prototyping*

In [162] it has been shown that an iterative design approach based on a personality profile is a valuable addition to the processes of robot application design. A software framework for developing robot applications needs to support these general insights.
From a designer perspective, this requirement is motivated by practical experience with the OPPR framework. A particular feature that supports rapid prototyping is the virtual representation of the robot. An application can therefore be implemented on the desktop computer, before it is evaluated on the hardware. The virtual simulation of the robot has been one of the most well received features and most useful features in the iCat community as exemplified by the following quote:

> I was looking at the slides from the OPPR presentation by Philips in the BOON forum, and I noticed that there were screenshots of what seems to be an animated iCat running in a window. Is this something that's included in the distributed software? if I run without the cat plugged in, the animation module says its using "Virtual iCat", but there isn't any actual window anywhere.
>
> If this actually exists, it would be great if it were included; it would help me stop annoying my office mate every time I start experimenting with the cat, for example. (posted on Feb. 15 2006, User: mef)

RQ-2.2: *Allow to easily adapt an application*

Closely related to an iterative design approach is the possibility to easily adapt and test an application. In an iterative development process, the resulting application is continuously tested and redefined. Throughout the development process, the complexity of each of these modules increased in terms of more files that had to be loaded, more initialization that had to be done and more functions that needed to be tested. It proved essential to be able to quickly change an application and retest the new version. Long-winded interruption in this process significantly slowed down the development process. This led to the requirement that an application had to be easily adapted.

RQ-2.3: *Support direct testing application artifacts*

Naturally, during a development process errors are made. In its original version, OPPR had only limited debugging support. This has been one of the major shortcomings of the OPPR system. For simple applications, it had been reasonably easy to maintain animations and behaviors manually. However, as soon as applications were growing in complexity, several side effects of interactions between different application modules and animations led to unplanned behaviors. For example, if a behavior unloaded an animation channel, while another still relied on it, the system showed unanticipated behaviors or the application stopped working altogether. These errors were difficult to track down, because of the asynchronous and non-deterministic nature of the system. Based on these experiences, debugging facilities are added as an essential feature for the framework.

RQ-2.4: *Development environment must be independent from the deployment environment*

This requirement is motivated both from a technical and a designer point of view. First of all, from a technical perspective, a robotic platform usually has very unique specifications, which are not suited to support application development. Secondly, from a designer's perspective, a development framework for creating applications should not be bound to a specific embodiment. In analogy, high level programming language abstract from the underlying hardware. It would result in an intractable overhead, if for every new processor generation a new programming language would have to be learned. In the field of robot programming environments this has been recognized and an increasing number of environments have been published that allows to program different types of robots with the same tools.

RQ-2.5: *Create reusable application artifacts*

In the course of this research, multiple applications for the same embodiment have been developed, for example a 'waiter application' (Chapter 3.1), a chat-buddy (Chapter 3.1) and a tutoring application [211]. Several further applications have been contributed by the iCat research community and have been published in literature [238, 96, 95, 102, 151]. During these developments, several patterns surfaced. For example, in the majority of iCat applications basic animations for greeting, saying goodbye, answering "yes", answering "no" or pet like behavior, such as purring have been created. Reusing these application artifacts significantly shortened the development time. If software reuse for the development of social robots takes

a similar trend as in general for software engineering, then it could increase the software output exponentially [137]. This trend shows the importance of being able to develop reusable artifacts.

Domain requirements

The goal of the SRD framework is to be a special purpose solution to develop applications for social robot interfaces, rather than to be a general programming environment. From the intended application domain, further requirements can be derived.

RQ-3.1: *Provide an interface to control robot embodiment*

Most obviously, the framework has to support the control of robot hardware. Given the variety and non-uniformity of available robots, this is not a trivial task. This requirement insures that the architecture provides general means not only to control and program physical robots, but also virtual representations such as virtual screen characters.

RQ-3.2: *Provide tools that support development along the design dimensions of naturalness, adequateness and development over time*

Once the developer has access to the robot, he faces the challenge to create an application that is meaningful for the user. In Chapter 3 several design challenges have been discussed. Equipped with basic control over the hardware, the design space is only constrained by physical limitations of the hardware. In Chapter 3.3 three major design dimensions have been identified that constrain the design space and provide helpful guidelines in the design process. To actively support the design process, the framework should offer tools that allow to develop along these design dimensions.

RQ-3.3: *Support the development of real-time systems that interact with a user*

According to Sommerville, a real-time system is:

> A real-time system is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced. (Sommerville [227] p. 340)

The overall use case scenario depicted in Fig. 5.2 on page 85 illustrates that the interaction between the application and the user constitutes a particular focus for the framework. In most of the cases, the robot has to react to the

user's input. Even though the degree of interactivity between the user and robot depends to a large degree on the application context, the possibility to design for interactivity has impact on the design of the architecture.

RQ-3.4: *Support research in the field of HRI with Wizard-of-Oz applications*

This requirement resulted from the domain specification of human-robot interaction and the need to acquire and validate knowledge through user studies. This has been one of the major pillars at the foundation of the iCat project from Philips Research and many of the universities that develop interactive applications with robots. The major goal of these ongoing research efforts is to understand fundamental rules and exploit the unique interaction capabilities of robots to provide easy to use interfaces. A framework for developing application including a robotic interface should be flexible and constructive. This requirement therefore has consequences on the features such as offering the possibility to perform controlled user studies.

Framework requirements

In this category major requirements that resulted from a technological point of view from creating a supportive software architecture are captured.

RQ-4.1: *The framework need to be extensible with new components*

In the field of human-robot interaction, ongoing research effort is spent to unveil basic interaction principles. As soon as new knowledge becomes available it needs be integrated in the architecture. Furthermore, extendibility is a fundamental requirement for an agile software engineering approach. Therefore, the architecture cannot be modeled as a closed system, but needs to be open-ended and extensible. This requirement is in its core a non-functional requirement that cannot be attributed to a specific part of the software. However, it must be explicitly adhered throughout the whole design process.

RQ-4.2: *Allow third party developers to contribute new components to the architecture*

Next to the requirement of being extensible, it is also of particular importance that the framework is accessible for both component developers and

component users. As outlined in Chapter 3, the development of applications for a robotic user interface is an interdisciplinary challenge. For this reason, it is important to streamline the knowledge transfer between different fields of expertise as far as possible. For example, one track of research might focus on expression of emotions [95]. As soon as updated models become available, it should be easy not only to provide the knowledge to a broader community, but also to make it directly usable in terms of software. This influences the design of interfaces that are publicly available.

RQ-4.3: *Make integration of expertise easy*

Component interfaces are not only important for component developers, but also for the designer. The integration of different components in a robot software architecture is not trivial [168]. Even though multiple reusable artifacts are available (e.g., speech recognition or text to speech generation) it often takes considerable effort to utilize them in an application. Often these modules require a very technical understanding on how these algorithms work, which hampers the development process. In consequence, it is not a matter of providing the functionality, but it also has to be made easily usable by developers who are not experts of a particular domain.

RQ-4.4: *Align with existing development platforms*

Another technical requirement that concerns the integrability of the framework is that it most likely will be applied by developers who already have an infrastructure available. A disruptive solution that does not integrate in existing environments would render existing software unusable. This requirement makes this integration problem explicit and poses another challenge on the interface definitions and the libraries that are used for the framework. For example, some research laboratories operate using Windows operating system, while others work in a Unix environment. In consequence, it prohibits the use of libraries or features that are only available for one platform, at least as far as the central architecture is concerned.

RQ-4.5: *Integrate with OPPR tools*

In line with the requirement to integrate with existing platforms, a particular environment that the SRD framework has to integrate with is the OPPR development framework that has been described in Chapter 4.2. OPPR already includes several prototyping tools that are relevant for the development of robot applications and is available to the iCat research community. It therefore already defines interfaces that are not local to one

development environment, but are available to a larger community. This requirement denotes the relevance of OPPR for the SRD architecture as a particular development environment the framework has to integrate with. The integration with OPPR can be seen as an exemplary case to demonstrate generality of defined interfaces, to which also other frameworks can connect.

5.4 Global architecture

Defining requirements is only a first step in creating a software architecture. Requirements do not determine an architecture. An architecture denotes the result of technical design decisions. In the following basic design decisions for the overall architecture are motivated.

In the field of software engineering several template architectures have been developed to categorize the type of a system. Commonly used characteristics to typify an architecture are overall system organization, component model, execution type control organization and communication organization. Examples for the overall system organization are layered architecture, client-server models, and repository centered architectures [227]. The used decomposition style is independent of the overall system organization. Object-oriented decomposition and functional-decomposition are two widely used component models. Another design dimension is the type of execution, e.g., whether different components are executed in sequence or whether they run in parallel and whether a synchronous or asynchronous execution model is chosen. Furthermore, the designer may choose between different control styles. In a centralized approach, a single main program is responsible for overall execution in the architecture. However, also a decentralized control structure might be chosen in which control is localized in a component that collaborates with other components through messages.

The above architectural design decisions are very general models that can be applied in a variety of application domains. For this reason, more specific reference architectures have been defined that are more suitable to the needs of a particular application domain. A layered architecture for communication protocols is an example of such a reference architecture [227]. Many other reference architectures have been established, for example distributed software systems, including client server architectures, software bus systems, peer-to-peer networks and service-oriented architectures.

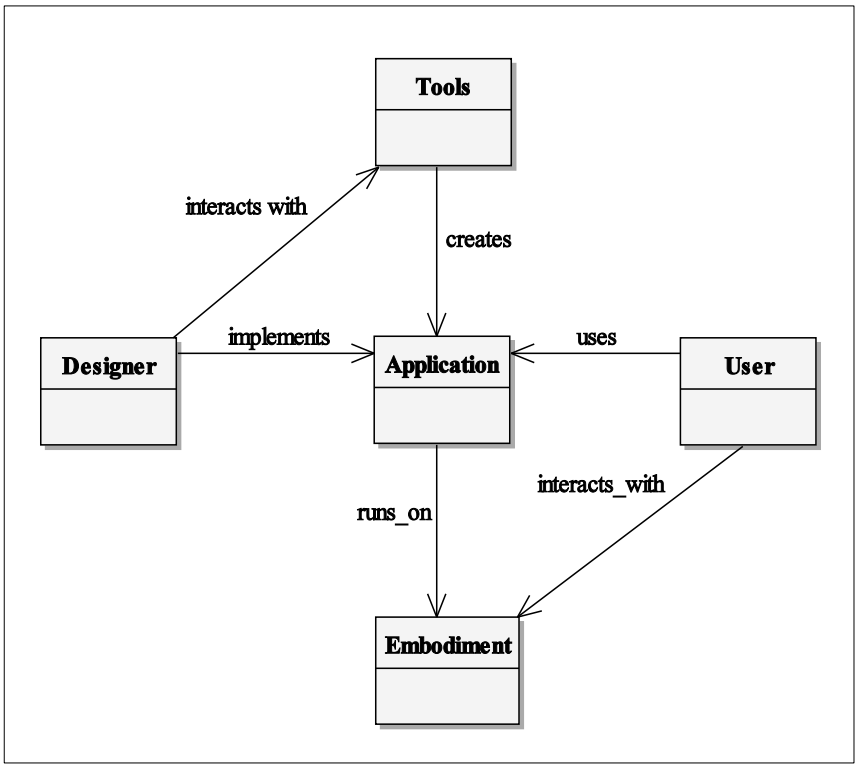


Figure 5.3: Design domain overview with an application as the central concept.

The different software architectures illustrate the variety and complexity of conceivable architectures. In the following the major design decisions for the general architecture of the SRD framework are motivated.

5.4.1 Architecture design

As a first step in the design process the overall design space has been decomposed to identify central constructs. The main central construct to the architecture is the concept of an application. This central role is visualized in Fig. 5.3. The overview shows a symmetric structure of the design domain that is derived from the overall use case. On the one side, a designer interacts with the SRD design tools in order to create an application. The application, in turn, runs on an robotic embodiment that serves as interface to a user. From this perspective, the SRD framework and the embodiment are connected through the concept of an application. The same abstraction

can be made for the roles of designer and user. Both relate to the application at a similar level. The designer implements an application that is going to be used by the user. The user, on the other hand, has demands on the application that have to be met by the designer. Application requirements and implementation are manifested in the general concept of an application.
From this global view on the application, the most important fact that can be derived is that development and execution are only loosely coupled and are not directly linked. Consequently, in the design of the software architecture for the overall SRD framework a separation can be made between a 'Development-Environment' and an 'Execution-Environment'. These two environments are linked through the concept of an application and share related components.
This separation has several advantages. For example, it allows that the development environment can be formally deployed on a different hardware than the execution environment. In the case of developing applications for robotic user interfaces, the development usually takes place on a desktop PC, while the application is rendered on an embedded hardware. Furthermore, it has an impact on the component decomposition, because different communication protocols may be used during development and execution.
Notwithstanding this separation, both environments share important commonalities, that is they complement each other in a well defined area. For example, both environments share similar concepts such as animations and behaviors. This connection becomes important if the designer needs to debug or evaluate an application. Additionally, shared concepts reduce the effort to maintain two separate environments, because they need only to be coded once.
The main conclusion from this discussion is that the overall architecture can be separated in two major parts, a Development-Environment and an Execution-Environment. The following chapters will discuss both environments in detail.

Chapter 6

Development Environment

In the previous chapter a general separation between *Development-Environment* and *Execution-Environment* has been introduced. In this chapter the architecture of the Development-Environment is developed. The main task for the Development-Environment is to support the designer with appropriate tools.

The Development-Environment represents the system perspective from an application designer point of view. The design space is analyzed and together with the requirements of the previous chapter an architecture for the Execution-Environment is derived. The major challenge to overcome is that crowd of currently available technologies to design applications for robots is 'fragmented'. Fragmented in this sense means that it is very difficult to bring different tools together. However, in the previous chapters it has been shown that multiple roles participate in the design process of a robotic application. For example, in the category of application designer requirements, it was demanded that the designer is equipped with tools that provide an appropriate level of abstraction. In consequence, the SRD framework has to provide tools that allow editing an application for example on the level of general interaction rules as well as on low-level behavior design. These different tasks are addressed in the following. First, a general reference architecture is developed, which is subsequently elaborated to address specific editing tasks. The general architecture defines a unified component interface and communication model. In this framework, concepts such as an editor, preview and logging facilities are integrated.

6.1 Structural decomposition

For identifying an appropriate structural decomposition, the design domain (visualized in Fig. 5.3 on page 96) has been modeled in greater detail by taking the formulated requirements into account.

6.1.1 Reference architectures

For identifying basic components and relationships, overall reference architectures have been selected from literature [140, 227, 16]. Given the application domain of creating social robotic interfaces, two template architectures were considered to be relevant for the architecture: *event processing systems* and *language processing systems* [227]. These two architectures are introduced in the following.

Event processing systems An event processing system consists of identifiable components between which messages are passed. Messages might contain plain data as well as commands to be executed by a component. A special case of event processing systems are editing systems. Editing systems are mostly single-user systems that are constrained by time to provide immediate feedback to the user. This constraint must be satisfied also during heavy loads of the system to keep the interface responsive. Another characteristic of editing systems is that they usually have long sessions in which the user produces data of some type. Many PC applications such as word processors or games are event processing systems. A simplified example of an architecture for an editing system is depicted in Fig. 6.1. The rectangular boxes represent separate components and the arrows the messages that are passed between them. For example, the user might trigger through the interface an editing command. As a result, the interface can trigger the corresponding command to modify the data. Subsequently, the change is reflected on the screen that has to be updated with the new content.

An important feature from an architectural point of view is that most of the editing systems can be realized with a single main loop that does not require multiple concurrent processes. Another important point that can be derived from a known reference architecture are associated risks. Due to long interaction sessions, an editing system is at risk of data loss in case of a system failure. Knowing these characteristics helps to anticipate and resolve known design issues.

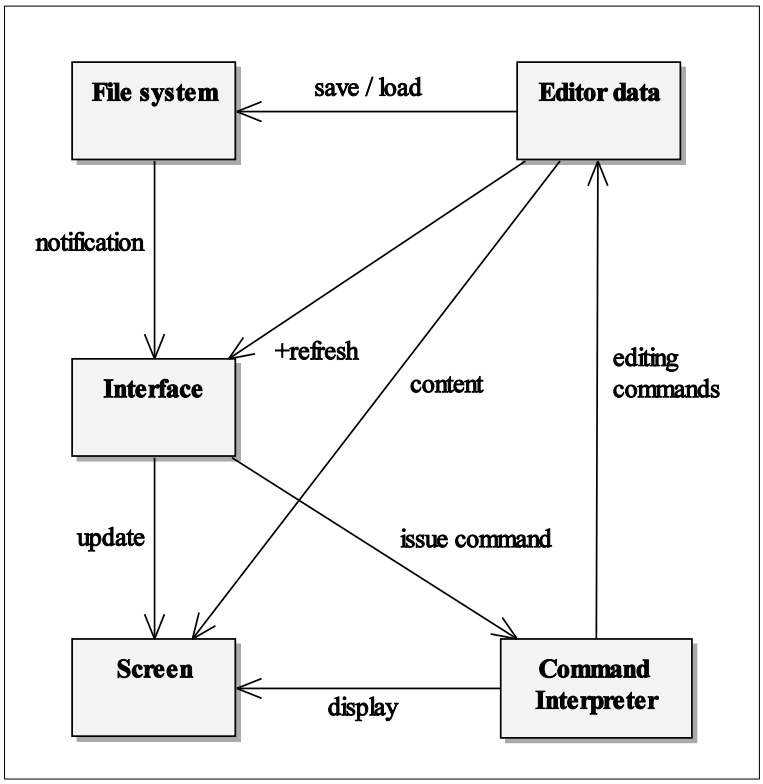


Figure 6.1: Simplified overview of an editing system

Language processing systems The second reference architecture is the class of language-processing systems. Language processing systems interpret input that has been specified in a formal language. The best known examples of this class are compilers. Language processing systems are also employed for Internet applications such as browsers that render the content of an XML file. Common characteristics are streamed input and output as well as specialized modules for data parsing and syntactical analysis. A typical language processing system is shown in Fig. 6.2. As shown in the overview, a core component is the interpreter that executes the commands. In Chapter 3.2 it has been shown that availability of a command interpreter makes the difference between configuration and programming in interface design. The major challenge for designing a language processing system is to define the language that is interpreted by the system.

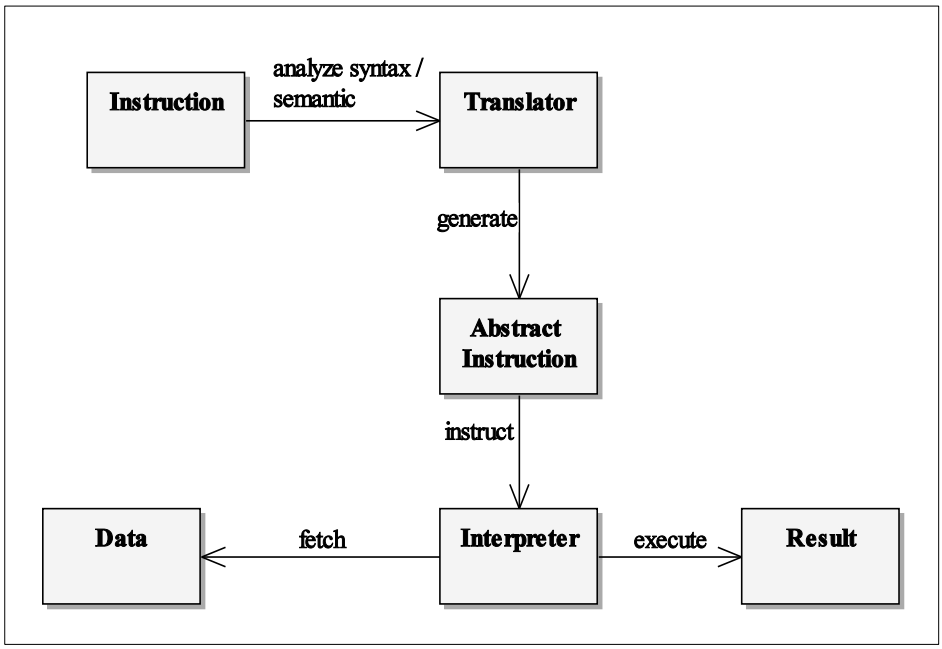


Figure 6.2: Simplified general setup of a language processing system

6.1.2 Component structure

Based on the two selected reference architectures, event processing and language processing systems, the major components for the Development-Environment can be modeled. A language processing system captures the aspects of creating a behavior description that is interpreted and executed during run-time. Throughout this definition task, the designer relies on common editing techniques, including cut-copy-paste commands, which are best modeled by an editing system reference architecture.

An overview of this model for the SRD architecture is visualized in Fig. 6.3. The model is an extension to the overall domain model that has been shown in Fig. 5.3 on page 96.

The central concept remains the concept of an application. The developer uses the tools provided by the Development-Environment to create an application for the user. To model this relationship the concept of an editor has been added to the global domain overview, derived from the general structure of an editing system. The editor produces application artifacts. The framework contains multiple editors, which provide different views and editing concepts for different application artifacts. For example,

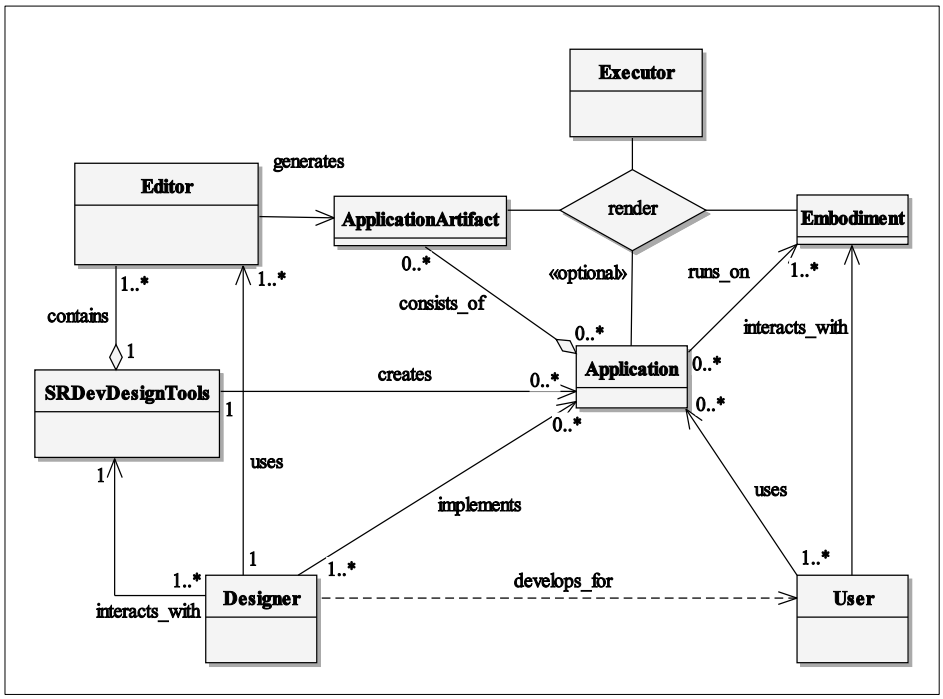


Figure 6.3: Domain model for the Social Robotics Development framework

the framework includes a special editor for creating keyframe animations and an editor for specifying the interactivity.

The current implementation indicates that a designer interacts with only one instance of the SRD framework. Collaboration can be achieved by working with multiple designers on multiple application artifacts with different editors.

The rendering process of the application has been modeled by a four-way *render* relationship between *application*, *application artifact*, *executor* and *embodiment*. The application determines when to render which artifact and the executor encapsulates the knowledge how to render the artifact on the embodiment. Application artifacts are only loosely coupled to an animation and can be reused across applications, but are defined for a particular embodiment definition. For example, a 'purr' animation might rely on the specifics of the iCat robot embodiment, but it may be reused in different applications. In the same manner, the embodiment can be used for different applications. For example, the iCat embodiment can take the role of a waiter in a restaurant, the role of a receptionist for a museum

exhibition, or the role of a tutor. For the user, the implementation in terms of application artifacts is transparent, because he only directly interacts with the interface. The user does not know how many artifacts constitute an application.
In this architecture, basic concepts of editing systems and language processing systems have been combined. On the one hand, the concept of an editor session and data artifacts are derived from a general editing system architecture. On the other hand, the basic concept of an interpreter is derived from language processing systems. In the presented architecture, the interpreter is represented by the executor component. The language that the interpreter understands is defined in terms of applications artifacts that are produced by the different editors.

6.1.3 Development Engine

In the above discussion, a general reference architecture has been established and several distinct components with different areas of responsibilities have been identified. These serve as input to define a central component model for the architecture. For this design decision, both structural and dynamic aspects have to be taken in consideration. In the following, first the structure of the decomposition is explained.
From the domain overview (see Fig. 5.2 on page 85) it can be seen that the main task for an application designer is to create an application. During the analysis of a reference architecture, this task has been addressed by the general concept of an editor that provides specific editing tools to the designer. However, the design framework has further responsibilities such as providing different views on the design problem, e.g. in form of a virtual representation of the posture of the robot, or a classification of the currently expressed emotion. Furthermore, several supporting functions are required which are not directly available to the designer, but are required from a systems perspective for example regarding communication, library access or for interfacing with the hardware.
Based on this observation, the concept of an editor can be generalized to the concept of a component. The resulting architecture is depicted in Fig. 6.4. This overview stresses two main design decisions. First, a highly modularized approach has been taken by defining the central concept of a component and second, the components are handled by a centralized component engine, in the diagram named 'Social Robot Development Engine' (`SRDevEngine`). A component has a defined component interface that includes a representation of the components functionality and definition, in

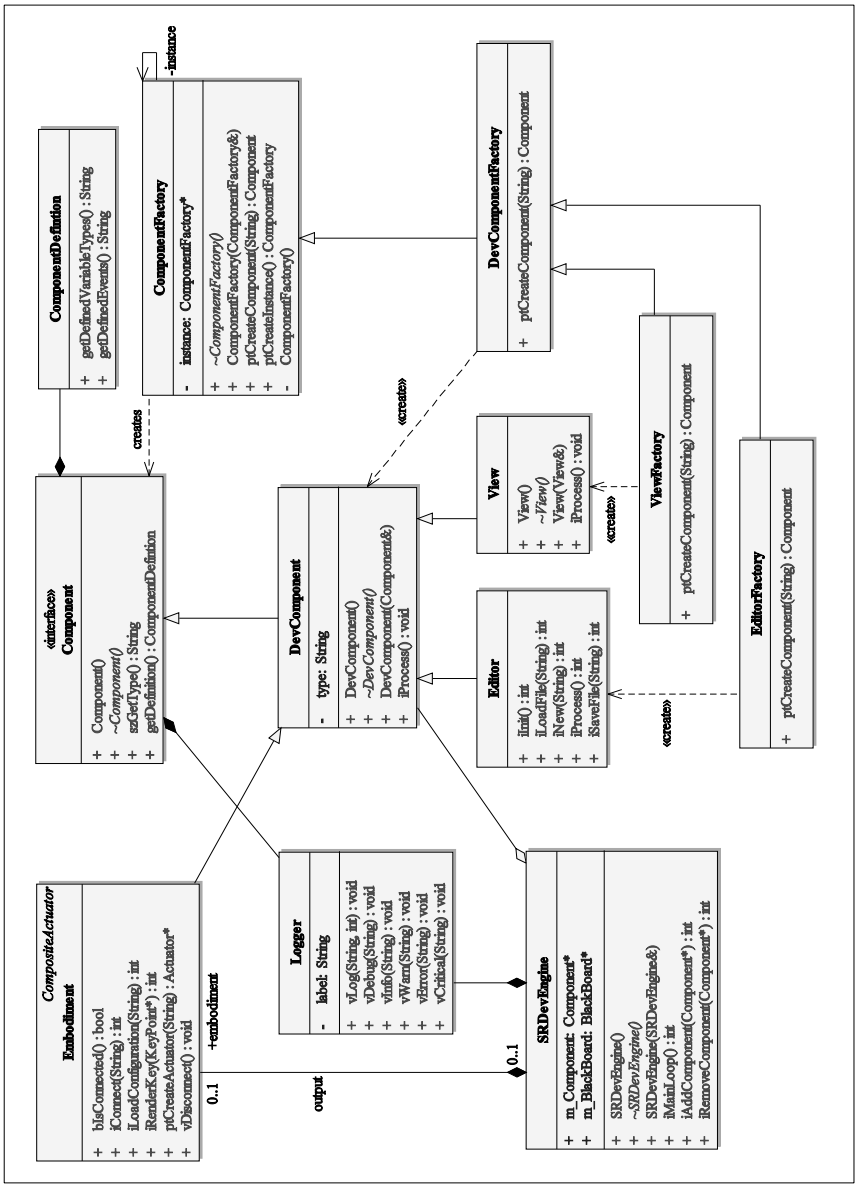


Figure 6.4: Structural decomposition for the Development-Environment.

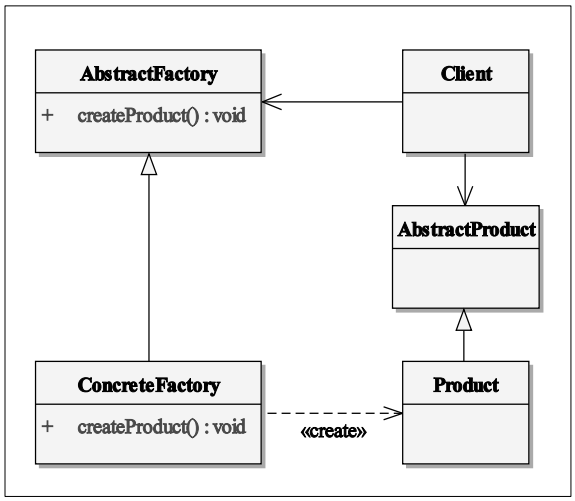


Figure 6.5: Factory design pattern

the diagram depicted by the `ComponentDefinition` class. This class provides meta information including which variables are defined by the module or which events are triggered. A component is the central concept in the architecture that encapsulates system functionality.
A certain type of component can have multiple instances during run-time. For example, a designer might want to edit multiple animation files simultaneously. That is, two instances of the component type `AnimationEditor` are needed. Therefore, the 'abstract factory pattern' has been used to decouple instantiation from the concrete class representations [78]. The abstract factory pattern is depicted in Fig. 6.5. The pattern defines five central concepts, an abstract and concrete factory, an abstract and concrete product and a client. The client is decoupled from the concrete representation of the product. It is only aware of its abstract interface. In the SRD architecture, the client is the central development engine that handles components of different types. However, the amount and types of the components is not known in advance. In order to decouple concrete implementation from how the objects are created the development engine uses an abstract factory (`DevComponentFactory`), which is implemented as a singleton. The singleton pattern [78] models the concept of global access variables for object oriented design. It is used whenever an object has only a single instance in a program. The general pattern is depicted in Fig. 6.6. The creation of the object is hidden in a private constructor and therefore not accessible to any client. This setup provides flexibility to extend the

Figure 6.6: Singleton design pattern

framework later on with further components, without having to change the core engine.
Having defined the structural model of the SRD framework architecture, the next critical decision concerns how these components interact with each other. Therefore, the following section discusses the communication model for the Development-Environment.

6.2 Communication architecture

Requirement 1.3 and requirement 1.4 demand that the framework provides multiple views on the design problem and that these views must be synchronized. In order to synchronize between different components, there needs to be the possibility to transmit data from one component to another. In this section a communication model is developed that allows editors to communicate with each other.

6.2.1 Communication models

In computer science, multiple models have been developed to realize the exchange of data between software components, including message passing, remote procedure calls, shared data, channels and sockets. These concepts have primarily been researched in the context of operating systems and inter-process communication theory. Gray [93] and Tanenbaum [229] provide an extensive theoretical overview and practical implementations for Unix environments. In theory, all of the above structures are equally powerful, because every of these models can be implemented with the concepts of the others. However, their performance and response time can be classified by the data access model in sequential and random data access. The sequential model only allows to read a sequential stream of data, in analogy to magnetic tapes that were used for system backups. Random access models, on the other hand, allow to read data in any order but require addressing overhead.

The architecture decision is ultimately also impacted by the application designer, in particular concerning the ease of use. Skillicorn et al. [224] surveys parallel computation models and defines six criteria to rate the applicability of the models which are: 1) ease of programming, 2) support for software development methodology, 3) architecture independence, 4) ease to understand, 5) guaranteed performance, and 6) availability of cost measures. These criteria demonstrate the importance for ease of use.

Blackboard system For the editor framework a shared data approach in the form of a blackboard system was chosen [80]. Blackboard systems allow very flexible decoupled n-to-n communication which is especially useful for prototyping systems. In an iterative design process, the focus usually is on flexibility, i.e. minimizing the effort that is necessary to implement a certain feature. Due to the flat communication structure and decoupling of sender and receiver semantically as well as timely, a blackboard architecture is very flexible to changes in participating components. A common use case for blackboard systems is to realize global variables. As a consequence, this kind of architecture also provides less security, because in principle every process is allowed to read and write any data that is on the blackboard. In literature, several approaches have been proposed to add security to blackboard systems by adding access restrictions to operations such as read and write but also introducing separate sections on the blackboard that may only be accessed with specific rights. The disadvantage is the administrative overhead that hinders the development process especially of rapid prototyping. For the development process it is more practical to replace advanced security features with a few behavioral policies. For example, one policy could regulate the ownership of the data object by defining that always the process that created the data object also has to release it again, unless the ownership is explicitly transferred to a different process. Nevertheless, security becomes crucial for deployment. This issue is solved in Chapter 7 when discussing the Execution-Environment.

Synchronization An additional problem concerns the synchronization of access to the data [229]. In theory, the attempt of two asynchronous processes to read and write a data element from the blackboard with non atomic operations at the same time could result in an undefined system state. While there is no conflict if two process read a data element at the same time, write access has to be synchronized. In the asynchronous model all requests to read data are blocked until the write operation has

been finished. In turn, all attempts to write data are blocked until a read operation is finished. If two processes issue a read and write request at exactly the same time, a deadlock situation could be created, assuming that read and write are not atomic functions.
In literature, synchronous and asynchronous execution models have been mainly studied in the context of reactive systems. The term 'reactive systems' was chosen to distinguish reactive system from 'interactive systems'. Halbwachs defines reactive systems by:

> Reactive systems are computer systems that continuously react to their environment at speed determined by this environment. (Halbwachs [99] p. 1)

He contrasts this definition with interactive systems in terms of scheduling. Interactive system continuously interact with the environment, but based on their own schedule. One example are operating systems. Furthermore, interactive systems usually employ an asynchronous communication model. Following Halbwachs, the main difference between synchronous reactive systems and interactive systems is that the first ones are deterministic while the latter ones are non-deterministic. For example, in the case of an operating system, the result of a function depends implicitly on the scheduler that at any time can interrupt the function.
For the theoretical foundation of synchronous reactive systems, two important assumptions are made: 1) stimuli and effect are simultaneous 2) broadcast of information is instantaneous. Even though these assumptions can never be met in real systems, they allow to mathematically model synchronous reactive systems and to assign clear semantics, which eases the process of programming significantly and solves common problems of programming in asynchronous architectures [5]. Furthermore, the advantage of a synchronous approach is that full process algebra can be developed [19, 5, 20] and that programs can be formally verified [99].
For these reasons, an overall synchronous communication model model was chosen for the SRD framework [99]. However, existing robot middleware tends to promote the asynchronous approach. This is one of the key points of making a distinction between Execution-Environment and Development-Environment, because it allows different communication models for development and execution. During run-time, multiple requirements of robot hard and software forbid to use the synchronous component approach for the execution (see Chapter 7). In the following, the modeling of synchronous components for the SRD architecture is explained.

6.2.2 Component collaboration

The overall component architecture is depicted in Fig. 6.7. At its core, the general type of a component has been introduced. A component defines a processing loop that is triggered by a central engine of the Development-Environment, in Fig. 6.7 labeled as Social Robot Development Engine (SRDevEngine). The central engine defines the clock for the processing cycles of the system. Furthermore, the integration with the blackboard system for communication is illustrated. Every component can access the blackboard and post data. Therefore, modules can read messages from the blackboard system during their processing time.

In order to avoid that the whole blackboard has to be searched for relevant data, the common data structure of an event queue was defined. Components that want to inform other components that a specific event has occurred post the event in the event queue. Events have a type identifier for classification and may be associated with any type of additional data. After adding to the queue, events will become available to other modules in the next main loop cycle. They remain in an internal buffer until posted to the official data, which is done by the main loop. Components that need to react to certain events, e.g. user input, only need to scan through the event queue. All events will be automatically removed by the main loop. An overview of the call sequence is illustrated in Fig. 6.8. In the main loop, the engine loops over a list of all components and triggers their execution. The components, in turn, may call further handlers to delegate the events for processing. This system works in analogy to GUI libraries that also define a central loop to process an event queue. In this sense, messages passing between components of the Development-Environment mirrors the 'chain of responsibility' [78] design pattern. This design pattern models the decoupling between sender and receiver of a request. In the architecture of the Development-Environment one component can take the role of a sender and another the role of a receiver. The general pattern is illustrated in Fig. 6.9. A client issues a request to a general handler. In turn, this handler delegates the execution of this request to derived modules through the successor relationship. The successor relationship defines a chain of execution from general modules to specialized modules.

Even though the modules of the SRD architecture are called in sequence, they are called in the same abstract instance of time, meaning that they are executed in parallel. Every component perceives the same state during an execution cycle. For general access to the blackboard this means that components that have a higher index in the execution list also have a higher

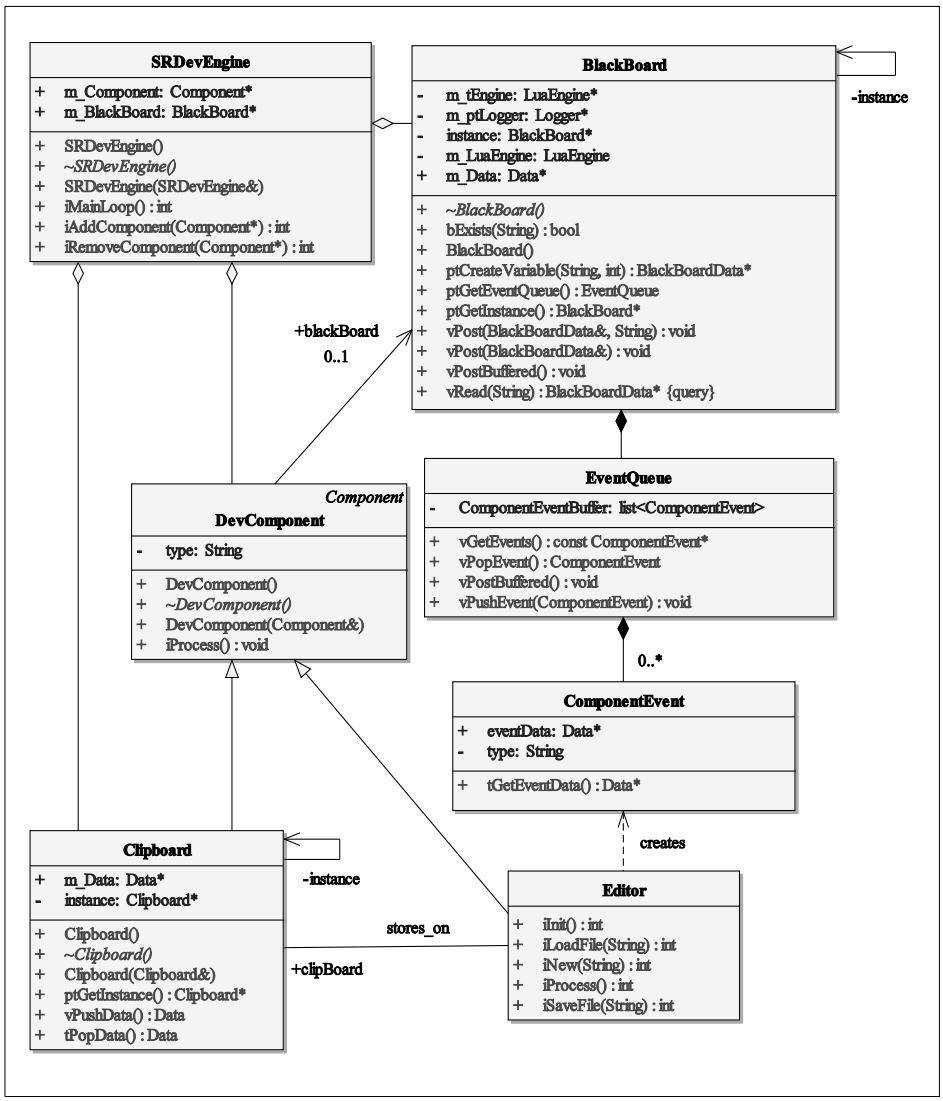


Figure 6.7: Modeling of synchronized component architecture

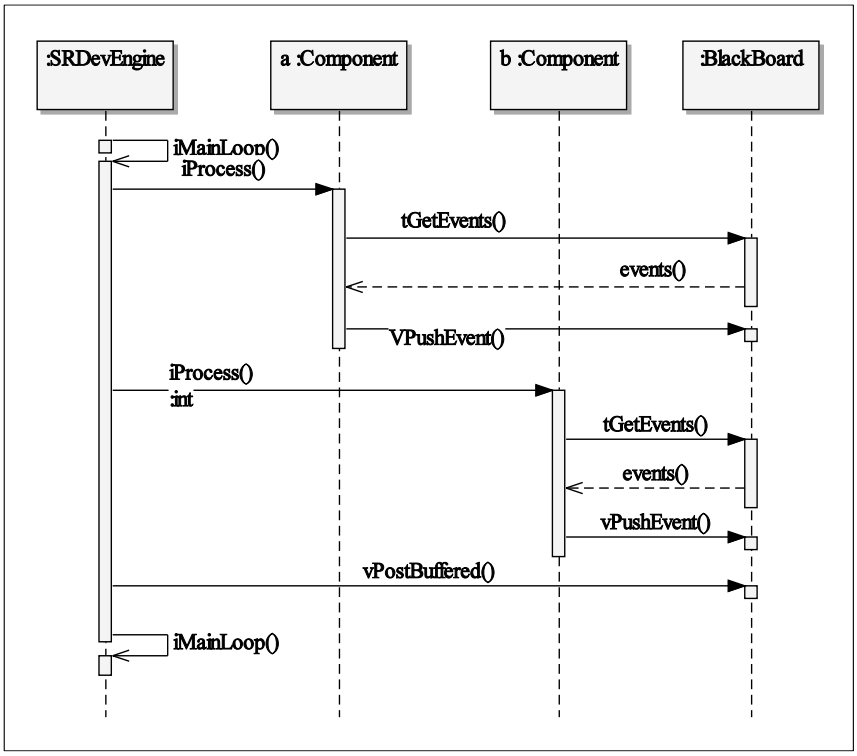


Figure 6.8: Main loop call sequence for passing events

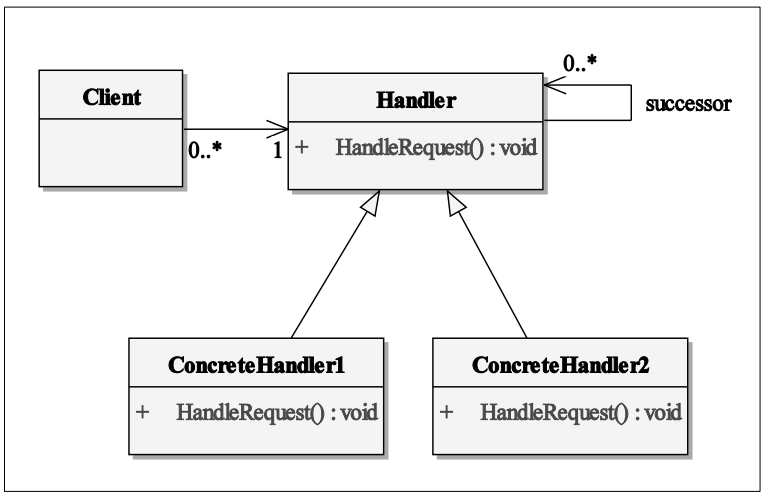


Figure 6.9: Chain of responsibility design pattern

priority with regards to data manipulation. If two components write the same data on the blackboard, only the one with higher priority remains. Priority is assigned by time of instantiation, resulting in a unique priority number for every component. The main advantage of this approach is that the overall system behavior is deterministic and dependencies are explicitly specified.
Dependencies are encoded in a component definition by defining which variables are written to the blackboard. For example, a basic module for generating a behavior creates motion trajectories for the actuators of the robot. These values might be overwritten by a specialized emotion generation module, which has more domain knowledge and operates on a higher level of abstraction.
In summary, the important modeling decision is that time is quantized to concrete execution cycles in which all components run in parallel. All components are assigned a unique priority that depends on time of instantiation. Components communicate using a shared blackboard. The crucial component to enable this setup is the blackboard system. It needs to provide global access to data and maintain updates for every execution cycle. The implementation of the blackboard system is covered in the following section.

6.3 Blackboard system implementation

The blackboard system serves as central communication point to exchange data between components of the Development-Environment. Therefore, the blackboard system needs to be accessible by all components in the Development-Environment, and exists only once. The blackboard system was designed according to the singleton pattern [78] (see page 107).
In computer science, several models for storing and maintaining data have been proposed, ranging from simple key-value pairs to complex relational and active databases [220]. A multitude of ready to use database systems exist, commercial and royalty free versions alike, including MySQL[1], Oracle [2] and Microsoft SQL Server [3]. Systems are available for desktop use as well as for server side data management for large enterprise structures. Furthermore, databases are a widely used technology for managing web content [252]. An interesting point from this application domain is that

[1] http://www.mysql.com
[2] http://www.oracle.com
[3] http://www.microsoft.com/sql

they are accessed using scripting technology, which offers great flexibility for the development of web content, e.g. e-commerce systems, social networks or private homepages.

6.3.1 Scripting engine

For the blackboard system the Lua scripting (see page 4.3) engine was chosen instead of a dedicated database. Lua offers a powerful data description syntax based on associative arrays. It can be integrated in a core application and data can be accessed and modified through a general programming interface. A particular advantage is that data can also directly be manipulated using the Lua scripting language itself. For example, a data entry might contain a string that contains Lua code. On access, this code can be executed to return a desired result. Furthermore, elements can be linked using Lua's table reference mechanism and several hooks can be defined for a meta level control over queries. However, the implementation is not bound to Lua. In principle any scripting language with the concept of variables and scoping could be chosen, e.g. Perl, Tcl or Python. Scripting offers powerful possibilities for rapid prototyping and quick results.

The interface to the blackboard system is modeled in Fig. 6.10. For example, it includes methods to post and read data. As explained in the previous section, all writes to the blackboard are buffered until the `vPostBuffered` method is called. All requests to read data, on the other hand, are executed directly. In its core, all requests are delegated to a Lua scripting engine that handles the underlying data. For example, Lua handles allocation and freeing of memory and allows high level methods for fully scoped variable access. An example snippet to illustrate the code syntax is given in code example 6.10.1. Comments start with `--` and finish at the end of the line. For a full description of the Lua syntax see [110]. One of the most powerful elements is that functions are first class values and can be assigned to variables. This enables to implement variables of which the value depends on the value of other variables on the blackboard.

In the above described approach, the allowable data types are determined by the Lua language. By default, Lua supports integers, floating point numbers, strings, Boolean and a couple of Lua specific types such as tables and threads. However, the data must not only be accessible to Lua, but ultimately to all components that access the blackboard. Therefore, an overall data type model has been defined for the SRD framework, which is explained in the following section.

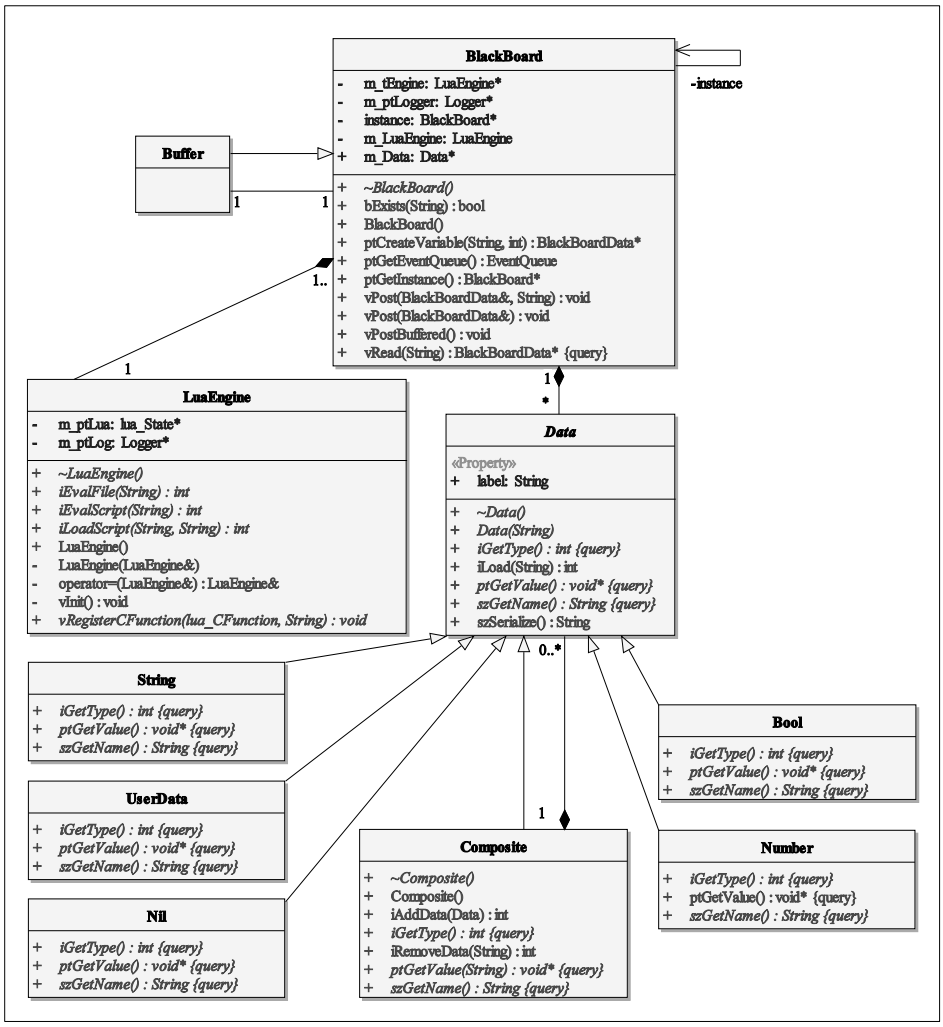


Figure 6.10: Overview of the black board communication architecture

Program 6.10.1 Example snippet illustrating Lua syntax for data access.

```
emotion = "happy"      -- Assignment of a string to a variable
                       -- in global scope

obj.pos.x = 100        -- Assignment of structured variables
obj.pos.width = 50

actuator[30] = 20      -- Indexing with integers (e.g. over time)

leds["green"] = 255    -- Indexing with strings

center = function()    -- Assignment of a function that computes
                       -- a value based on other entries
   return obj.pos.x + obj.pos.width / 2
end
```

6.3.2 Data types

The black board system allows components in the Development-Environment to post and read data. The data interface is therefore common to all components within the Development-Environment and needs to be specified. The general data abstraction hierarchy is shown in Fig. 6.10. A general data class defines the interface to access data. Within the architecture, any type of data is always associated with a label and type identifier. The label uniquely identifies the data in a given scope. The fully qualified label prefixes the label with a scope identifier and must be unique in the system. The identification number identifies the type and can be used for example for data conversion. This abstraction of a label and type identifier, in combination with the data object, allows to access the data independent from the programming language. For example, this allows to communicate between dynamically typed programming languages such as Tcl or Lua and typed programming languages such as Java or C++. Furthermore, it decouples the component interface from the underlying implementation of the blackboard system and therefore increases flexibility.

As illustrated in Fig. 6.10 six different basic data types have been defined: 1) Number 2) Boolean 3) String 4) UserData, 5) Composite and 6) Nil. The distinction between different types serves two purposes. First, it defines the allowed operations and second, it increases the performance of the implementation. The Tcl scripting language has demonstrated that in principle

all data types can be represented with strings. However, also Tcl converts the data implicitly for internal computation for performance reasons.

Every of the six defined basic data types has clear semantics. For example, a number is used for numerical calculation, irrespective of the precision. That means that no distinction is made between integers and floating point numbers. Therefore, the designer only has to remember one set of mathematical operations. The precision of the mathematical computations can be globally configured. If a certain operation is not natively available on a given platform, it can either simulate it or simply define that a platform is not suited for this type of computation. Therefore, the designer will have a clear overview on what precision is required for his application and which hardware platforms are supporting it, but the semantic that a number is used for numerical computation is unaffected.

Also the `Boolean` data type is a separate type and not simply coded as 0 or 1 number values. The Boolean data type represents a truth value that in the current implementation can take the values *true* or *false*. Assigning semantics to data types increases flexibility for later extensions. For example, the concept of a truth value could be extended to a degree of certainty about a specific entity. In this case, the two current values are only a subset of the possible values. To apply this change in the architecture only the coding of the truth values and logical operators such as test for equality would have to be redefined, leaving the rest unaffected.

The `String` data type is used to represent text labels when data needs to be presented in a human readable form. For example, in configuration files parameter values are connected to a human readable label that informs about the meaning of these values. The string data type is also used for text in- and output, for example in form of debug messages or to define speech in- and output of the application. Furthermore, the string data can hold executable chunks of code, i.e. in the form of scripts that can be dynamically generated and interpreted by a scripting engine. Scripts, in contrast to binary code, provide a human readable representation of a program.

The `UserData` type was introduced to efficiently capture artifacts such as images or sound files. It therefore serves as a base class representing arbitrary chunks of binary data. The actual type is stored by the type identifier. Every component may define new types. In order to avoid conflicts, the interface of a component is defined by a component definition as illustrated in Fig. 6.4 on page 105.

The `Nil` data type represents all cases that are not regular data, i.e. if data is not defined. Therefore, it is a valid replacement for all other types. For example, if the calculation of a function fails, it can simply return nil, which is also special type of number.
More complex data types can be constructed using the `Composite` type. A composite type contains a type and a label, but its `getValue` function accepts an optional parameter that identifies the label inside the composite structure. If this label is omitted, or the specified label is not included in the structure, by default the nil data type is returned. New elements are added to the composite structure, using the `addData` function. The only restriction is that inside one composite structure, a data label must be unique. If the label is already present, the add-operation fails and the structure remains unchanged. In order to replace an existing element with a new one that has the same label, the `removeData` function has to be called first. Analogously, if the remove operation specifies a label that does not exist, the operation leaves the data structure unchanged.
Having introduced the general overview and constructs of the Development-Environment, the focus of the following sections lies on a more detailed level on the structure of specific components.

6.4 Editors

The concept of an editor is one of the central concepts within the Social Robot Design framework. Requirement 1.2 demands that the Development-Environment offers tools that satisfy the needs of different approaches to the design task. The concept of a tool is captured by the general concept of an editor. The overview of the editor concept is given in Fig. 6.11. An editor is a component that allows to create and edit a specific application artifact. Every editor provides its own set of editing functions and user interface. For example, for writing scripting code, one valid option is to use a text editor. The user interface of the editor provides the editing functionality as well as visual feedback. This allows to improve the appearance by syntax highlighting that eases the coding task for the programmer. As a specialized editing functionality, it could offer auto-completion of known words, such as function names. However, a completely different editor approach is also conceivable to generate the script. The script could be generated with graphical programming as used for the Lego®Mindstorms. Instead of text editing facilities, a graphical editor provides graphical building blocks and connectors that can be arranged on a 2D canvas.

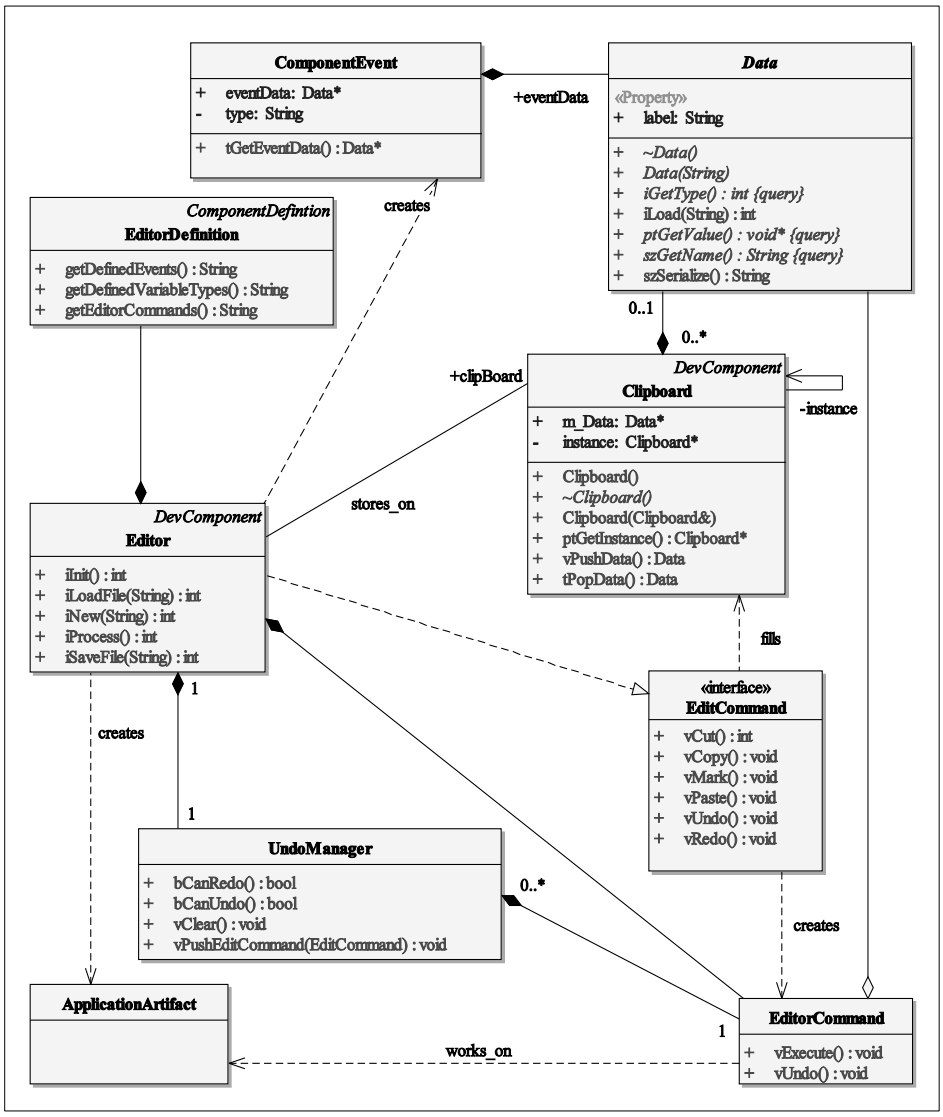


Figure 6.11: Editor architecture overview

Conversely, with the same editor different application artifacts could be realized. The text editor concept could also be used to define a list of numbers representing keypoint positions, and therefore to implement an animation. These examples show that neither the type of editor determines the application artifact nor does the application artifact determine the editor.

Despite the variety of editing methods, there are a few editing concepts that are general for all editors. At its core, an editor is responsible for creating application artifacts. Therefore, all editors have in common to be able to create a new file, to load a file and to save the changes to disk. Furthermore, every editor provides a set of operators that work on an application artifact. Common operations for an editor are to mark, cut, copy and paste certain content. Therefore, all editors provide an interface to unify these operations and to allow cross-editor collaboration. In very special cases these common editor functions might be left empty, e.g. if an editor is only able to modify existing data, or if content cannot be copied for technical or legal reasons.

Cross editor collaboration can be realized in two ways. First, an editor may use the component message passing system and write data to the blackboard and second, it may use general cut-copy-paste functionality. The second option is realized with a central clipboard, that itself is a component in the framework. It allows to store general data, similarly to the blackboard. The general data type concept as described in section 6.3.2 is ubiquitous throughout the architecture. However, the clipboard defines an order in which the data was copied. A paste operation will return the most recent copied or cut data.

Furthermore, the general editor concept defines an undo-manager. Especially in long editing sessions, which are at the core of the editing concept, humans tend to make mistakes. A supportive framework needs to be able to compensate these mistakes and offer an undo functionality. The undo-manager is localized in an editor, so that every editor can track local changes. For implementing the concept of an undo-manager, the concept of an editing command has been introduced. An editing command encapsulates the specification of a command to the editor. Such a command can either directly be generated through a user interface, or by automating the task using a batch file script. In the case of the undo-manager, the commands are stored in order to enable a trace of the commands. Every command defines an execution function and an undo-function. For the design of the editor command, the general command design pattern [78] has been used. The command pattern decouples command invocation from

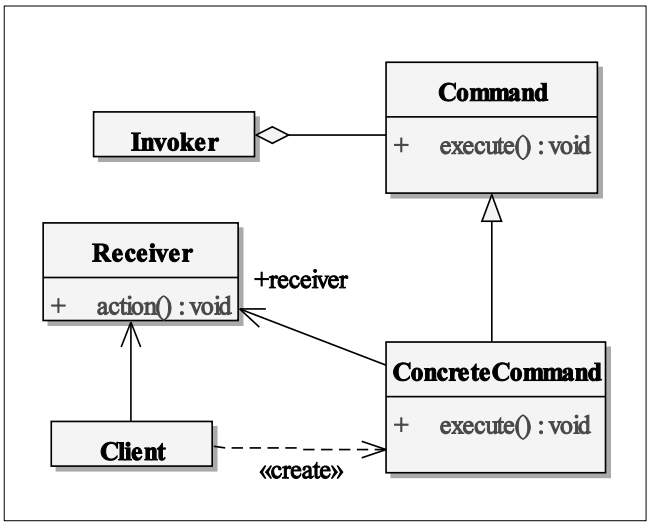


Figure 6.12: Command design pattern

command implementation. The command is transformed in an object that can be passed just like other objects as depicted in Fig 6.12. This command object defines an abstract interface that invokes a certain function. A receiver of the command is stored in the command object, which has the knowledge how to carry out the requested command. A typical use case scenarios for the command pattern are graphical user interfaces and cut-copy-paste commands. In the case of the SRD editor, the invoker of the undo-command is the undo-manager, but only the target editor knows how to execute the task. Therefore, the undo-manager is localized in the editor to keep a local history of edit commands. As described in the central SRD architecture a component may have multiple instances. Multiple instances of every editor can be active, for example if two or more animations are developed at the same time. For creating multiple editors of the same kind, but with different parameters, the factory pattern was applied as described in section 6.5 on page 106.

6.4.1 Functional animation editor

In Chapter 3.3 the concept of functional animations has been introduced. This section describes the implementation for functional animations and how it is integrated in the global animation framework.

The functional animation extends the concept of keyframe animation with the possibility to generate dynamic expressive behaviors. In traditional animation, a keyframe describes a posture of a character at a given moment in time. For example in comic animation, this posture is given by a drawing that was created by a lead animator [253]. The missing frames between these keyframes are filled by different animators. Once the animation is finished, it is static in a sense that it cannot easily be updated. Changing the posture of a keyframe entails to change all neighboring interpolation frames. However, movie characters do not need to adapt their behavior during the play of the movie, because for traditional movies the script is fixed and there is no interactivity with the audience or the environment of the theatre.

In order to create dynamic animations for social robots, the concrete behavior must be computed during run-time of an application. Therefore, the movement needs to be numerically parameterized to be computable. For this, the constraints of the robotic embodiment are an advantage. While a comic character has virtually an unlimited amount of degrees of freedom, robots are constrained the amount of available actuators and physical laws. Therefore, the posture of a robot can be completely described by a relatively small set of parameters, in comparison to the number of parameters that are necessary to fully describe a comic character.

Functional animation architecture

For the architecture, the basic concepts of keypoint, interpolation and actuator have been modeled as distinct objects. An overview of the structural decomposition is given in Fig. 6.13. A functional animation comprises a set of actuators that perform a trajectory over time. The actuators that are defined for the animation have to match the available actuators for a concrete embodiment. Derived from the general framework, the abstract factory design pattern [78] (described in Chapter 6.5 on page 106) is used to parameterize the creation of a functional animation. The factory uses the interface of an embodiment definition to create a functional animation according to the degrees of freedom, i.e. the set of actuators.

The time dependent trajectory of an embodiment is partially defined by keypoints for this actuator. If no keypoints are defined, then the position of this actuator is undefined by the animation. A keypoint class has three basic elements, namely a start time, a state and an interpolation. The start time defines the lower bound of the definition interval of the keypoint, the

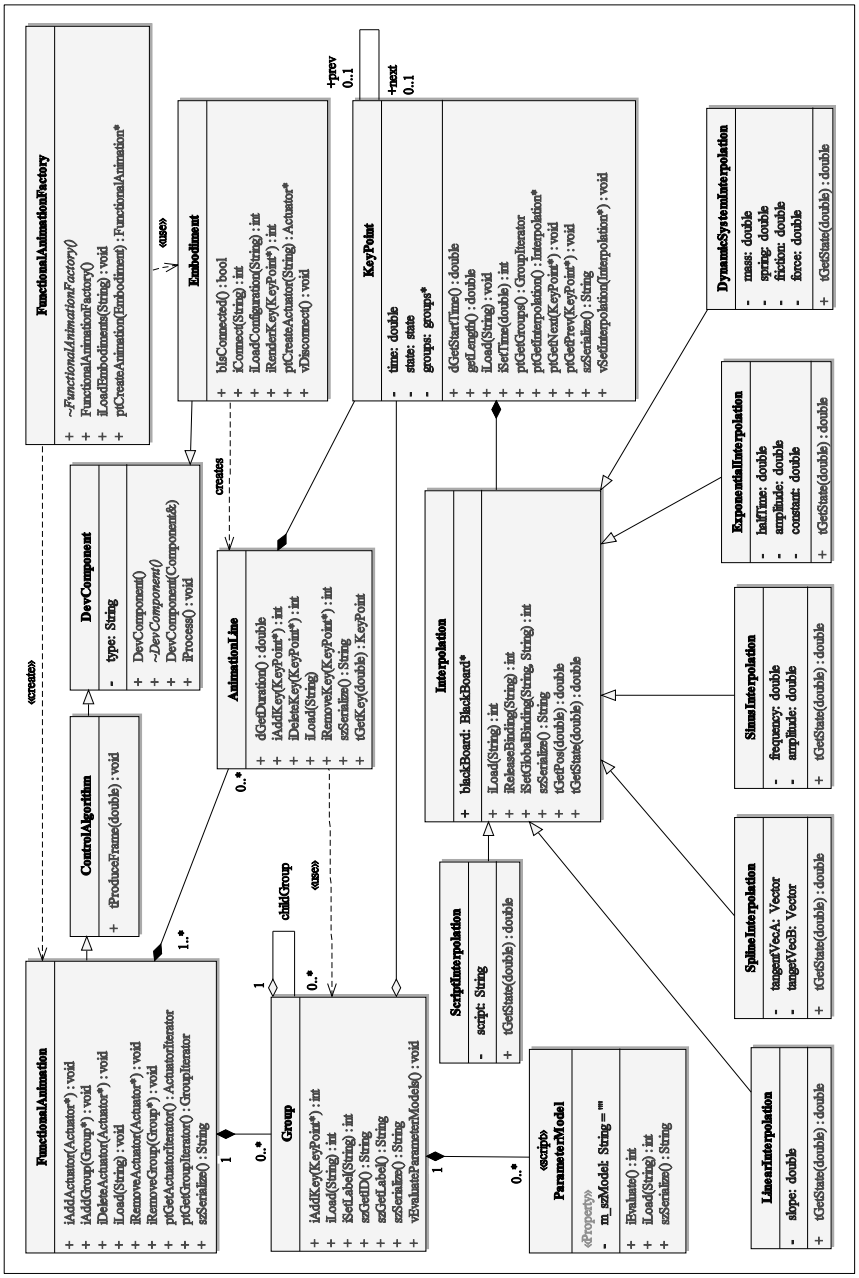


Figure 6.13: Architecture of functional animations

state defines the position of the actuator at the start time and the interpolation defines the function that the actuator performs in the definition interval. The upper limit of the definition interval is not explicitly stored but calculated on request. To this end, keypoints are organized in a linked list, sorted by the start time parameter. If the keypoint is the last point, i.e. if the next element is undefined, the definition interval is a single moment in time as for the definition of keypoints in traditional animations.

Interpolations The interpolation class generalizes different types of interpolation functions according to the command pattern as described on page 121. Linear and spline interpolation are two types of interpolation that are typically used in movie and game industry. Especially splines are commonly used because of the possibility to fade movements in and out for smooth and natural looking behavior. Several other classes have been added to the list of interpolations for quick realization of animations. A sinus interpolation is in particular suited to animate nodding or shaking of a head, for example for the iCat embodiment. The exponential interpolation supported natural looking blinks for the eyelid actuators and the dynamic-system interpolation allowed to simulate a physical mass-spring-damper model in order to give the impression of weight of the head. Lastly, for quick prototyping, also a general script interpolation has been implemented. The interpolation function depends on a script in the Lua scripting language. This script can furthermore be parameterized with parameters that are stored on the global blackboard. A functional animation is also a component in the general system and therefore has full access to the blackboard communication system.

Parameterization All interpolation methods are parameterized by a set of control parameters. By default, every interpolation stores the set of parameters locally. However, they can also be bound to a variable on the blackboard. Whenever a variable is bound to the blackboard, the value of the interpolation function is determined by the current parameter value from the blackboard instead of the local value. When the binding is released, the local value is restored.

Groups Another important concept for a functional animation is the concept of groups. Keypoints may be grouped to form a set that is treated as an entity. Groups may contain keypoints or other groups. Hierarchical grouping therefore forms a tree like structure of dependencies. Edit events

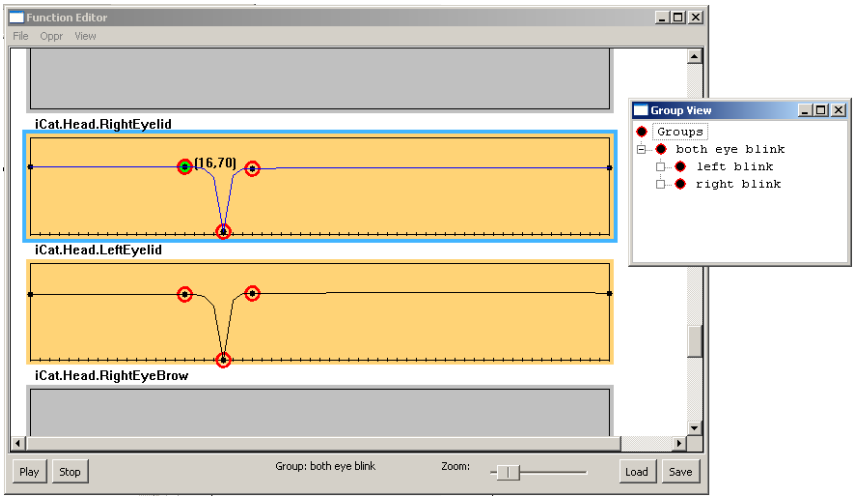


Figure 6.14: Grouping concept for functional animations. Groups from entities that can be attached a descriptive label.

such as cut-copy-paste are executed for the whole group, i.e. for all keypoints simultaneously. Keypoints within the group keep their relative time distance to each other. Furthermore, a label can be assigned to a group to annotate the group with a semantic meaning. For example, a blink with an eyelid can be parameterized with three keypoints as shown in Fig. 6.14. These keypoints can be grouped in order to avoid that the relative distance of the keypoint changes or that accidentally a keypoint is deleted. This way, the designer can create higher level abstractions for an animation library, e.g. by creating a 'left-eye blink' and 'right-eye blink' that can be reused to form a 'both-eyes blink'.

Variable scoping In the above example of the 'blink' animation, an exponential interpolation function has been used. The exponential function has the basic form:

$$f(t) = c + ae^{bx} \tag{6.1}$$

The three parameters (c,a,b) control the shape of the curve. They can either be saved locally, or bound to a global blackboard variable. The concept of groups adds variable scoping to this mechanism. For example, the fully qualified name of the amplitude parameter from the left eye blink is in Lua syntax given by: `blackboard['blink animation']['both eye blink']['left blink'].amplitude` The animation itself defines a group, with a label equal

to the name of the animation. Therefore, adding new keypoints to an animation just adds new keypoints to this group.

Parameter models The final concept that is introduced by functional animations is the concept of parameter models. Parameter models define a function that performs a coordinate transformation on the control parameters of the interpolation function. In the architecture these models have been implemented by scripts. Parameter models can be attached to every group and are executed synchronously with the main loop of the component architecture. The evaluation is executed according to the hierarchy of the groups, starting with the top level. For example, an emotion might be parameterized according to Russell's 'circumplex model of affect' along the two dimensions of valence and arousal [207]. These emotion parameters define a position in an affective space. The coordinates can be mapped from the emotion space to a motion space through parameter models. As explained in Chapter 9, perceived arousal is positively correlated to acceleration of an actuator in motion space. Therefore, a suitable mapping function might be approximated with a linear relationship. For high levels of arousal the parameter model will create parameters that produce high values of acceleration. Accordingly, low values of arousal will be mapped on low levels of acceleration. Parameter models form a hierarchy. High level parameters influence the evaluation of low level parameters.

6.4.2 Animation Editor iCat

For demonstrating the applicability of functional animations a prototype with the scripting language Lua was developed. The overall GUI of the editor is depicted in Fig. 6.15. On start, the editor offers to create a new animation for a given embodiment. The embodiment determines the actuators that are available for the animation. By default, all actuators are disabled, that is they define no values. Every actuator can separately be enabled through a context menu.

The designer may place keypoints along the timeline of the animation. The length of the animation is thereby dynamically calculated, depending on the keypoint positions. The definition area for the motion trajectory function f may freely be shifted throughout an animation by moving the first keypoint to a later time position t. The keypoint with the smallest start time value defines the overall begin of the animation.

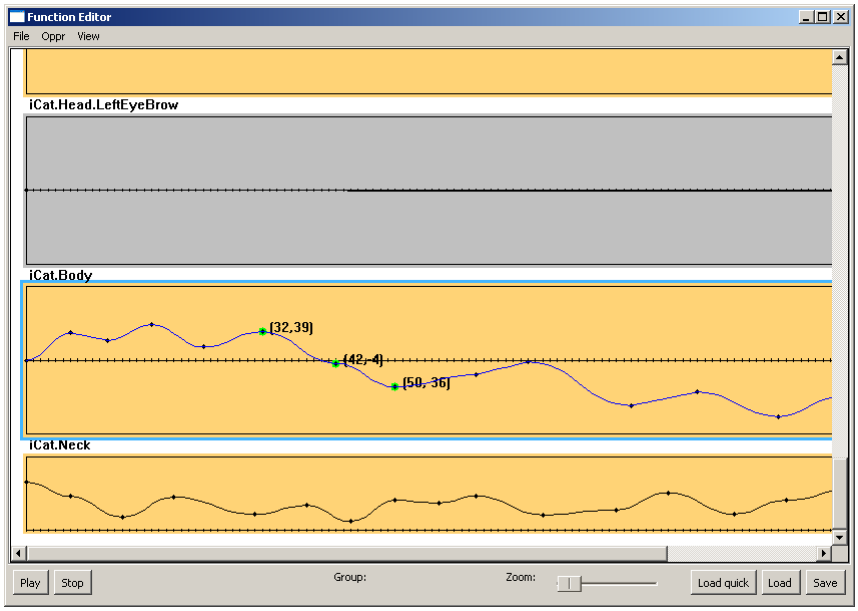


Figure 6.15: Prototype of functional animation editor

By default, the editor chooses a linear interpolation between two keypoints, but this default can be adjusted through a configuration file. Other parameters that may be configured include default embodiment and default animation length. However, all of the values may also be changed during run-time. In order to change the interpolation function of a keypoint, the designer may select a different interpolation type through the context menu of the keypoint. Every interpolation function is controlled by a different set of parameters. Therefore, every interpolation function is represented by an own GUI object that provides an interface to adjust the parameters. For example as shown in Fig. 6.16 the parameters can be adjusted through a special control window. Another possibility is to present handles directly on the interpolation line as demonstrated in Fig. 6.17. A separate window allows to define global bindings for the control parameters. In the current implementation not only the interpolation parameters can be bound to a global variable, but also the position of the keypoint itself. This gives full freedom to adjust the animation according to the parameters of parameter models. For example, if a 'reach action' is to be programmed with the editor, the keypoint position defines the end effector position of the robot. However, how the robot reaches this point is defined by the interpolation function. As a result, this approach allows to dynamically combine

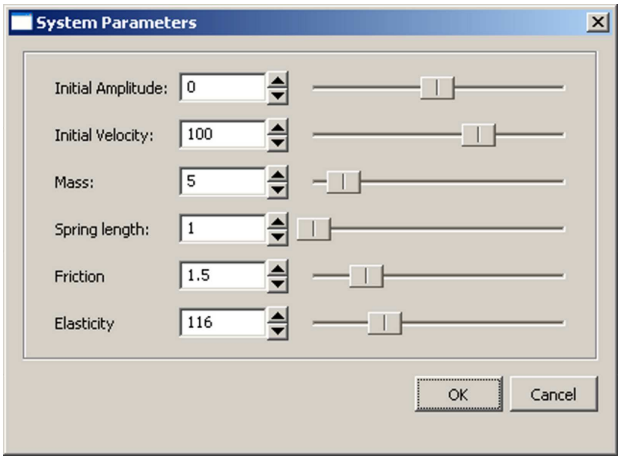


Figure 6.16: Configuration window for a second-order dynamic system interpolation

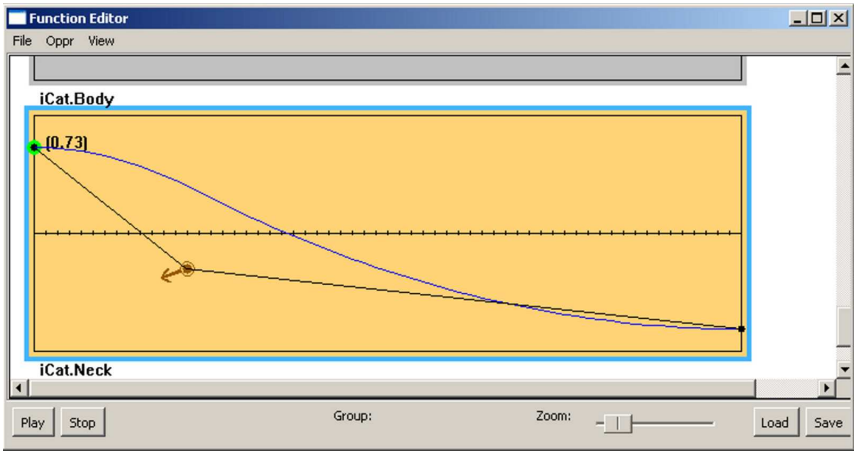


Figure 6.17: Spline parameters can be adjusted by dragging the gray handle

expressive and functional behaviors. Additionally, the dynamics of the system can be compared to the hardware constraints of the embodiment and impossible or dangerous hardware configurations can be visually marked.
The overall editing concept of the iCat editor follows the posture based keyframe animation of traditional character animation. The time dependent trajectories of the actuators directly represent the position of the actuators. In the following a different editing concept based on functional animations is presented for an editor for the Roomba platform.

6.4.3 Roomba Path Editor

The Roomba platform is a mobile vacuum cleaning robot developed by iRobot. An overview of the hardware is given in Chapter 2.3 and a detailed discussion on the drive system is given in Chapter 7.4.
The major difference between a velocity controlled mobile robot such as Roomba and a posture controlled embodiment such as iCat is that iCat is a holonomic system, while Roomba robot is a non-holonomic system. In particular, the two controllable degrees of freedom of Roomba are the wheel speeds of the differential drive system, but the total degrees of freedom are the x and y position plus an orientation degree of freedom. As a result, it is difficult for an animation designer to create a desired path only based on low level wheel speeds. An animation designer rather operates on a high level path abstraction than on a low level control of wheel speeds. For these reasons the Roomba animation editor takes the path of the robot as basic editing concept.
A prototype of the resulting editor is shown in Fig. 6.18. It applies the same functional animation principle as the iCat animation editor, but instead of a position of an actuator, a keypoint denotes a position and orientation on a two dimensional surface. The similarity is that both approaches describe the overall state of the robot.
For editing an animation for the Roomba editor, the designer places consecutive keypoints in a two dimensional editing area. The resulting path forms a two-dimensional trajectory. The robot faces always in the direction of the trajectory.
Like for the iCat editor, keypoints are organized in a linked list. Time progresses along the two dimensional path. However, the position progress of the robot is not only determined by the trajectory, but also by the velocity by which the robot moves along the designed path. Therefore, the designer may edit the path and velocity independent of each other. This independence is realized by assigning two interpolation functions to

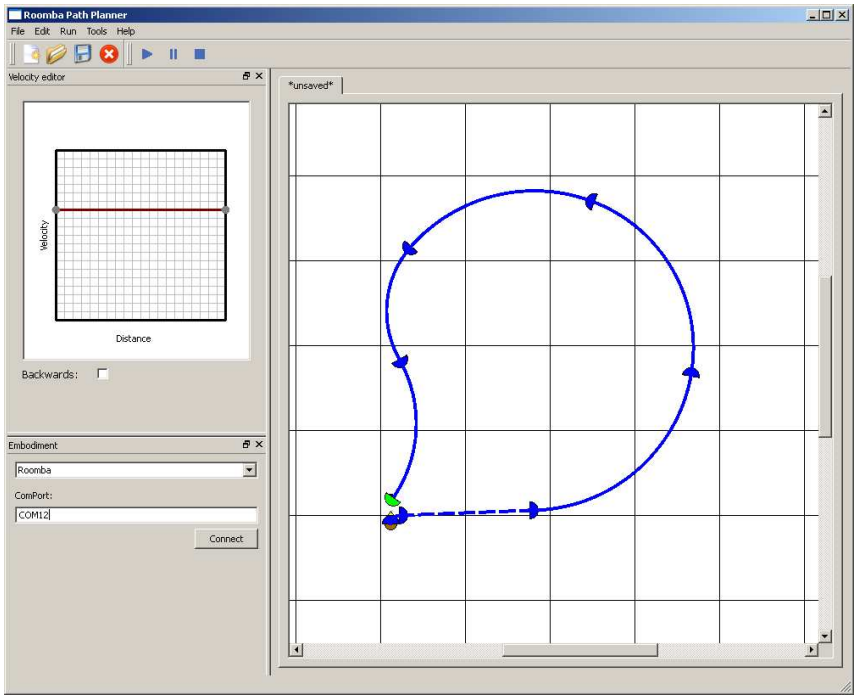


Figure 6.18: Roomba path editor

every keypoint. The first interpolation function determines the path and the second determines the velocity along the path. In Fig. 6.18 the velocity trajectory is adjusted in a separate windows, displayed in the lower left corner.

Based on this principle, multiple editing approaches are conceivable. As outlined above, the position and orientation of the robot are determined by keypoint position and interpolation function. For editing, the animation editor may take two approaches. First, the editor may regard the position of keypoints as fixed and constrain the interpolation function accordingly, or second, it may take the interpolation function as fixed and place the keypoints accordingly. This restrictions result from the physical constraints of the robot. The path that is taken between two keypoints also determines the orientation of the robot at the end of the path, and therefore also the initial orientation for the next interval. For this reason, either the interpolation function may freely be chosen, but constraining the positioning of keypoints along this path, or the position of keypoints may freely be

chosen, but constraining the path between these keypoints, to also end up at a defined orientation.

For the implementation of the path editor, the first approach was chosen. The main reason for this design decision is that this approach offers easier integration of functional and expressive behaviors. From a functional point of view, a cleaning robot needs to cover a certain area and to move along a path that avoids objects. For example, if a corner needs to be cleaned, the target position and orientation of the robot are the constraining parameters to assure proper floor coverage. The path that the robot the robot takes to the corner is ultimately constrained by the setup of the environment and the robot's physical movement constraints, rather than the path defining the end position.

The calculation of the path takes the drive constraints of the hardware platform into account. In particular, the interface to the controller takes two parameters to steer the robot, namely *radius* and *orientation*. A detailed description of the drive system can be found in Chapter 9. In summary, the drive system allows three different modes of operation. First, the robot travels along a perfect circle with given radius and velocity, which is the default drive behavior. Second, the robot drives a straight line, which is realized by a special case value for the radius and third, the robot turns on the spot. Theoretically, with these modes any path can be approximated using sufficiently small path segments. However, the maximum velocity of a wheel is limited by the servo hardware. Therefore, not every combination of radius and velocity are possible, because on a circular trajectory the outer wheel needs to rotate faster than the inner wheel. If already the inner wheel rotates with maximum velocity, the outer wheel cannot rotate faster to stay on the desired circular path.

The designer needs to be aware of these hardware constraints, otherwise the intended application requires a different hardware. For this reason, the path editor provides visual feedback to the designer. First of all, the editor indicates the position and orientation of the robot. The combination of both cues is important, because both need to form a differentiable function. The robot cannot jump from one orientation to another. Secondly, the editor indicates if a trajectory is possible or not, given a particular embodiment.

For storing the animation, the resulting functional animation parameters are coded in an XML file. Functional animation files are therefore also editable with a standard text editor. Few constraints on the saving format increase the flexibility of the system, so that the files may also be edited by new editor concepts for the SRD framework.

```
<state name="startUp">
  <action>
    <seq>
       <par>
          <say utterance="Hi! I'm iCat." />
          <play anim="\Anims\Hello.raf" />
        </par>
        <par>
           <say utterance="Have a seat please." />
           <play anim="\Anims\OfferSeat.raf" />
          </par>
        <par>
     ...
    </seq>
  </action>
  <transition> to="startLesson" />
 </state>
```

Figure 6.19: Example snippet of the 'Robot Interaction Markup Language' (RIBML), taken from the interaction script of the tutoring application. The three dots indicate the omission of some code for simplicity.

6.4.4 Interaction design editor

Most commonly, how to react on an event depends on the context of the current situation. Within the OPPR framework, a special state based interaction design language was developed named *Robot Interaction and Behavior Markup Language*(RIBML) by van Breemen and extended in the context of this thesis. An example snippet is shown in Fig. 6.19. The language includes constructs to describe an application in terms of sequential and parallel statements. The XML statements are parsed and executed in an internal virtual machine. Ultimately, the statements trigger events that are communicated within the communication framework in order to trigger actions of the robot.

One important extension that has been made introduces commands to query states from the environment, using a situational awareness architecture as described in Chapter 7.3. This allows the designer to control the behavior of the robot using high level commands such as "`<look_at_user/>`" or "`<wait_for_reply/>`".

6.4.5 Application logic

The application logic defines the robot's autonomous behavior as well as how to react to events from the environment. As described in the general use case scenario of the SRD framework, an application designer with a computer science background may choose a programming paradigm of his liking, including imperative, functional, object-oriented or rule based approaches. Within the SRD framework this is generalized by an executor for application artifacts. An application artifact might be an animation that controls the state of actuators over time, but also an executable script using scripting technology. In general, a concrete `ArtifactExecutor` is realized by sub-classing the general `ArtifactExecutor` class as depicted in Fig. 6.20. For example, the depicted `ScriptingModule` executes a Lua script.

6.5 Views

In an iterative design approach, the designer needs to be able to observe the current state of an application artifact. This can be illustrated with the scenario of creating an animation for a robot. For example, a keyframe editor or a posture editor can be used to create a basic animation. The keyframe editor might provide a visualization of the motion trajectory for every actuator over time. From the basic shapes of the trajectories, the animator already can get an idea on how the animation will look like, but it cannot replace a detailed visualization. The problem is that a numerical representation is not the appropriate view on the problem. For this reason, the preview facilities have been introduced in the framework as depicted in Fig. 6.20. The preview facilities are able to render certain application artifacts on the embodiment. In particular, it is important to visualize the dynamic aspects of the animation or behavior.

The preview facility was directly derived from the domain model as shown in Fig. 6.3. In the domain model, a four way 'render' relationship was established between application, application artifact, embodiment and an executor. This model decoupled the rendering of application artifacts from the underlying hardware. The role of the application in this relationship is to control the executor to render application artifacts on the embodiment. In this model, however, the overall application needs to be defined before application artifacts can be tested. For this reason, the application has been modeled, as a compound application artifact, which in itself has the

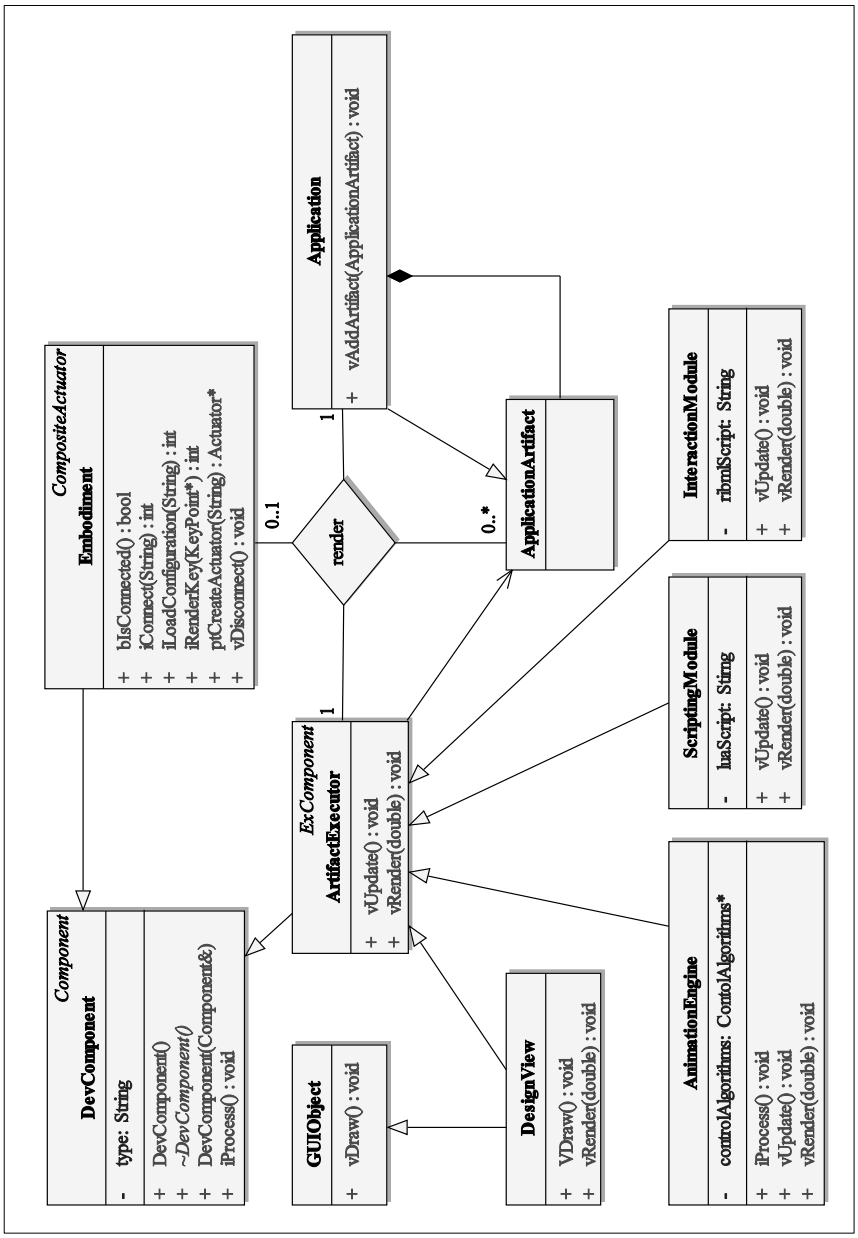


Figure 6.20: Preview facility for the Development-Environment

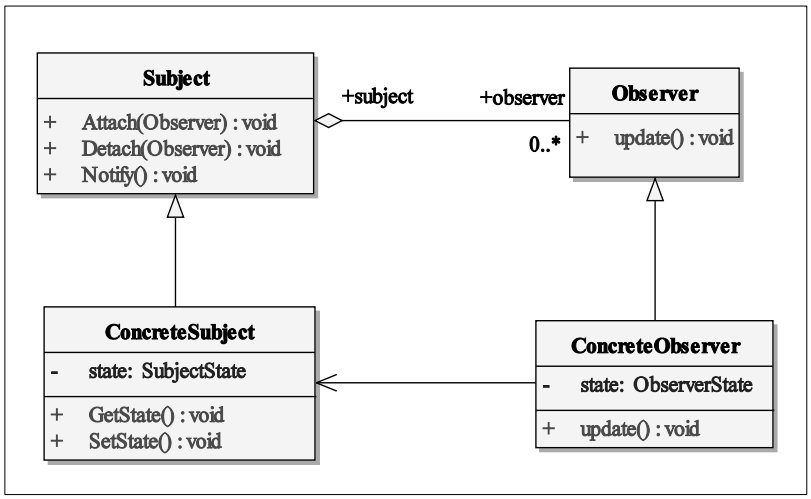


Figure 6.21: Observer pattern

same type as an application. This enables to wrap any application artifact in an application object for the purpose of testing.

The preview facility reuses the Animation-Engine as defined in [241] for the role of the executor. A detailed description of the Animation-Engine is given in Chapter 4.2 on page 71. It has been integrated in the SRD architecture following the observer design pattern [78]. The observer pattern models a one-to-many dependency between objects in which a change of the state of the one object is communicated to all other objects participating in the collaboration. A typical application area for the observer pattern is graphical user interfaces that provide multiple visualizations or controls to the user. An overview of the general pattern is illustrated in Fig. 6.21. In the context of a design framework for social robots, the common object is the application that is rendered on the robot. The design of an application can be approached from multiple points of view. For example, a designer may focus on the behavior of the robot in terms of movement patterns, in terms of light feedback, or in terms of interactivity. The above introduced editor concept does not only provide tools to create and edit animation artifacts, but they also provide different views on the application. The preview facility extends this concept by a general visualization method. The Animation-Engine is generalized by an *artifact executor* which renders application artifacts. However, because the executor is also a general component in the architecture, the artifact does not necessarily need to be rendered on the embodiment but also a graphical

user interface object may be updated. In this sense, the GUI object takes the role of an embodiment. The concepts of a view and embodiment have deliberately been kept separate, because an application is always bound to one robotic embodiment.
With the preview approach multiple views can be shown at the same time. Through the observer pattern, they reflect the modifications of the application artifact. This enables the designer for example to track the consequences of an altered movement trajectory on the conveyed emotional message that the user perceives when observing the embodiment.

6.6 Embodiment

In the above described architecture, the embodiment has been referenced as the primary rendering device for an application. In this section, the structural decomposition of the embodiment is explained.
For the SRD architecture an embodiment is a generalization of a concrete set of sensors and actuators. The general concept of embodiment is depicted in Fig. 6.22. In these terms, also a remote controller that only sends control commands to the hardware, but does the main computation outside of the robot embodiment, is considered as belonging to the embodiment. An embodiment is the central concept for abstracting the underlying robotic hardware on which an application is rendered, i.e. a robotic vacuum cleaner or an interface robot such as iCat. However, the concept is not limited to physical robots. It also includes virtual representations such as screen based characters.
As shown in Fig. 6.3, an application is always bound to an embodiment. This dependency has consequences on the development interface for the developer, because a virtual representation is formally a different embodiment than the physical embodiment. Therefore, an application that is created for the physical iCat embodiment cannot be tested with a virtual character. In order to enable that an application can be used with multiple embodiments, the architecture needs a possibility to compare the similarity of two embodiments.
In the SRD architecture exist two basic possibilities to introduce a virtual representation of a hardware embodiment. The first possibility is to define a GUI object to present an additional view on the design problem. The advantage of this approach is that the virtual representation is not bound to the embodiment and may present an abstract view. However, the disadvantage is that virtual representation and physical hardware must be

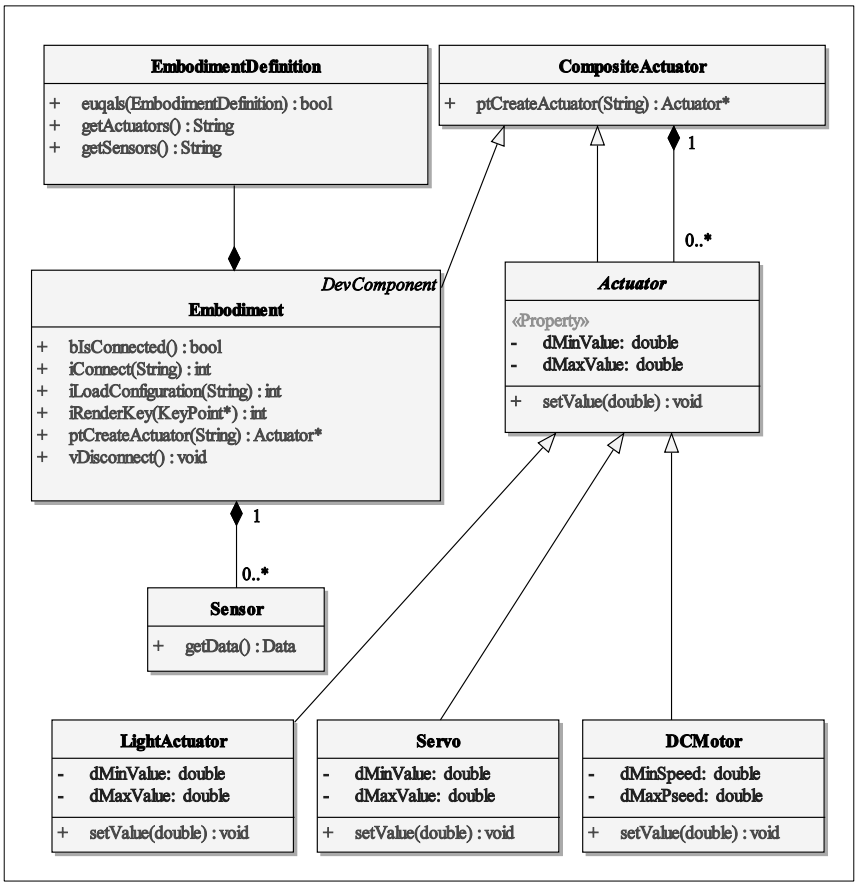


Figure 6.22: Embodiment implementation

manually kept consistent. Consequently, it cannot be generally guaranteed that the application will run on the real embodiment as intended, because the virtual representation and embodiment might be too different from each other.

Furthermore, from an architectural point of view, an embodiment is conceptually different from a view on the design problem. A design view is usually more abstract and can be applied across different embodiments. In this sense, an iCat view is not necessarily a useful view for a Roomba embodiment. Also in terms of software architecture and software reuse, embodiment dependencies should be modeled in a single place in order to keep dependencies manageable.

The second possibility, which is used in the architecture, is to define an overall *embodiment definition*. The concept was reused from the OPPR framework. The embodiment definition describes an abstract setup of the hardware in terms of available sensors and actuators. Using this approach, two embodiments that have the same embodiment definition can be treated as being the same embodiment. Therefore, an application is only bound to a particular embodiment definition instead of a concrete embodiment. Two embodiments that differ in implementation may still have a comparable physical setup. For example the virtual representation of a physical robot has the same controls, but different hardware.

The advantage of this approach is that an application is developed for a whole class of robots that exhibit a specific physical setup, rather than for a particular robot. For example, iRobot offers different models of their vacuum cleaner Roomba, which have a similar differential drive system consisting out of two wheels. A similar result could have been achieved by defining a new generalized class of Roomba-like robots, from which all concrete Roomba models are derived. However, following a strict object oriented design, for every new generalization a new class would have to be introduced to represent that two embodiments have the same type. In consequence, all existing embodiment classes would have to be changed to be derived from this newly introduced generalized class. The concept of an embodiment definition decouples the hardware setup from the implementation of the embodiment and avoids therefore an exponentially increasing implementation effort for introducing new embodiment types.

6.7 Component developer interface

In the above described architecture, flexibility was achieved by defining a general component model. Several components for the Development-Environment have been introduced. One advantage of the modularized approach is that the architecture can easily be extended by new components. The SRD architecture provides two general possibilities to define new components.

The first possibility is to provide a library containing executable binary code that complies to the general component interface. The easiest way to achieve this is by deriving a new subclass from the central component class of the Development-Environment. The instantiation of the object is decoupled using the factory pattern as described above. During run-time,

the central handler `iProcess()` is invoked according to the synchronous component model as described above.

The second possibility is to provide a loosely coupled component using a socket interface. Requirement 4.2 demanded that it should be possible for third-party developers to provide new components for the architecture. The definition of a general component interface and the decoupling from instantiation already fulfill this requirement. For convenience an abstract socket component has been defined. The socket component is a host for remotely connected components that routes messages over a TCP connection. The socket component can be controlled using a simple command language and is integrated in the synchronous execution cycle of the main-loop of the architecture. The command language contains messages for starting of the execution cycle and implements generic calls to access the blackboard communication system in term of read and write access. For convenience, the syntax has the form of Lua scripts in terms of functions that are executed. The return values of the functions are casted to a string and sent over the TCP connection. Using this remote interface, a component developer is not restricted in his choice of programming language, operating system or development tools, as long as his tools provide the generic possibility to open a socket and send string data.

The new component may even run on a remote machine communicating with the socket host over a network. For the Development-Environment there is no semantic difference to the execution on the same machine, because of the synchronous execution environment. Therefore, the socket-host provides the possibility to export computationally heavy modules to remote machines. In turn, the computation delay of a processing cycle is reduced to the communication delay of the socket interface.

A specific concern for the development of new components is the availability of logging facilities. Due to an increasing number of components it becomes more cumbersome to trace logging output of a particular component. For this reason, a general logging mechanism has been introduced, similar to the logging mechanisms *Log4j*[4] or *Log for C++*[5]. An overview of the logging mechanism is depicted in Fig. 6.23. Every component within the architecture is equipped with its own logger. The logger class defines 5 different logging levels, which are 1) Debug, 2) Info, 3) Warn, 4) Error, and 5) Critical. A module can therefore use prioritize messages according to how serious a certain event is for continuing execution. For example,

[4]http://logging.apache.org/log4j/1.2/index.html
[5]http://log4cpp.sourceforge.net/

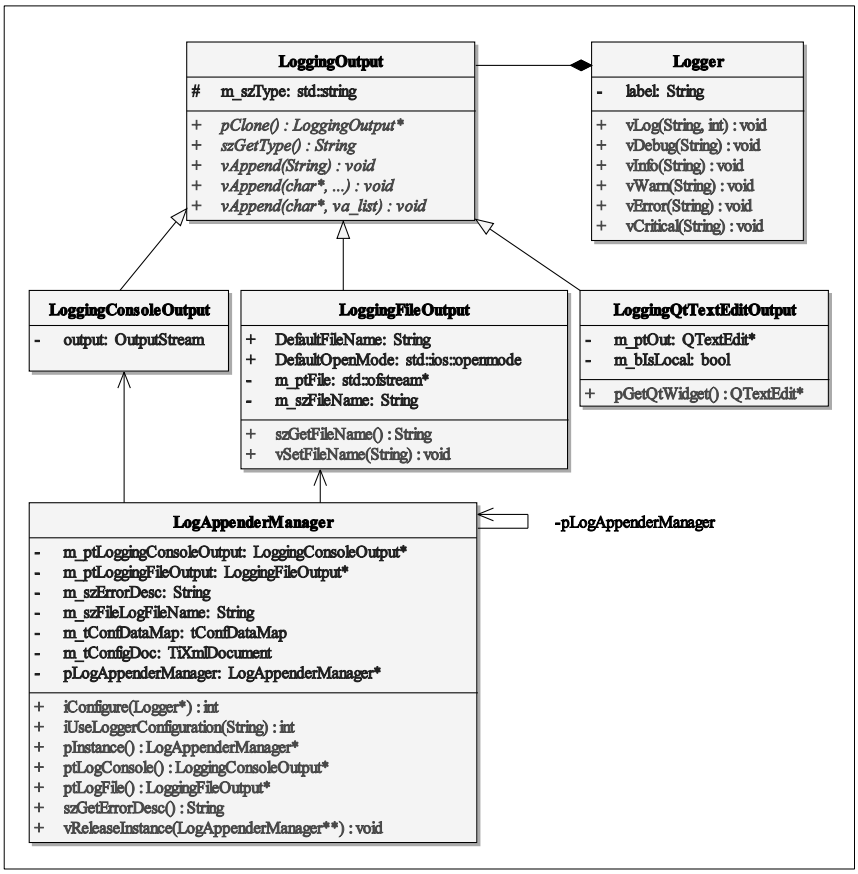


Figure 6.23: Logging mechanism within the SRD architecture

internal messages can be flagged with Debug. Messages that are intended to be seen by a user only for information purposes fall in the Info category.

A logger delegates the output to an internal LoggingOutput. This might be a console output, a file or a graphical widget. Every logger defines a label that uniquely identifies the source of the message. With every message the message source and a time stamp are recorded as well. The output of the logger can be centrally configured by the `LogAppenderManager`. The system can be configured to keep track only of a number loggers. Furthermore, for every logger the log level can be set, which means only messages that have a priority above the log level are recorded. This system allows for an easy trace of the progress of a particular component.

All the different components may have a representation in the graphical user interface for the application developer. An overview of the overall layout is given in the following.

6.8 Graphical User Interface

In order to realize a flexible and customizable visualization, a layered architecture has been chosen. The layered approach abstracts the graphical representation of components from their functionality. A clear separation of functionality and visual representation allows easy replacement, for example with a different GUI library. An overview of the layered architecture is shown in Fig. 6.24. The lower boundary box contains the core functionality that is to be represented with the graphical user interface. The functionality is decoupled, so that the visualization may change without affecting the functionality.
The upper boundary box contains the classes that realize the visualization. A central GUI object has been defined, from which all other objects for the visualization have been derived. Every component of the Development-Environment that needs a visual representation is represented by an object. A graphical component is derived from the central GUI object class, which contains a link to the object that is to be visualized. This way, the object can map incoming events, including mouse or keyboard events, to trigger the functionality of the underlying object. The overview in Fig. 6.24 indicates that the central engine has a graphical representation in the main window of the application. Furthermore, common graphical functionality has been decoupled using the decorator design pattern (depicted in Fig. 6.25) [78]. The decorator pattern describes a method to extend the functionality of a class dynamically, without having to generate new subclasses. Instead, the new functionality is coded in a decorator that surrounds the original component. The decorator pattern therefore avoids an exponential number of classes for the realization of the visualization. Another advantage is that it facilitates to reuse graphical features. A scroll-bar decorator can be applied to more than one class, so that all scroll-bars in the framework will always have the same appearance and behavior.
An example layout of the graphical user interface of the Development-Environment is depicted in Fig. 6.26. It allows customizable layouts so that application designers with different preferences can adjust the Development-Environment to their needs. For example, the designer might choose a layout using a single main window, or free placement of dialogs on the

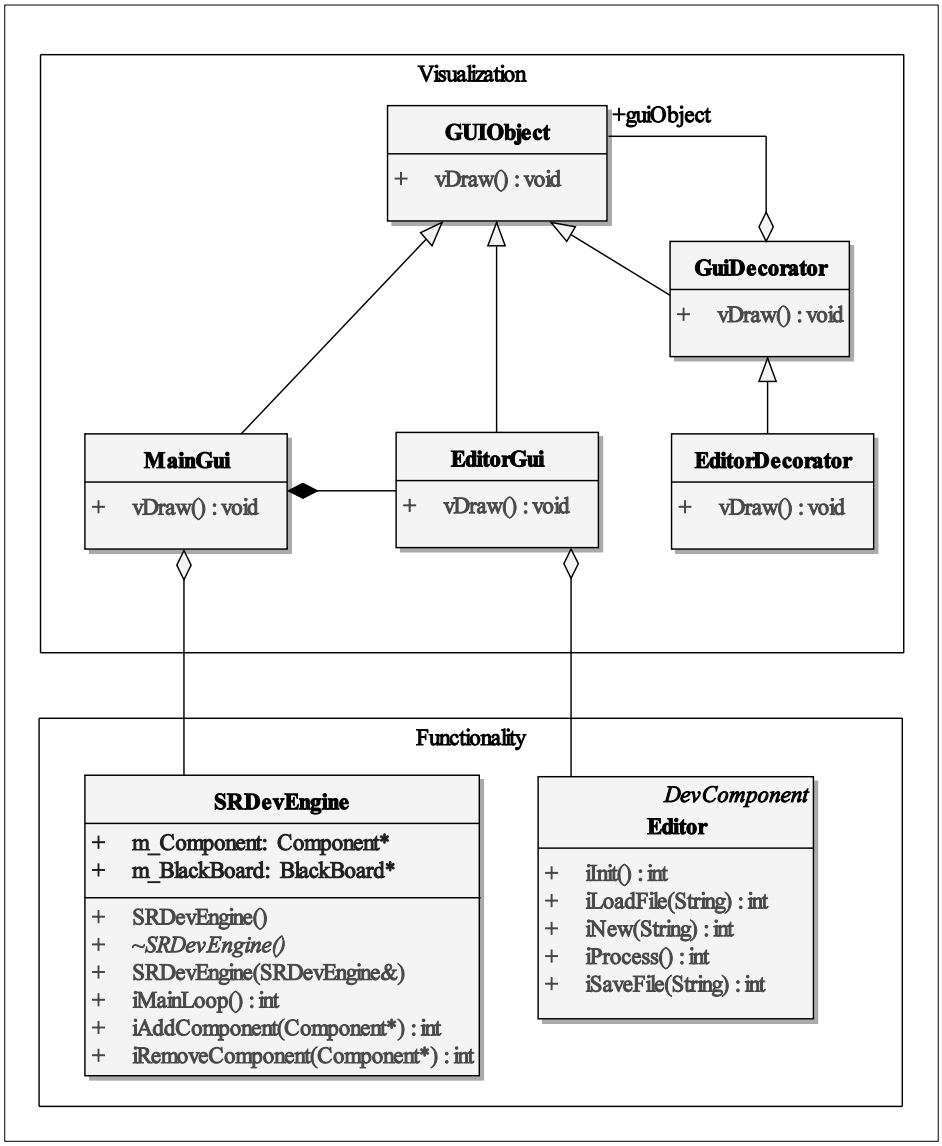


Figure 6.24: Layered architecture to separate visualization from functionality

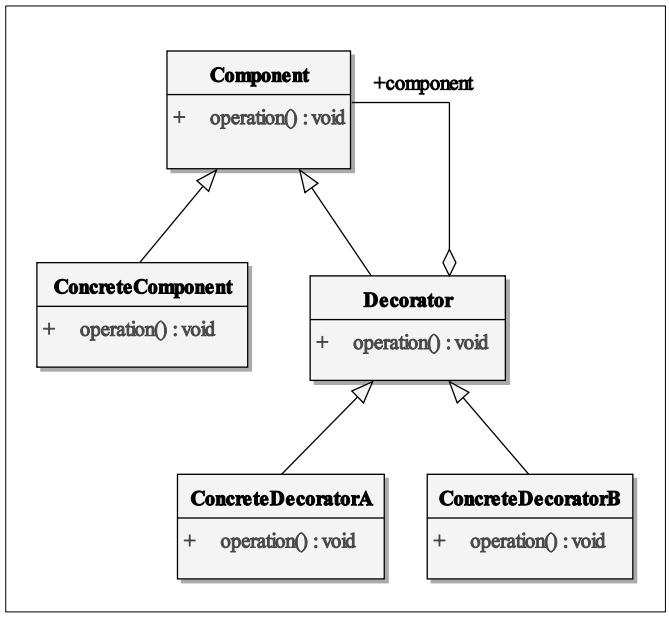


Figure 6.25: Decorator design pattern

desktop. The standard mode for Windows® environments is to arrange interface elements inside one main window. Sub-windows are maintained inside the boundaries of the main window. In Linux environments, however, multiple specialized dialogs are often provided as top level widgets, which allow the window manager to control placement of the dialogs. These differences become most obvious when comparing the image editing programs Adobe Photoshop® [6] and the GNU Image Manipulation Program[7] (GIMP).

In the current implementation, a component is visualized by its own docking window. It has been developed using the Nokia interface library QT[8], which is flexible enough to suite multiple desktop arrangements and furthermore is available on multiple platforms, including Windows® Linux and Mac OS® X environments.

For placing the different windows inside a main window, four docking areas around a central widget are supported as displayed in Fig. 6.27. Multiple dock windows may be placed in the same docking area, where they are

[6] http://www.adobe.com/products/photoshop
[7] http://www.gimp.org/
[8] http://qt.nokia.com/

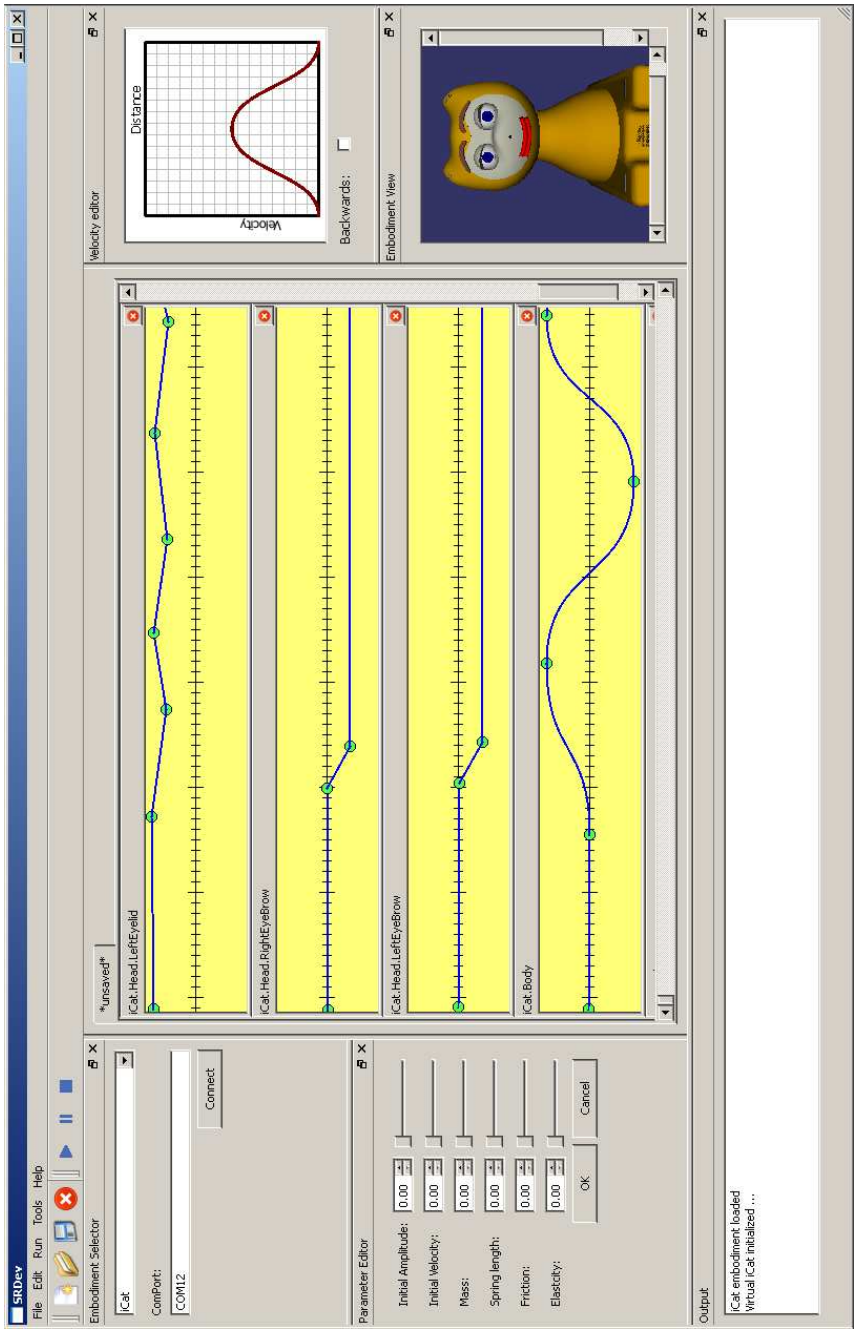


Figure 6.26: Example layout of the graphical user interface

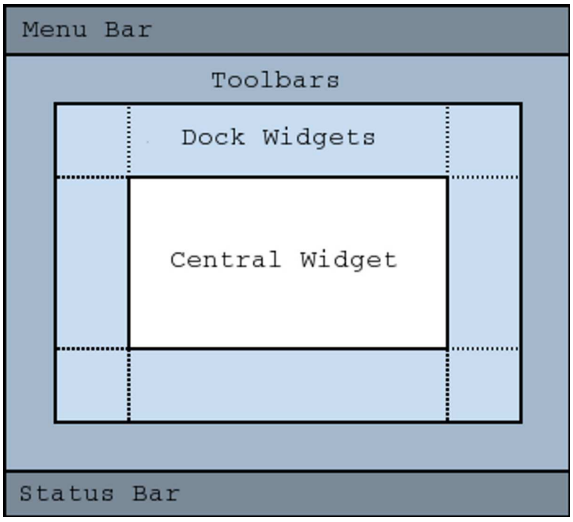


Figure 6.27: Docking areas inside a QT main window. Source: Documentation of QT library (http://doc.trolltech.com/4.4/qmainwindow.html)

either vertical or horizontally arranged, if all of them are visible at the same time. Alternatively, they can be stacked on top of each other and are accessible through a tab interface.
The screen shot of Fig.6.26 displays in its central widget a functional animation editor for the iCat embodiment. In comparison, Fig. 6.18 on page 130 showed a screen shot of a path animation using the path editor. Multiple animations can be edited at the same time and are accessible through the tab interface displayed at the top of the central widget.
For example, the left dock contains a dialog to select a target embodiment and a parameter editor to modify the shape of a trajectory, arranged vertically. The displayed parameters control the behavior of a simulated second order differential system and control the shape of a selected interpolation. The right area of the screen displays a velocity view on the motion trajectory of an actuator and a preview facility to simulate the behavior on a virtual robot. The bottom dock area contains a central output widget. Status and debug information are displayed in this area.

6.9 Summary

The above described architecture introduces a generic and flexible software environment for the design of social robots.

Starting with the identification of an appropriate reference architecture, the structural decomposition of the software architecture has been presented. A combination of editing- and language processing system was modeled to address the requirements of the design task. Furthermore, a unified component interface has been developed, based on the commonalities of the elements of the Development-Environment. For communication between components, it has been motivated that a synchronous communication model best suits the demands for the Development-Environment. A central blackboard system has been modeled using scripting technology, which allows to define relations between variables and flexible sharing of data.

The component structure has been elaborated to define central constructs such as the concept of an editor, embodiment, and preview facility. Three concrete instances of editors have been developed. For expressive behavior design, the functional animation principle has been applied to animate the iCat and Roomba robot. For the iCat robot, the editor provided direct control over the position controlled actuators. The advantage of this approach is that it gives the designer direct control over the expressiveness of the robot, but preserves high level control through the parameterization of the functional animations. For the Roomba robot, on the other hand, it was motivated that an abstraction on the level of path segments is more appropriate to create expressive and functional behavior.

These graphical editing concepts provide an animation designer with familiar editing concepts. For describing the application logic and modelling the interactivity a textual representation was chosen, because it allows to capture abstract behavior rules and algorithms that define how the robot reacts to a certain input.

In the process of defining the architecture, several software design patterns have been applied in order to increase maintainability of the software and to decouple close relationships. One design pattern that reemerged is the usage of a component definition. This general relationship is depicted in Fig. 6.28. The pattern decouples object specification from its type definition. For example, for the modeling of embodiments, an embodiment definition was introduced that allows to represent different embodiments with similar physical setups. Furthermore, the pattern was used to give concrete information about components, e.g. which variables are written on the blackboard so that a module hierarchy could be established. Encapsulating the information about an object in a separate object allows the comparison of objects decoupled from its type definition.

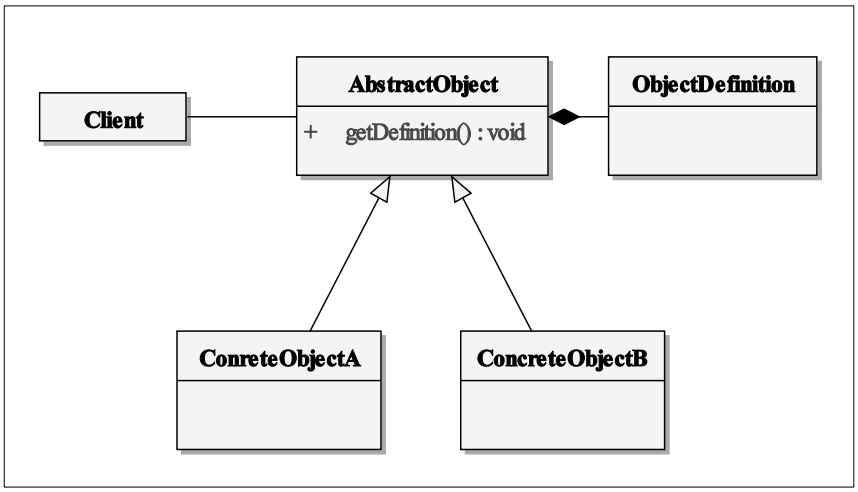


Figure 6.28: Object definition pattern

Having described an architecture for a designer to create social robots, the following chapter discusses an Execution-Environment that is used to render the applications on the deployment hardware.

Chapter 7

Execution Environment

In this chapter the architecture for the *Execution-Environment* of the Social Robot Development Framework is described. This chapter extends the general concepts that have been defined in the last chapter. In contrast to the Development-Environment, which had the main focus to support an application designer with design tools, the main goal for the Execution-Environment is to provide a deployment environment for applications to be used by a user. The Execution-Environment has therefore to satisfy a different set of constraints, including constraints from the hardware, user-interaction and reliable application rendering. Reliable application rendering means that the application is rendered as the designer intended, that is minimizing the differences between design views and the resulting behavior of the deployment embodiment.
First the design space of the Execution-Environment is analyzed and relevant architectural differences between the requirements for the Development-Environment and the Execution-Environment are identified. Based on this analysis a general component architecture and communication model is developed. Furthermore, this chapter discusses the challenges for reliable rendering of applications in detail, before the overall architecture is evaluated in the next chapters.

7.1 Design space

The overall goal of the Execution-Environment is to provide a deployment environment for applications that have been developed with the tools from the Development-Environment. The requirements of the two environments differ in three main dimensions, i.e., in terms of key stakeholders, mode

of operation and hardware constraints. First, the main stake holder of the Execution-Environment is a user who uses the application. In consequence, the Execution-Environment does not need to offer development tools such as graphical editors to create animation artifacts. Instead, the Execution-Environment needs to read these artifacts and render them in a timely manner as defined by the particular application.

Secondly, the mode of operation differs from the Development-Environment, where the main control is residing on the side of the designer who triggers actions in the design tools. In return the Development-Environment has to respond accordingly and to update various views in order to give appropriate feedback to the designer. For the Execution-Environment the main control is in the hand of an application script. The application defines when to render application artifacts such as animations and how to react to user input.

Thirdly, the Execution-Environment has to deal with different hardware requirements than the Development-Environment. While the Development-Environment could presume a desktop computer, the Execution-Environment is limited to the hardware of the robotic embodiment. Usually, the hardware constraints for a robot are more restrictive than for a desktop environment. Additionally, for robotic embodiments no standard set of interaction devices is defined such as mouse and keyboard for desktop environments. This lack of unification across robotic embodiments makes an application very dependent on the available hardware and hence poses additional challenges on the Execution-Environment.

In the following, a general use case scenario for the Execution-Environment is derived from the overall use case scenario of the SRD framework for the application of the Execution-Environment (see Fig. 5.2 on page 85).

7.1.1 Use case scenario

The overall use case scenario for the SRD framework that was introduced in Chapter 5.2 already identified the user as an actor who uses the application. For the Execution-Environment, the important relation is the interaction between application and a user of this application as depicted in Fig. 7.1 This refined use case scenario introduces the robotic embodiment as an autonomous actor who participates in the use case of executing an application. The robot serves as an interface for the application and is therefore part of the Execution-Environment as indicated by the system boundary.

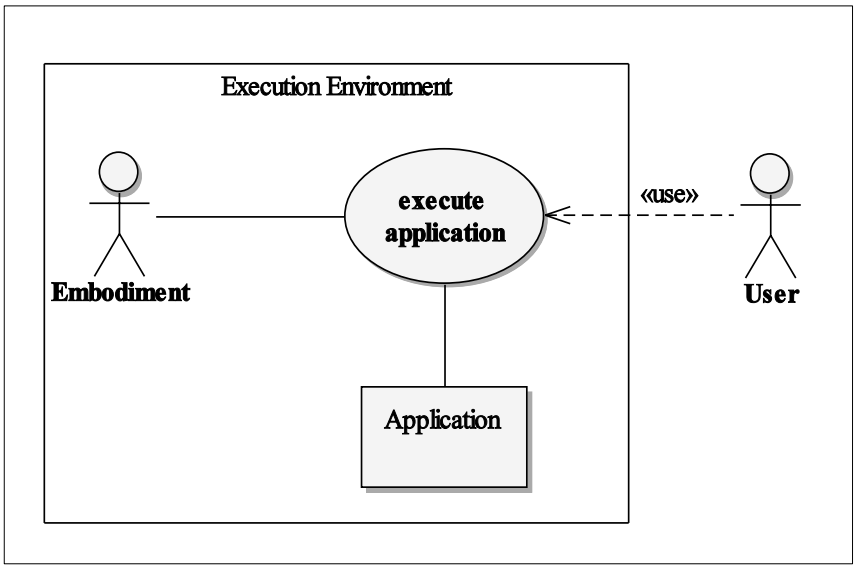


Figure 7.1: Use case scenario and system boundary for the execution environment

In the most generic form, the deployment environment of the robot contains a set of sensors, a controller and a set of actuators, which in total is referred to as the robot embodiment (see section 6.6). However, no further generic assumptions can be made about a particular setup of the robot.

During an execution of an application, the user interacts with the robot by means of its sensors and actuators. The internal control is not visible for the user. The sensors and actuators of the robot constitute the interface. In terms of information flow, the robot uses the actuators to perform actions or to convey certain messages to the user. In return, the responses of the user are perceived through a set of sensors. Based on the current state of the application the next actions are selected and executed.

Despite their different hardware environments, Execution- and Development-Environmentshare several concepts with each other. A major requirement for the overall Social Robot Design framework has been flexibility and maintainability of the software. The amount of changes that are necessary to realize a certain feature have to be reduced.

7.1.2 Software reuse

The need for software reuse has long been recognized in software engineering [160]. However, even though there are plenty of benefits to software reuse such as accelerated development, reduction of costs, increased reliability and exploitation of expertise, software reuse remains difficult. A major obstacle for software reuse is that software components are usually specialized for one particular use case. Generalizing components to enable them to serve for multiple contexts requires additional specification work, which for most software projects is out of the scope of its initial specification. In literature, several methods for effective software reuse have been proposed. Sommerville lists several approaches that support software reuse:

- Design patterns
- Component-based development
- Application frameworks
- Legacy system wrapping
- Service-oriented systems
- Application product lines
- Commercial of-the shelf (COTS) integration
- Configurable vertical applications
- Program libraries
- Program generators
- Aspect-oriented software development

(Sommerville [227], p. 420)

In order to guarantee maintainability and flexibility of the overall SRD framework, one design requirement is to develop one consistent set of components for both, the Development-Environment and the Execution-Environment. As has been shown in Fig. 5.3 on page 96, the Development-Environment and the Execution-Environment intersect with all components that are related to an application. Increasing the number of components for this intersection therefore reduces the number of software change that is required for implementing a new feature for both components. Software reuse across both environment has been an important factor for increasing maintainability and flexibility of the overall architecture

7.2 Component model

In this section the overall component model of the Execution-Environment is introduced. First an overall reference architecture is selected. The refer-

ence architecture is then further refined to meet the particular requirements of the Execution-Environment.

7.2.1 Service oriented architecture

For the Execution-Environment a service oriented architecture (SOA) has been selected as a reference architecture. Service oriented architectures have been developed to facilitate software integration across organizational boundaries. In particular, SOA provide means for developing distributed systems of which every component is a stand-alone service.
SOA is a generalization of a client-server architecture as it is widely used for the Internet. The major difference between servers and clients is that servers may not ask clients for services. This restriction has been removed for SOA. Objects may be distributed across a network or running locally on the same machine. The service oriented architectures received increasing attention in recent years because it promised several key qualities that are relevant for business [227]. However, as Sommerville stated in 2007:

> Because service-oriented software development is so new, we do not yet have well-established software engineering methods for this type of systems. (Sommerville [227] p. 747)

The main driving forces have been to reduce the development time, reduction of software costs and increase of software quality. Especially for business applications the loosely coupled integration is an essential feature, because it allows outsourcing of the development of certain software components to specialized companies. Furthermore, parts of the overall system can be easily replaced by a new service from a different vendor. Also for vendors of services the service oriented architecture model is beneficial because it allows then to offer services to multiple customers.
In Fig. 7.2 a generic architecture overview of a service orienteted architectures is given ([227] p. 286). The central concept of the model is a `Service` that is provided by a `ServiceProvider` and used by a `ServiceRequestor`. The `ServiceRegistry` is a central point that manages the establishing of a connection between the two.
The service oriented architecture model was chosen for several reasons. First of all, it allows for great flexibility in the overall Execution-Environment by integrating stand-alone services from different vendors. This enables a modular setup of the robotic system, e.g., by integrating sensors and actuators as independent services. For example, a camera offers a

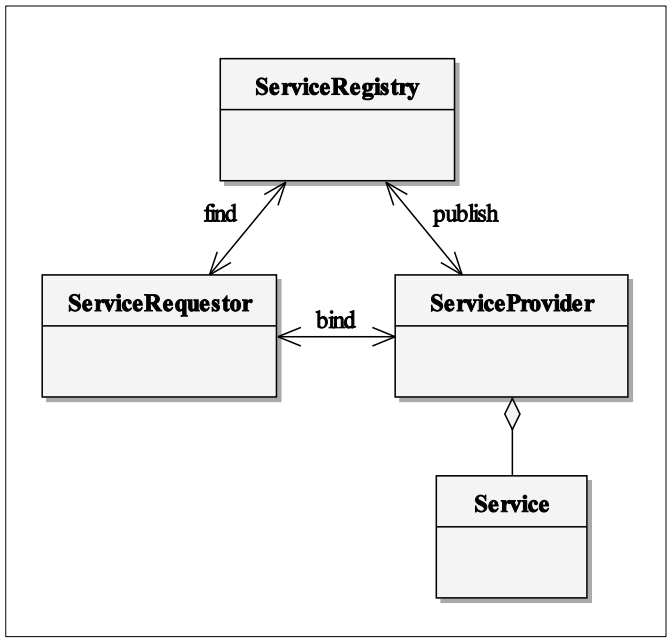


Figure 7.2: Service oriented architecture

service to provide an image, and an image recognition service provides the service to process the image. This setup is later explained in detail.

Secondly, distributed components as defined by a service oriented architecture are highly configurable also during execution. It allows to dynamically allocate system resources to computational tasks for the application. For example, it allows to temporarily halt not required components or to dynamically restart parts of the system after a component failure. In the OPPR system, these tasks were enabled by the DML middleware, which belonged to the architectural package of the OPPR framework.

Thirdly, every component may run on its own terms and is therefore independent from the rest of the architecture. This affects the design of the communication architecture. In contrast to desktop systems that are usually modeled as one processing entity, robotic hardware often consists of multiple independent micro-controllers. This setup is best represented by loosely coupled components which cooperate over a network. The difference for the communication architecture are outlined in the next chapter.

7.2.2 Asynchronous communication model

For the Execution-Environment an asynchronous communication model rather than a synchronous model as for the Development-Environment has been chosen. A communication event is said to be asynchronous

> ... if no mutual interference is caused by action of communication. (Simpson [223], p. 35)

While the Development-Environment could benefit greatly from a synchronous component model, this approach is not suitable for the Execution-Environment. The main reasons for this are timing constraints for a robotic system. A robotic system is an interactive system with real-time constraints. Sommerivlle defined real-time systems by:

> A real-time system is a software system where the correct functioning of the system depends on the results produced by the system and the time at which these results are produced. A soft real-time system is a system whose operation is degraded if results are not produced according to the specified timing requirements. A hard real-time system is a system whose operation is incorrect if results are not produced according to the timing specifications. (Sommerville [227], p. 340)

A robotic system needs to be able to respond on demand as soon as an event occurs. There are two types of events that can occur, periodic and non-periodic events. Periodic events originate from sensors that produce a periodic stream of data such as a camera that produces images with a certain framerate. Non-periodic stimuli, on the other hand, can occur at any time. Examples for such signals are bumper events that are triggered when the robot collides with an object in the environment. The timing for the response can be critical to prevent the robot from causing damage to the environment or to itself. However, in order to avoid response delays due to computationally expensive signal processing, a reactive control might directly connect 'Data collection' to 'Actuator control'. The mere presence of the bumper signal can be used to trigger a response in the motor control. This type of direct linkage between stimuli and action is difficult to realize with a synchronous system because of the time constraints for the handlers in the main loop of the synchronous architecture. The frequency by which a handler is processed depends on the sum of processing time of each individual handler that is registered in the main-loop. A delayed execution of event

handlers could potentially have negative consequences for the operating security of the robot. For the Development-Environment the synchronous approach helped to synchronize multiple design views, but the focus for the Execution-Environment is to react to a particular event as quickly as possible. Therefore, an asynchronous decentralized communication model is better suited than a synchronous model.

Furthermore, the asynchronous architecture is enforced by the typical hardware setup of robotic systems. Robotic systems are often controlled by a set of individual micro-controllers with no central clock, which allows true parallel processing of data. For example, Ananda and Tay stressed the importance of asynchronous communication for fully exploiting parallelism in a survey of multiple systems for asynchronous procedure calls [3]. They argued that in a synchronous approach mutual synchronization events cause delays and thus fail to optimally utilize available system resources.

Asynchronous communication can be realized using different communication models, including message passing, remote procedure calls or shared memory. The selection of a particular communication model has also consequences on the rest of the architecture. For example, remote procedure calls introduce a tight coupling between system components, because a client has to know about the existence and location of a function before it can be invoked. Event broadcasting, on the other hand, allows a loose coupling, because the sender has no knowledge about recipients of a message. However, in this setup it is difficult to guarantee a reliable message passing because no feedback is given if a packet was lost due to network errors.

Eugster et al. analyzed the abstraction level of several communication models, including remote procedure calls, message passing, message queueing, shared memory and publish-subscribe mechanisms [67]. For classification they introduced as criteria time, space and synchronization decoupling. Based on these criteria they argued that publish-subscribe mechanisms offer the most flexible and scalable abstraction for communication in distributed systems. Therefore, the publish-subscribe mechanism has also been adopted for the Execution-Environment. In order to avoid dependencies on a particular middleware or programming language, a general communication protocol has been defined on-top of a message passing system. The details of the communication protocol are discussed in the following section.

7.2.3 Communication protocol

In the literature, several publish-subscribe mechanism have been proposed [150]. In their survey Liu and Plale presented a taxonomy of published-subscribe mechanism based on the criteria:1) Subject-based versus content-based 2) System architecture, 3) Matching algorithm, 4) Multicast algorithm, 5) Reliability and 6) Security.
The first four criteria describe the technical realization and the last two criteria judge qualities beyond the core functionality of transmitting data. For the Execution-Environment, especially the performance and reliability of the communication protocol is of importance. The performance is of importance due to the strict hardware constraints of robotic hardware and the reliability is of importance, because the system has to insure that critical events such as a bumper event do not get lost. The performance is mainly determined by the realization as categorized by the first four criteria, i.e., which broadcast strategies are used or which architecture for the components is chosen. For example, an event can be broadcasted to all components in the architecture or sent to another component using a point-to-point connection. Another possibility is to organize the components in a hierarchy so that an event can be forwarded between components along the position in the hierarchy.
Taking the requirements of performance and reliability into account, as well as common hardware constraints and typical use case scenarios, the following communication architecture has been designed. An overview of the communication architecture is given in Fig. 7.3. In order to reduce network load, a point-to-point connection model has been chosen. The connection class is the central concept of the communication architecture. In terms of the taxonomy by Liu and Plale, the selected architecture falls in the peer-to-peer category in which all components may take the role of either a subscriber or publisher. For exchanging data, components establish a connection to each other. The reason for this is that not every component needs to be informed of events in the system. For example, raw sensor data such as a camera image or sound data do not need to be distributed to all other components.
A publish-subscribe mechanism based on individual connections requires a subscriber to register at a publisher and to establish a new connection. This would introduce a tight coupling between sender and receiver. However, this dependency can be avoided using the concept of the service registry from the service oriented reference architecture. In Fig. 7.3 the dispatcher takes the role of the service registry. Every component in the system regis-

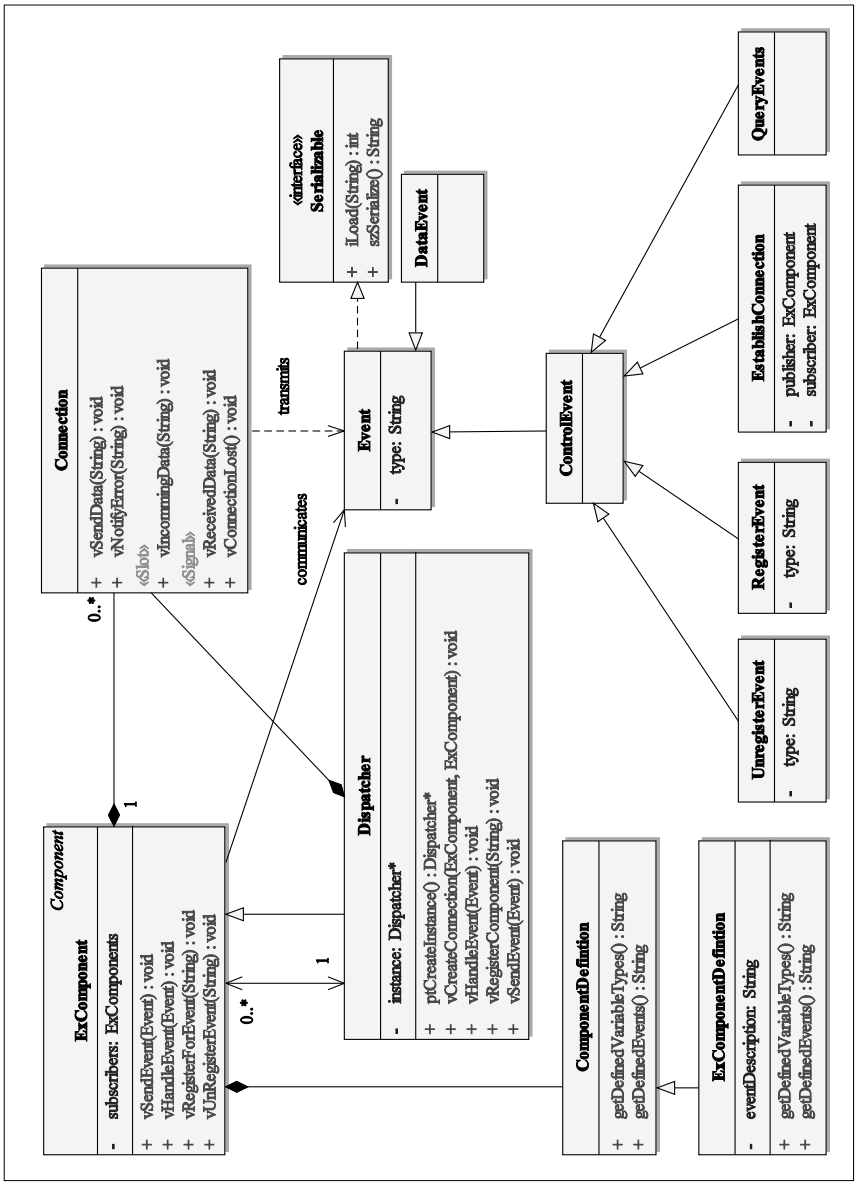


Figure 7.3: Overview of the asynchronous communication protocol

ters itself at the dispatcher with a description of the component using the component definition. The description also contains a list of the events that are provided by the component. A component can query the dispatcher to receive a list of all events that are available in the system. In order to receive an event, the component has to register for a particular event. In turn, the dispatcher establishes a new connection between the provider of the event and the subscriber. This connection is used for directly communicating between the components, i.e., sending the inquired events from the publisher to the subscriber. This peer-to-peer setup avoids a bottleneck that could emerge in a centralized communication model. Arguably, the peer-to-peer setup puts more administrative overhead on the side of a component, because a list of subscribers has to be maintained, but it also shortens communication paths. In terms of reactivity, this setup allows low communication delays because of direct connections.

The publish subscribe mechanism has been realized on-top of message passing using sockets. This design decision offers the flexibility of reliable communication using TCP, but it is still generic so that the architecture is not bound to a special communication library. In the component model of the Execution-Environment, the administration of connections has been encapsulated in a central component object of the Execution-Environment. In Fig. 7.3 this base class is represented by the `ExComponent` (Execution-Environment Component). The component internally keeps track of registered subscribers. The function `sendEvent()` internally posts events to all subscribers that are registered at the moment the event occurs. If no subscriber is registered, the message is simply discarded without producing any network traffic.

The administration of the publish-subscribe system over message passing is realized with special administration events. In the overview depicted in Fig. 7.3, events are modeled as objects with a hierarchy. This allows to assign a type to a message. On the first level of the hierarchy a distinction is made between Control-events and Data-events. Control-events encapsulate all events that are used for controlling the communication mechanism. For example, a component issues a `RegisterEvent` to the dispatcher if it wants to receive events of a certain type. The dispatcher in turn issues an `EstablishConnection` event to the publisher with the address of the subscriber. The event `QueryEvents` is issued to inquire a list of all available event types from the dispatcher. All events are serializable to a string, which is transmitted over the connection. This enables also non-object ori-

ented programming languages to participate in the communication and to filter and react to events.

Another advantage of the selection mechanism is that it can be used for an efficient specification to define the class of events that a component wants to register for. In order to keep the system as flexible as possible, while at the same time reducing the amount of messages that need to be sent on the network, a label based selection of messages was chosen, extended with the possibility of regular expressions. The label specifies the type using a fully qualified identifier. A label based selection mechanism allows a more powerful specification than a purely type based comparison. For example, Eugster et al. presented a comparison on several methods for identification of messages [67]. In the easiest form an identifier characterizes the type of the message that is sent by a publisher. Subscribers use this identifier to express their interest in a certain type of message. The expressivity of this identification method can be extended by hierarchical address spaces and regular expressions such as wildcards for the identifiers. An even more powerful method is to select the desired messages by directly addressing the content of a message, e.g., by checking whether a certain attribute is available in the message or, more specifically, whether an attribute value is in a desired range. All of these methods can be encoded in a label based selection mechanism. In the literature, several special purpose subscription languages have been devised [67]. For convenience and consistency within the SRD framework, XML has been used to serialize objects. Events can be identified using XML XPath expressions [34] which provide a powerful mechanism to identify nodes within a XML document tree.

However, a disadvantage of the label based mechanism in comparison to a fully type based comparison is that it sacrifices type safety during compile time. A trade-of had to be made between flexibility and formalism. In order to allow as many different environments as possible, the label based mechanism was preferred. Requirement 4.4 in particular demanded to integrate with existing environments. Limiting the architecture to only object oriented programming paradigms would exclude all environments that build on imperative or functional programming paradigms.

Having described the overall communication architecture, the following section describes the overall component decomposition of the Execution-Environment.

7.2.4 Component architecture

In this section, the overall component architecture of the Execution-Environment is developed. As has been shown in Fig. 5.3, the Development-Environment and Execution-Environment share the central concept of an application. For this reason, several constructs from the Development-Environment reappear in the Execution-Environment.

First of all, the Execution-Environment is modularized with a central component concept. The overall architecture of the Execution-Environment is depicted in Fig. 7.4. In analogy to the Development-Environment, the Execution-Environment defines a central component class. The component class describes a general interface to which all components in the Execution-Environment adhere. The general component model gives flexibility to dynamically extend and configure the architecture. The component model is driven by the communication mechanism as described in the previous section, but instead of defining a synchronous component execution, the interface provides functions to handle asynchronous events. Instantiation of components is decoupled from the class definition with the factory design pattern.

One generalization that has been introduced to the overall SRD architecture is the global concept of a component from which both, the development component `DevComponent` as well as the execution component `ExComponent` are derived. In both environments the components utilize the mechanism of a component definition to provide meta information that is independent from the type information.

The ExComponent furthermore specifies the interface to extend the architecture with new components. As for the Development-Environment, a component designer might choose to provide a library or a stand-alone module that adheres to the communication protocol as described in section 7.3. However, while in the Development-Environment a special host-component was developed to allow network access to the component interface, this is not required for the Execution-Environment. Components are already decoupled using an asynchronous message passing interface. TCP network access is therefore implicitly available.

The administration of the communication is handled by a dispatcher similar to the one presented in [41]. The event dispatcher maintains a list of all components in the system and thus a list of all events that are available. In case a component registers for a specific event, the dispatcher notifies the corresponding publisher to establish a connection to the subscriber. In

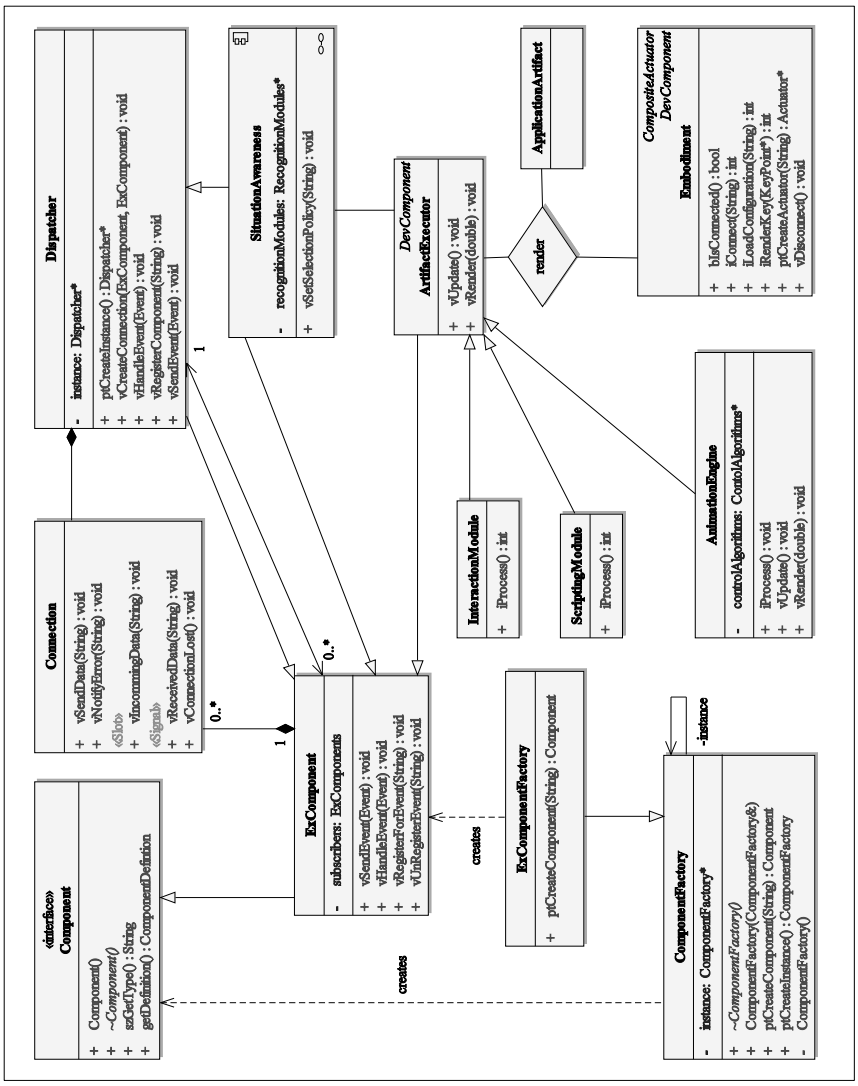


Figure 7.4: Execution environment

turn, the dispatcher only needs to be consulted for control events, for example to register for a new event or to reestablish a connection after a connection failure. Using this approach, the Execution-Environment provides a dynamically configurable framework in which modules can be attached and removed during run-time. Furthermore, it provides an easy debugging mechanism, because it allows to monitor events without interrupting the original data flow.
Using the generalization of an ExComponent, classes of the development environment can be integrated by implementing the asynchronous communication interface. This allows to keep certain functionality encapsulated in one dedicated class. For example, the `Artifact-Executor` that has been used to render application artifacts in the Development-Environment can provide the same functionality in the Execution-Environment. The hardware abstraction layer remains therefore intact. A particular `Artifact-Executor` encompasses the knowledge to decode the application artifact to the next lower level of abstraction. On the lowest level, the `Artifact-Executor` uses the embodiment interface to send commands to the underlying hardware. The modules may even be reused unchanged, except for the communication interface, if the deployment hardware has sufficient specifications. For concrete environments, however, the modules have to be adapted to the underlying hardware. This process is specific to every concrete hardware setup, but it is supported by architecture due to the fact that the specifications are defined by the type of applications that is to be executed. A mismatch between required and available hardware resources can therefore be detected early in the development cycle.
For robotic application developers, the more challenging task is to integrate reactivity to the environment with expressivity of the robotic embodiment. During concrete behavior development, this timely reaction to signals plays only a subordinate role. For the correct execution of an application it is essential. How an application developer can integrate a connection to the sensor hardware within the SRD framework is discussed in the next chapter.

7.3 Integrating context

Interactivity is a key principle for all major robot applications. Interactivity describes the ability to react to the environment. Access to sensor data is not only crucial for interacting with communication partners, but also for low level control algorithms such as a stabilizing routine. The ability of pattern recognition has a major influence on how the user will interact with

a device. Pattern recognition allows computer interfaces to extend beyond current screen based interaction controlled by keyboard and mouse. For example Rauterberg and Steiger integrate multiple interaction modalities to achieve a more 'natural interaction' [196].
Several libraries have been developed that aim to provide high level interfaces, but no unified interface standard has emerged yet. Robotic sensor and actuator components miss widely accepted specifications as are common for PC environments. In consequence, developers of robotic applications have to learn proprietary libraries to interface with a specific piece of hardware. Especially for high level interaction design, this approach becomes unpractical.
In Chapter 3.3 on page 46 three basic design dimensions for creating robotic behavior have been identified. They have been: 1) Naturalness, 2) Adequateness and 3) Development over time. It has been shown that the 'Naturalness' and 'Development over time' dimensions are covered by the functional animation principle. The missing dimension is therefore the 'Adequateness' dimension. The 'Adequateness' dimension goes even beyond simple reactions to sensor data as it already encodes a description of the context that influences the selection appropriate actions. The following section describes the design of a situation awareness model for the SRD framework.

7.3.1 Situation awareness

For successfully maintaining a social interaction with the user, devices have to react appropriately to a given situation [47]. For example, Endsley stresses the importance of situation awareness (SA) for human centered design [66]. For an application designer, the challenge is therefore to connect signals from the environment to the application logic, in order to trigger appropriate responses. That is, the designer has to select signals that are important for a given situation and define rules on how to react to them. The context may include internal values, such as battery power and position of the actuators as well as external attributes such as how many users interact with the robot and the actions of the user. Furthermore, situation awareness supports to establish a common ground to reference objects of an environment during a communication. A common ground is crucial for successful referencing and dereferencing certain objects. Referencing algorithms such as presented by Krahmer and Erk [132] require feature extraction and symbol grounding, thus a reliable representation of the environment.

In the literature, no consensus about a definition for situation awareness (SA) could be reached yet [202]. For example, it is still in debate if SA is solely defined as a set of pieces of information about a state, irrespective of the process that led to the description, or if SA is inherently defined the processes that constructs a representation about a state. The most cited definition of SA is the definition from Endsley, who defined situation awareness as:

> The perception of the elements in the environment within a volume of time and space, the comprehension of their meaning, and the projection of their status in the near future (Endsley [63]).

Endsley distinguished between process and data description by referring to the state with 'situation awareness' and referring to processes involved to acquire an awareness of the situation as 'situation assessment' [65]. These two aspects of situation awareness are often confused with each other.
In this research the definition of Endsley is adopted, because it includes the processes that are necessary to acquire information by perception as well as the resulting state that represent the current situation. However, from a designer's perspective, only the final representation of a state is relevant, because this information is necessary to take decisions on the next actions. The process of information gathering should be transparent to the designer.
In the literature, several computational models have been proposed [6]. For example, Dey presents a data driven modular model in which different steps in the information processing chain are represented by widgets [51]. Crowley presents a conceptual framework and architecture for analyzing situation models and for observing human activities [40]. He argues that a representation of behaviors from people can in return be used to increase a level of SA for an application and to trigger actions.
In her model of SA, Endsley describes three levels of SA, which are connected in a linear hierarchical process [65]. The model is depicted in Fig. 7.5. In this model, decision making takes place outside of SA so that action selection only receives the result from the SA process as an input. This general architecture has been adopted for the SRD framework. Situation awareness is described as the sum of all signals that are captured from the environment at a given moment in time. Using this model, the application designer can describe high level rules for action selection, decoupled from the sensing mechanisms of the robot. The process of acquiring and maintaining a representation of the state can be modeled on a lower level, so that the state representation also forms the interface for the application

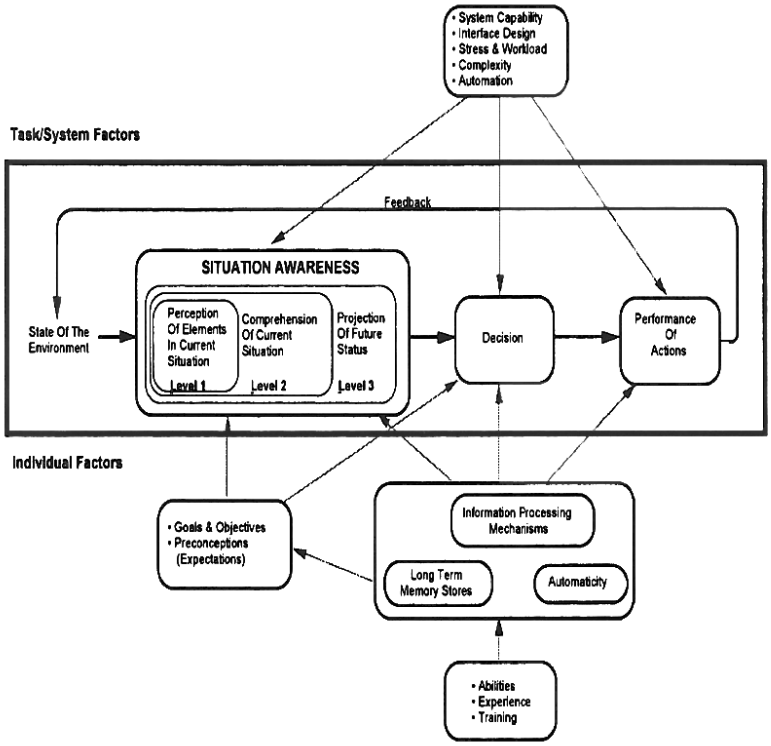


Figure 7.5: Model of situation awareness conceived by Endsley (Source: Endsley 1995 [64])

designer. A general data flow model distributed over layers of abstractions within the SRD framework is depicted in Fig. 7.6. Data originates on a hardware level from sensors. The application designer, in turn, can define high level rules on how to react to a certain stimulus.
For realizing a decoupled situation awareness architecture, a control model, a concept of time and an attention mechanism need to be defined. These three concepts are discussed in the following:

Control The control model defines how data is acquired and delivered to the application logic. A data driven hierarchical process model for SA, such as Endsley's 'perceive-comprehend-project' architecture, defines an one-way flow of information that originates from low level sensors. Assuming that the application logic has control over the execution of an application, it seems straightforward to define an interface to trigger information gathering

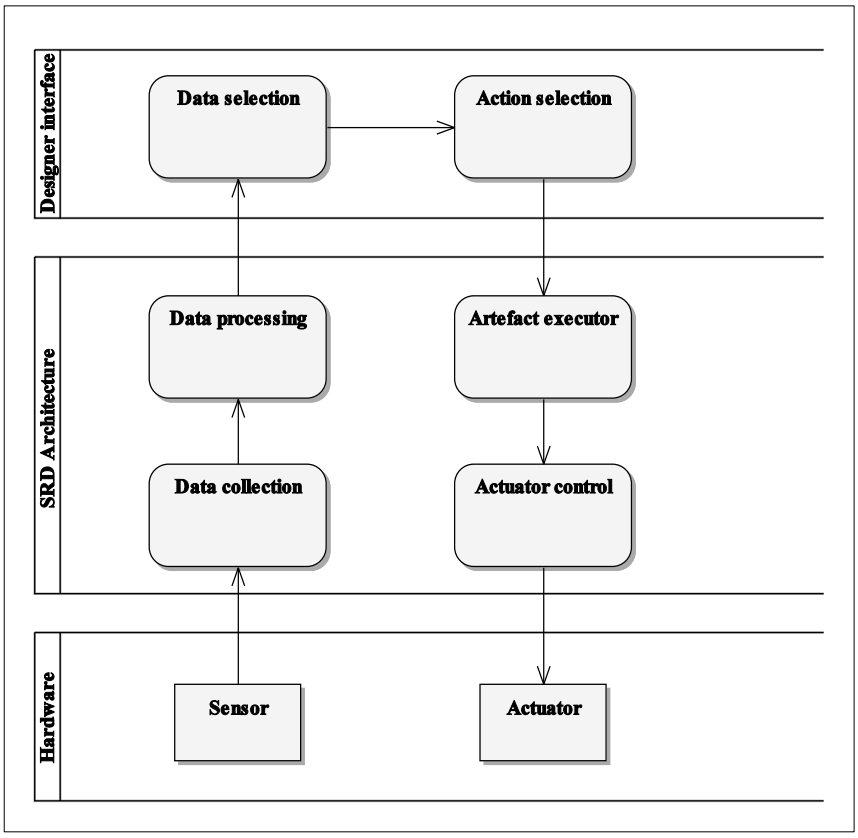


Figure 7.6: Information flow model

procedures form the SA and to query desired information. This model would turn the situation awareness into a passive system that only reacts to requests from the application logic. However, for reacting to sudden and unanticipated events, the SA component needs to be in control and trigger the application logic.

In both cases, the flow of information stays the same, only the initiative changes. In fact, both the top-down and bottom up are eligible control models. That is, SA is at least partly controlled either by a cognitive process or action generation process [187]. For these reasons, both control models have been adopted for the SRD framework. Using the asynchronous publish subscribe mechanism as described above, the application logic may register for events that can occur non-periodically. In the other direction,

the situation awareness defines a control interface for triggering information processing modules.

Time Next to the results of the sensor data processing, situation awareness encompasses values that cannot be perceived at the very moment in time, but are still relevant for describing a situation [159]. For example, an object or person can be temporally occluded from the robot's sensors, but is still meaningful for choosing the next actions. The situation awareness model has to handle these situations in order to provide a high level of abstraction across robot applications and to reduce the implementation effort for the designer.
The majority of models proposed for SA cover the notion of time, but mostly in the direction of the future by projecting a current state to likely future states [65]. Being able to predict future states enables a system to take appropriate precautions and is therefore integral part of SA. While prediction has been part of many models of SA [48, 65], history has often been excluded and analyzed separately, for example in terms of working memory [202].

Attention mechanism Being aware of a situation includes being able to focus only on relevant information in a given situation. An attention mechanism facilitates to efficiently select environmental properties, for example to compensate for hardware limitations in terms of computational power or sensor range. Patrick and James pointed out that before perceiving or examining certain aspects of a situation, a control mechanism has to decide *what* to examine [184]. Humans use a sophisticated filtering mechanism in order to reduce the complexity of processing information. Current sensor technology faces a similar challenge. Sensors are able to record a multitude of information from the environment, but processing this information becomes a problem, especially in real-time applications in which the user expects prompt feedback. Therefore, a SA component needs to be able to focus available computational resources on relevant information.

7.3.2 Situational awareness architecture

A domain model of the situation awareness architecture is depicted in Fig. 7.7. The central concept is an application that runs on an embodiment. The application defines the overall behavior of the robot in terms of autonomous actions, as well as responses to events from the environment. Events that the robot has to react to include, for example, user

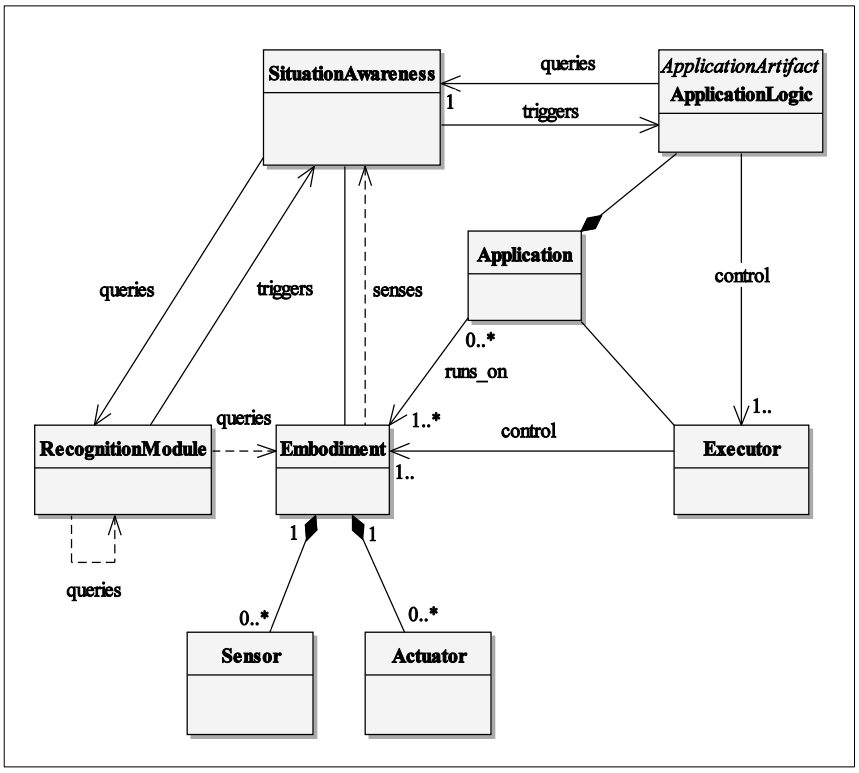


Figure 7.7: Domain model of a situation awareness architecture in the context of an application

events, e.g., by touching the robot or by giving a speech command, and environment events, e.g., a mobile robot that bumps into an object. In order to sense the environment, the robot is equipped with sensors such as buttons, cameras or microphones. The sensors build the foundation for a situation awareness as they are the sources for incoming information. However, the raw signal is only for emergency situations directly connected to a behavior, e.g., in case of cliff sensors. For sensors such as a microphone and camera, a considerable computational effort is required in order to extract a desired piece of information. Therefore, the raw signal data has to be processed. A `RecognitionModule` encapsulates signal processing algorithms that provide new, higher level information, extracted from input signals. `RecognitionModules` are hierarchically ordered and can query data from each other. The result of a computation is available to the `ApplicationLogic` through a general `SituationAwareness` interface.

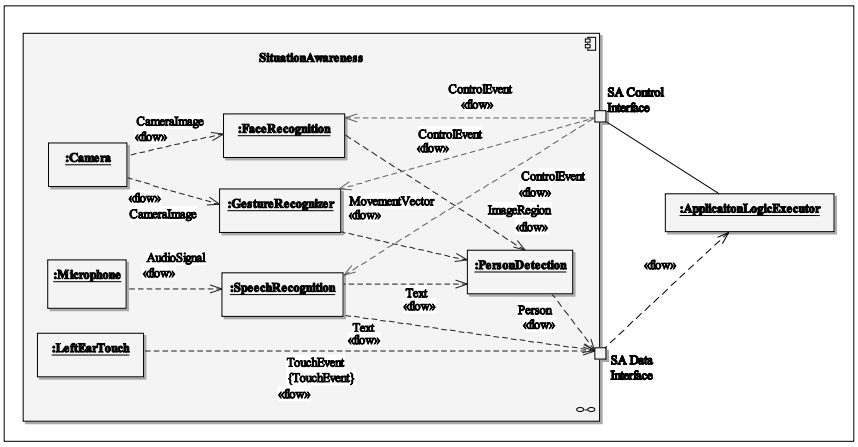


Figure 7.8: Example use case of the situation awareness architecture

An example setup of a situation awareness, of how it might be used for a particular application is depicted in Fig. 7.8. The main direction of data flow in the model is from left to right, with increasing levels of abstraction. It starts at the left with the low level raw signals from the sensors that are passed to recognition modules. For example, the image data from the camera are passed to an instance of a face recognizer and to a gesture recognizer. Every of the recognition modules processes the incoming data and provides new data on a higher level of abstraction. The communication uses the publish-subscribe mechanism as described above. Therefore, all data communication is encapsulated in typed events. For example if a new image is available from the camera, the camera module publishes this image within an event of a unique type. Other components may register to receive these events.

Using this typed system for a situation awareness has several advantages. First of all, the recognition modules can be specified as reactive components with a typed input and typed output. They process the data asynchronously as soon as it becomes available. A general model of recognition module in terms of a reactive system component is depicted in Fig. 7.9. The recognition module implements an input interface that specifies which data it expects as an input. The description is given as an element-selection string using regular expressions. For example, if a component requires an image as input this string specifies the identification label of camera images. Analogously, a recognition module provides a type description of the data that it is able to compute. In the case of a face recognizer that might be an

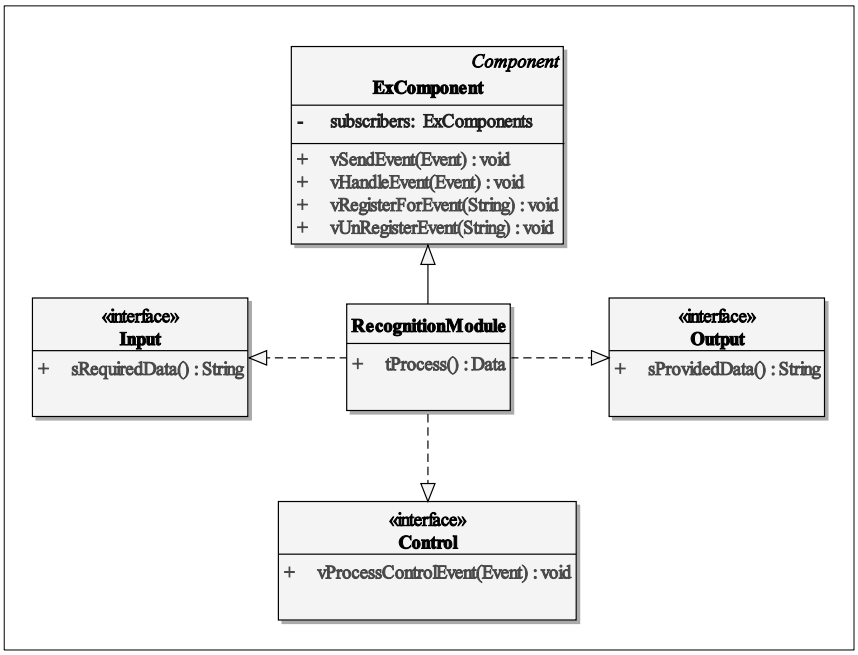


Figure 7.9: Recognition module as reactive system component

image region. In the case of a face identifier that might be a person name. This mechanism provides a flexible interface for specifying new types of recognition modules. For example, a module to find an interaction partner might be interested in face regions above a certain size, so that the person is close, and additionally require a transcript of speech commands in order to asses if the user is addressing the robot.

Secondly, the typed system allows to specify a hierarchy of components. A component has a set of required and provided interfaces [69]. A component that only relies on raw sensor data has a lower position in the processing hierarchy than a module that requires high level data abstractions. Furthermore, this hierarchy can be constructed automatically by the situation awareness component. To this end, the situation awareness component maintains a list of all recognition modules that are available to the system. If the application logic requires a certain data type, the situation awareness searches through the list if any of the modules provides the requested data as output. If one component is found, the situation awareness subsequently tries to resolve the input requirements of this module down to sensor level. If no configuration could be found, the missing data type is indicated to

the developer. If, on the other hand, multiple configurations are possible, a selection policy can be set, based on meta information of the concrete recognition modules. Meta information might include computational performance, recognition quality or even software vendor. A simple matching procedure is applied analogously to the event selection mechanism. If still multiple configurations are possible, a random configuration is chosen.
Thirdly, the typed interface abstraction allows different software vendors to provide new recognition modules that can be seamlessly integrated in the environment. In fact, as soon as the new module is registered to the situation awareness module, it may be integrated in the signal processing chain. Registration is in the current version is realized by loading the available modules from a directory. However, the interface also allows the modules to be registered as web-services from a remote location. Configuration of the modules can be handled using control events. The actual implementation and location of the recognition modules is transparent to the application designer. Analogously to the dispatcher, the situation awareness component can be queried for all available data types. The application designer may use this information for example to see if a face recognition module is available and subsequently connect the input to a gaze behavior that keeps eye contact with the user.
All calculated data from the recognition modules is available to the application using direct connections of the publish-subscribe mechanism. These connections are established by the situation awareness component as soon as the application logic registers to receive a certain data type. From an architecture point of view, the situation awareness component performs similar functionality as the dispatcher and is therefore modeled as specialization of the dispatcher. It organizes data connections between sensory and data processing units to subscribers of the data. As for the general communication architecture, this avoids bottlenecks in the network by distributing connections locally between components.

Wizard interface

A particular instance of the above described general situation awareness framework is used for realizing a wizard-of-oz interface. In requirement 3.4 it was demanded that a particular use case of the SRD framework is to prepare controlled user studies in the field of HRI. The main challenge for the design of an experiment is to vary only the parameters that are to be researched and keep all other factors constant. One of the main points of uncertainty is the variance of quality of recognition results of signal

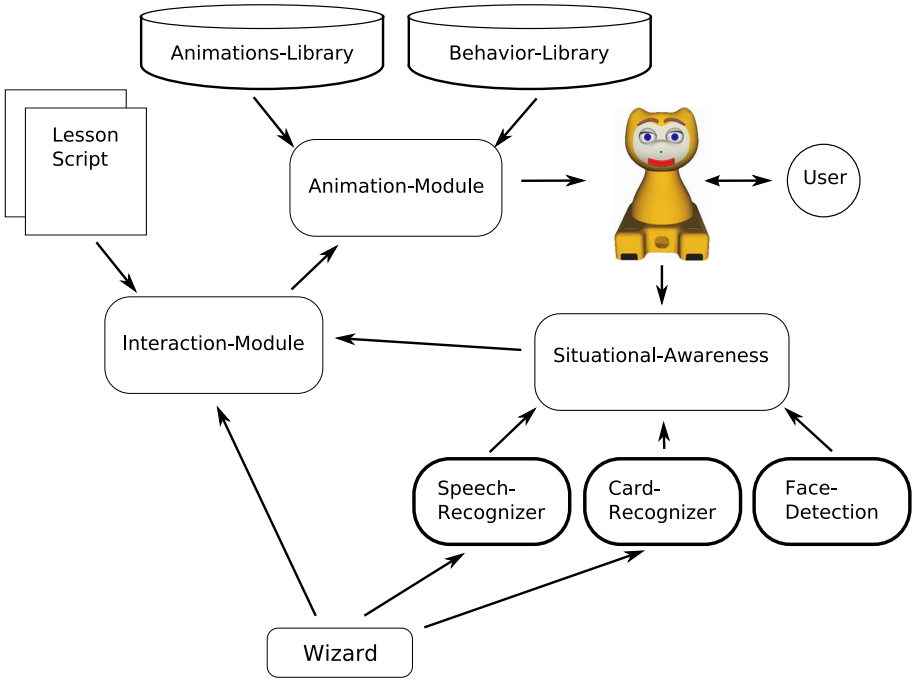


Figure 7.10: Architecture of the tutoring application

processing algorithms such as speech or face recognition [255, 72]. The uncertainty increases for less understood fields such as emotion recognition [38]. These factors have to be controlled in user studies.
The reference system for all social and emotional interaction is still the human. Therefore, wizard-of-oz experiments are a common experimental setup in the field of HRI, because it allows for quick prototyping and insures controlled responses to input from participants. Within the SRD framework there are two main entry points for inserting a wizard interface. The first possibility is to externally control the application logic, for example by a control GUI that lets the experimenter decide on which actions to perform next. In the second possibility, these interfaces are modeled as recognition modules within the situation awareness framework. An example of this setup has been used for a user study with an educational robot [211]. A high level overview of the architecture is given in Fig. 7.10. This architecture made a controlled experiment possible, avoiding the confounding factors of speech and card recognition failures.
However, while the perception of the robot can be controlled using the wizard interface, it cannot be controlled how particular behaviors are rendered

by the hardware. For example, if a robot uses motion to express emotions in the behavior it is essential that the hardware accurately performs specified motion paths. Naturally, the accuracy by which an action can be performed depends on the underlying hardware. For position controlled robots, such as iCat, accuracy is less a constraining factor, because the current position of an actuator can be measured absolutely and hence inaccuracies do not add up over time. For velocity controlled mobile robots such as Roomba, however, such an absolute reference system is not available. An additional limiting factor is that consumer robots have strict price constraints which usually conflict with high accuracy control requirements. The next section presents a study on the accuracy of a Roomba vacuum cleaning consumer robot as an example of a low cost mobile platform. It is investigated if such platforms are generally suited to be controlled for expressive interactions. Furthermore, a modular neural calibration method is developed that allows to improve the drive accuracy without adapting either the Execution-Environment or the robotic hardware. Instead, the calibration method can reuse principles from functional animations to on-line update the calibration.

7.4 Robot calibration

The in the following presented study has separately been submitted for publication in the International Journal of Social Robotics.

Abstract – Low-cost behavior based robots have entered our every day homes. Despite their simple control algorithms and low-cost hardware, they are able to perform tasks such as vacuum cleaning or lawn mowing autonomously. Traditional research on mobile robots optimizes for accuracy, coverage and speed, which usually conflicts with the strong cost constraints of such platforms and neglects the social impact of a domestic robot.

In this paper we present a simple on-line calibration method for closed low-cost robotic controllers. We apply a neural network in a model predictive control (MPC) approach to train a relationship between control signals and resulting robot configuration. The system is based on a geometric model of the drive system, removing the need to give a complete description of the dynamics of the system. We demonstrate the effectiveness of

our approach through experiments with Roomba, a commercially available robotic platform for vacuum cleaning.

Low-cost consumer robots have entered our private homes and begin to take over tasks such as vacuum cleaning or security related tasks. Furthermore, researchers explore further application areas such as medical applications, space exploration, health care or as general multi-purpose interface robots [219]. It is foreseen that personal robots become ubiquitous available, not only at our work places, but also as in our private environments.
One of the biggest technical challenges for mobile domestic robots is to operate autonomously in unstructured, dynamic and unpredictable environments. Recent studies have shown that next to these functional requirements robots also need to integrate in the social environment of a personal home. However, most of the current robotic platforms underestimate their social impact and do not address how they are perceived by the user in the development. For example, Forlizzi demonstrated that the domestic robot Roomba had an effect on the social family life [74]. Bartneck argues that researchers have only just begun to understand the social implications of domestic robots [11]. In literature several studies are reported that investigate how humans perceive and respond to autonomous robots. It is believed that the robot's behaviors, especially its movement patterns have an influence on the perception of the device. Already in 1944, Heider and Simmel demonstrated that humans are naturally biased to attribute life-like features such as animacy, intentions and emotions to moving abstract geometrical shapes [104]. In this line of research, Tremoulet investigated the motion features of a single moving dot and found that, among others, the magnitude of direction or velocity change has an impact on the perception of animacy [235]. Reeves and Nass have found that people apply social rules when they interact with media (e.g., through a computer or television) even if they are consciously aware of dealing with a machine [198]. Especially the interaction with robots enhances this tendency, due to their physical presence [193].
From these observations it follows that household robots have an additional requirement of behaving in a way that is interpretable by humans. One track of research explicitly focuses on utilizing social interaction technology as an interface paradigm [28, 47, 114]. Instead of designing for functionality, research investigates on how the robot is perceived. For example, a certain perceived personality can impact the interaction and acceptance of the device [58]. Meerbeek et al. investigated the influence of different personalities of an interface robot with an animated face in a TV-assistant

application [161]. Goetz et al. go one step further, by explicitly measuring the mental model a person has of a robot [88].

New robot control requirements

In the above examples, a major factor in the perception of the personality has been the behavior of the robot, in particular also its motion patterns. While traditional research on robotic motion focuses on optimizing for accuracy, efficiency and speed, research on social interaction technology also investigates how movement patterns can be used to facilitate interaction [238]. The emphasis here is on the user perspective in contrast to the functional perspective. Taking the example of a mobile robot, one of the crucial differences between a functional and user perspective is that from a functional perspective it is only important that a robot reaches a certain position, not how he reaches it. Therefore, a control algorithm may perform additional control movements to closely track a desired path. From a user perception perspective, however, the relative motion patterns have more impact on the perception than its absolute positions [235]. For example, abrupt control movements might give the impression of an insecure or nervous personality which the user may not appreciate.
Given the above observations it follows that also low-cost consumer robots, such as the Roomba vacuum cleaning robot, may benefit from explicit designed expressive behaviors. Usually those behaviors consist out of sequences of relative movements that the robot has to track. Therefore, the robot has at least locally to be able to track a designed path, even if no global positioning is available. Traditional approaches to accurate path tracking often encompass more accurate sensors and actuators that conflict with the hard cost constraints of consumer products.
In this paper we propose a simple calibration method for a mobile robot that improves the path tracking accuracy without requiring any additional hardware, but optimizing for perception of the robot by avoiding additional control movements. Given the application domain of a low-cost consumer robot, we set the following three constraints: (1) The method may not violate the cost-constraints, for example by adding new hardware. (2) The method has to work with closed systems to be generally applicable also to platforms on which there is no access to the internal control algorithm, which also supports a modular design and rapid prototyping of social robots [12]. (3) The local path tracking performance should be improved with a focus on the perception of the movement. The last constraint sets a frame of reference for the application domain. It stresses that the optimization

might not add additional correction movements that, for example, could be interpreted as nervous behavior. In this study we focus on the technical aspects of the problem rather than performing a cognitive experiment. In order to investigate this research question, we developed a simple robot animation software that enables us to create expressive motion paths and render them on a differential drive robot. We demonstrate the effectiveness of the calibration method through experiments with the Roomba vacuum cleaner.

7.4.1 Related work

In literature, a vast body of studies on robotic control is available. A general overview of the field of robotics is presented in [219]. In the following we will focus on the difficulties of robot path tracking and name common approaches to address them.

Sources of error

What makes accurate path tracking so difficult is the fact that the robot controller has to deal with uncertain data. Uncertainty originates from external as well as from internal sources of error [233]. External errors result from the fact that sensors can measure properties of the environment only with a certain degree of accuracy. Internal errors, describe the errors that are being made due to simplifying assumptions, for example about the device's dynamics, geometrical setup or physical interactions with the environment.
Internal errors can further be classified in three categories: (1) systematic, (2) random, and (3) numerical errors [116]. Systematic errors are caused by inaccurate knowledge of the drive system and its kinematic properties. For example, a systematic error is introduced, if it is assumed that the wheels are equal, but actually differ slightly in size or if mechanical inertia are ignored. Random errors are caused by random physical processes such as slip on the floor or inaccuracies in the actuators. The last source of errors, numerical errors, are introduced by digital information processing. Inevitably, numbers can only be represented up to a certain precision, but many kinematic approaches require integration over time, which therefore has to be quantized in a digital micro controller.
In this paper we address the systematic errors that are caused by inaccurate models of the drive system, but in contrast of proposing to change the already existing control algorithm we developed a calibration procedure

that can be added on top of an existing embedded controller. However, the goal is not to develop a high precision drive system so that the robot ends exactly at a specified position, but to improve the drive behavior so that the robot is able to render local expressive animations.

Path tracking

Naturally, the accuracy by which by a path is tracked, very much depends on the available hardware and control software. In this paper we focus only on the software, in particular on the control algorithms.

In general, the control algorithms can be classified in reactive and proactive approaches. A common reactive approach to error correction in path tracking, which is commonly used by commercial applications, are PD or PID (proportional-integral-derivative) controllers [230]. PID controllers linearize the dynamics of the robotic system and construct an error feedback signal from a weighted sum of the current error (P), integration of past errors (I) and the rate with which the error has been changing (D). The controller can be tuned by adjusting the influence of these three measurements. Due to the simplifying assumption of linearization, PID controllers are very easy to implement, with few requirements to the micro controller, and are therefore well suited for low-cost domestic robots. Another class of reactive approaches are probability based algorithms such as particle filtering [233] or Markov Models [222]. For example, Webers et al. propose two motion control approaches to reach a noisy, shifting or stochastically moving target [250]. However, due to the fact that reactive approaches can only react once the error already has been made, such control approaches tend to result in additional control movements. A typical example is the overshooting behavior of PID controllers, by which they oscillate asymptotically around a target value. As outlined above, this kind of additional movements can cause negative interpretations by the user.

While reactive approaches only react when an error already has been made, proactive algorithms try to predict future states and minimize the error by adjusting the control signal beforehand. In particular, model predictive control approaches (MPC) construct an estimation of the result of a certain control command and compare the prediction with the actual execution result [251]. From this comparison an error signal is constructed that is used to update the control command. The update of the model can either be done on-line or off-line, for example by using machine learning techniques, such as neural networks [256] or evolutionary algorithms [94].

In [97] Dongbing et al. presents a path-tracking algorithm based on a multi-layer back-propagation neural network to model the non-linear kinematics of a mobile robot. The choice to employ a neural network is motivated by the ability to learn any continuous function, which is needed to adequately represent the non-linear drive kinematics of a mobile robot.
A common technique that can be applied in both, reactive and proactive approaches, to more accurately track a path but at the same time focus on smooth execution, is to explicitly take the physical properties of the hardware into account, for example using Lyapunov functions [2]. The problem of this approach is, however, that it requires more computational power and digital accuracy than usually available on an embedded micro controller. Nelson compares three further methods to achieve continuous steering functions for a mobile robot [174], namely (1) changing the steering mechanism, (2) changing the guided point and (3) changing the curves on the path. For our research the major disadvantage of changing any of these properties is that access to the internal micro controller is required to change the existing control algorithm.
The question that we want to answer is, whether we can improve the path tracking performance within the boundaries of the existing control algorithm where we have only control over the command parameters. In literature, updating the parameters is also referred to as calibration. Calibration techniques offer a simple solution to achieve an acceptable path tracking performance without requiring to change either the hardware, or the control software.

Calibration techniques

In general, calibration techniques can be classified in off-line or on-line techniques. Hybrid solutions start with an initial off-line calibration and adapt those initial parameters later on during execution as in [33].
Off-line calibration methods are characterized by special requirements for the calibration procedure, such as that the robot needs to follow a well defined, known path or a special measurement equipment to determine the robot's current location. For example, Borenstein and Feng present a practical calibration method to correct systematic odometry errors [25] called UMBmark (University of Michigan Benchmark) in which they let the robot drive on a known square path. They also showed that the shape of the path cannot be chosen arbitrarily, because errors in wheelbase and difference in diameter of the wheels can compensate each other. The method received some critique, for example in [62], because UMBmark assumes a smooth

surface for calibration and only takes the end-point locations of a test path into account. Doh et al. proposed a more accurate calibration method, called PC-method (POSTECH CMU-method), which uses the whole path for calibration [55]. In their study, they analyzed six sources of systematic error, namely (1) wheel misalignment (2) offset distance between center of rotation and wheel (3) offset distance between the nominal center of wheels and the center of mass induced by uneven mass distribution (4) different wheel radius (5) different forces on each wheel because of uneven mass distribution (6) position shift during rotation. They found that misalignment of the wheels is the major source of error, while the others are negligible. In their experiments they used a low speed to minimize the influence of non-systematic errors such as slip and they ignored numerical errors.

On-line calibration, on the other hand, allows to update the parameters of the robot controller during execution. For example, one track of research aims to improve the accuracy of odometry data by sensor fusion techniques as extension to the sole measurement of the wheels. Martinelli and al. propose a technique to simultaneously localize and calibrate the motion model, using Kalman filters and sensor fusion with absolute position estimation with a laser range finder [157]. Meng and Bischoff offer an odometry calibration for a custom built robot platform with two steerable drive wheels and a method for slippage detection which is used to trigger an update by a global localization procedure [166]. Wei and Fan present a model predictive control algorithm for an articulated robot in which they apply a neural network as the predictive control model [251]. The training set is obtained by measuring joint variables and torques. The authors criticize that the non-linear coupled and time varying dynamics are often ignored, or have been treated as disturbances and train instead the non-linear relationships with a neural network. In their experimental setup, the systematic errors mainly result from unknown link properties, loads or unknown torque constants and the random errors mainly result from non-linear friction, high frequency modes and time delays. Their results show that the neural neural network approach is an effective method for holonomic constraints. Xu et al. confirm this result and calibrate a robotic joint module with two degrees of rotary freedom for which they use a feed forward neural network to predict errors in joint angle [254]. A second network is trained with the residual errors of the first network to further increase accuracy, but also increases computational complexity by a factor of two.

In general, the effectiveness of the calibration methods depends to a large degree on the accurateness of the chosen motion model that is subject of the calibration. The models that better represent the non-linear kinematics of the mobile robot yield better results than those who ignore the influence of non-linear relationships, but are usually mathematically more complex and computationally more expensive. From our comparison we conclude that neural network based approaches are especially promising, because due to their implicit ability to learn also non-linear continuous functions they require less modeling effort of the motion dynamics. Additionally, neural networks have the ability to update the calibration on-line, which is in particular useful for low-cost hardware that may change drive behavior significantly during long term usage due to wearing out effects.
Even though the presented odometry calibration techniques cannot claim to make absolute localization unnecessary [166], they are able to improve local path tracking capabilities sufficiently to follow short designed paths in a behavioral control approach. In the following, we will introduce our approach to expressive path generation and present a simple neural network based on-line calibration technique that we applied to a commercially available differential drive robot.

7.4.2 Hardware setup

In our setup we took Roomba as an example of an available low-cost consumer product. Roomba is a behavior based robot that also provides an open interface (ROI) [111] to steer it with serial commands. This possibility of external control makes it equally attractive for hobbyists and researchers. On several websites and web forums, robot enthusiasts share and discuss several projects for Roomba and developed several little applications, also besides its original vacuum cleaning task [138]. Furthermore, it is an attractive research platform for human-robot interaction research, not only due to its low price in comparison to other research platforms, but also due to its robustness as a consumer product and the fact the Roomba is directly associated with a task, which allows research in real world settings instead of laboratory conditions [74].
Roomba has a differential drive system, consisting out of two velocity controlled wheels with a radius of $r_w = 33mm$, which are mounted with a base distance of $b = 258mm$ (see Fig. 7.11). A configuration of a differential drive robot is given by the triple (x, y, α) where x and y determine the position of the robot, measured from the center between the coaxial wheels and α is current orientation. From its geometry it follows that the forward

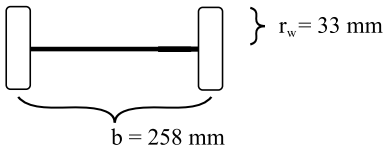


Figure 7.11: Roomba differential drive geometry.

velocity v and angular velocity ω can be calculated from the left (v_l) and right (v_r) wheel speeds:

$$\begin{aligned} v &= \frac{1}{2}(v_l + v_r) \\ \omega &= \frac{1}{b}(v_r - v_l) \end{aligned} \tag{7.1}$$

The according kinematic description of this drive setup is given by:

$$\begin{aligned} \dot{x} &= v \cos\alpha \\ \dot{y} &= v \sin\alpha \\ \dot{\alpha} &= \omega \end{aligned} \tag{7.2}$$

However, the open interface of Roomba does not allow to directly control the wheel speeds. Instead, it provides a higher level interface consisting of the two parameters radius r and velocity v, which are calculated according to:

$$\begin{aligned} r &= b\frac{v_l + v_r}{2(v_r - v_l)} \\ v &= \frac{v_l + v_r}{2} \end{aligned} \tag{7.3}$$

The two parameters are illustrated in Fig. 7.12. Using this setup, driving straight would mean to have an infinite radius. Therefore, the controller distinguishes between three different drive cases, depending on the value for radius: (1) driving a curve, (2) driving a straight line (3) rotating on the spot. For the first drive case, Roomba drives on a circle with radius r and velocity v. A positive radius and a positive velocity will make Roomba turn left. A combination of negative radius and positive velocity will make Roomba turn right. If negative velocities are specified, Roomba will drive backwards. According to the documentation of the command interface, the value for the radius has to be in the range of $[-2000\text{mm}, 2000\text{mm}]$ and the velocity has to be in the range of $[-500\text{mm/s}, 500\text{mm/s}]$. The other two drive cases are identified by special values for the parameter of the radius. Roomba drives in a straight line, if a value of 32768 is specified and rotates

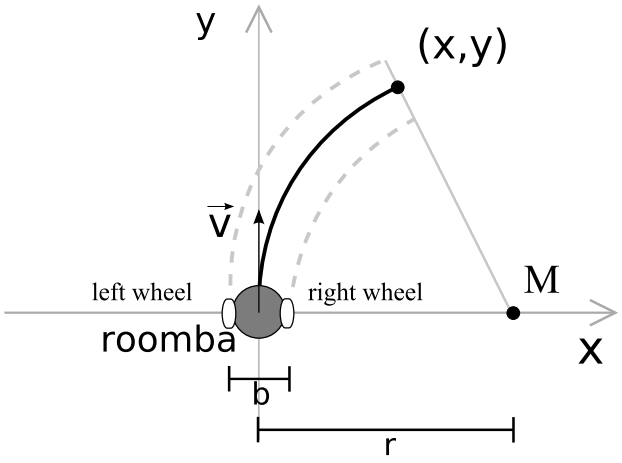


Figure 7.12: Control space for navigating Roomba®.

on the spot if either -1 (clockwise) or 1 (counter-clockwise) are passed to the drive command.
On the one hand, the high level command interface makes it easier to control the robot, because no dynamic equations need to be solved or integrated over time, but, on the other hand, makes it impossible to influence how these commands are executed by the controller. This is one of the drawbacks of using a closed system. For example, the parameter values of the drive command suggest an execution precision with an accuracy of 1 mm. However, translating these natural numbers to wheel velocities, requires some floating point operations. No specification about numerical quantization is given in the documentation. However, these are minor drawbacks in comparison to the effort to build or program a drive control from scratch. The power of using a closed system like Roomba is that the researcher does not have to spend time and effort on the hardware, but can directly start to program Roomba.

7.4.3 Calibration procedure

When we started animating Roomba with our animation editor, we quickly found that Roomba was considerably off the expected positions, which makes it unusable for controlled and systematic research, for example on the users' perception of the robot movements. Therefore we developed a calibration procedure that can be added on top of an existing drive system. With our calibration procedure, we aim to correct systematic errors made

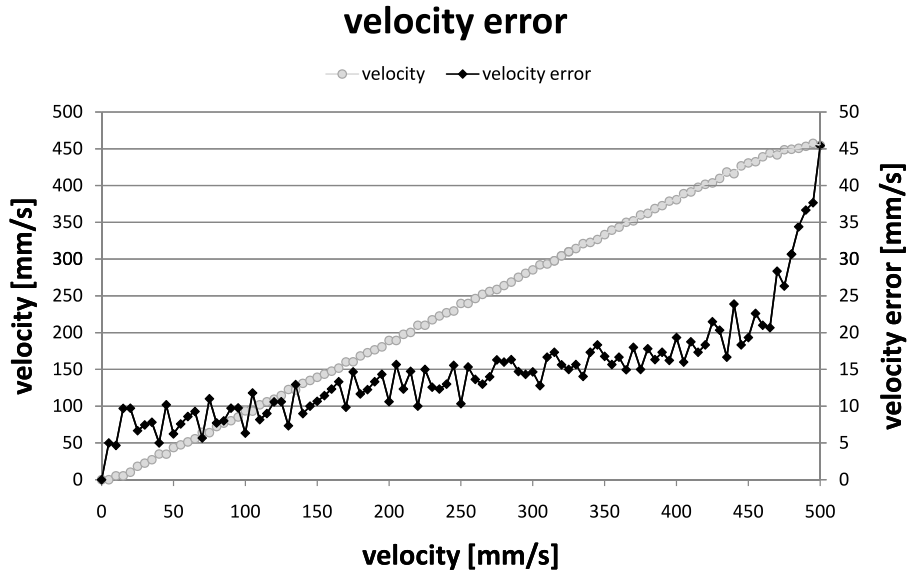


Figure 7.13: Inaccurate drive system. Left: Measurement of resulting velocity in relation to the given command. Right: Velocity error.

by the existing implementation of the micro controller. A typical example of such an error for Roomba is depicted in Fig 7.13. The graph shows the absolute velocity tracking performance (gray bullets) and the relative error (black diamonds) that is made for a fixed radius value. In an ideal case, the measurement of the actual velocity would be an identity function, but it can be seen that the slope is actually lower than one and flattens close to a velocity of 500. The error line depicts the difference between the command that is sent and the velocity that is executed. From this example it can be seen that the error could be corrected by using this knowledge and changing the execution algorithm accordingly. Usually, this kind of calibration would require to change the existing control algorithm or to implement a new one from scratch. However, in closed system such as Roomba, this is not an option, because the internal algorithm is not accessible without major modifications to the hardware. Even if it would be accessible, the question remains how much effort would be required on low level hardware implementations, instead of working on the research question of interest.

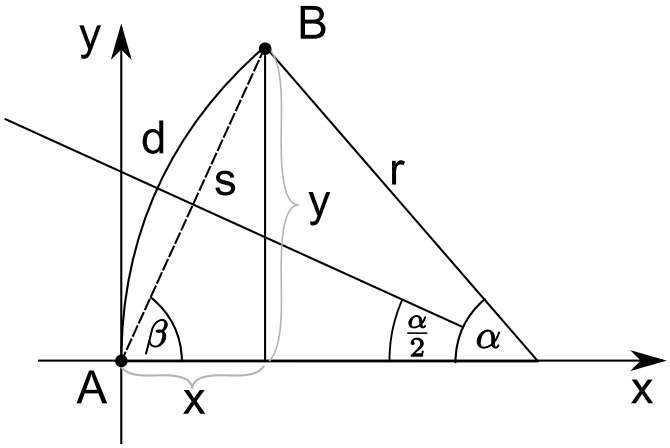


Figure 7.14: Geometrical interpretation of the path of the robot from position A to position B.

Reference system

A fundamental requirement for all calibration procedures is a reference system. This means we need to establish a relationship between the control commands and resulting physical configuration changes of the robot. The overall goal of the calibration procedure is to insure that the robot renders the designed path as specified. Following the documentation of the command interface for Roomba, this relationship is given by the entities for radius and velocity. In particular, the radius is given in millimeters and the velocity is given in millimeters per second. With this information, we are able to calculate geometrically the current position of the robot, relative to the starting position by integration of the circular path over time. A geometrical interpretation of the movement path is depicted in Fig. 7.14, in which the robot travels from position A to position B. Using this notation the deltas in the x-y plane can be found by using the travel distance d to calculate the travel angle α and subsequently the x and y offsets:

$$d = vt = r\alpha \tag{7.4}$$

$$x = s\cos\beta \qquad = r - r\cos\alpha \tag{7.5}$$

$$y = s\sin\beta \qquad = r\sin\alpha \tag{7.6}$$

This model also serves as common unit to communicate between our animation editor and the robot. As a result, the paths from the animation editor can be directly translated to real world coordinates. The real world

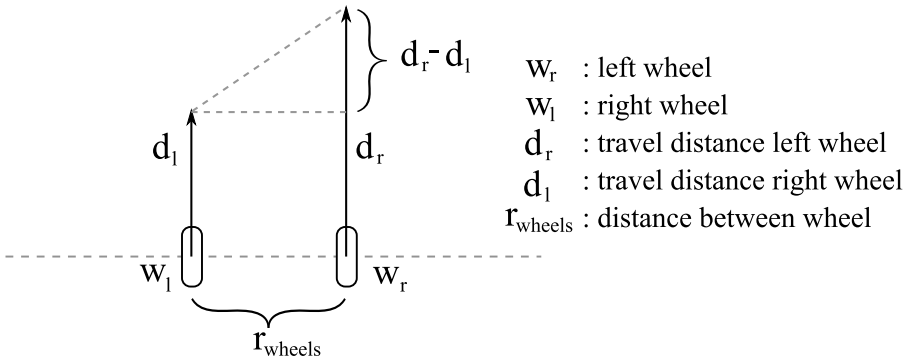


Figure 7.15: Odometry information of travel distance and turning angle are calculated from the travel distances of both wheels.

mapping also helps the designer to develop an idea of how the animation will look like (e.g., how much space is required). However, the only sensory information available to measure the actual travel distance are Roomba's odometry sensors. For this reason, we run a series of initial experiments to test the accuracy of current sensor readings.

Odometry validation

Roomba's odometry sensors consist of two optical sensors attached to the wheels that track the rotation angle of the wheel. The information can be accessed through the command interface which contains a special command to request the current sensor readings. The odometry sensors return the distance d Roomba has traveled and the angle α Roomba has turned between two successive inquiries of the sensor values. From the documentation it follows that the distance is calculated from the distances traveled by the left (d_l) and the right (d_r) wheel according two $d = (d_l + d_r)/2$. The setup is depicted if Fig. 7.15. However, instead of directly returning an angle in degrees or radians, the micro controller returns a difference $d_{\text{rot}} = (d_r - d_l)/2$. Given the distance between both wheels to $r_{\text{wheels}} = 258[\text{mm}]$ the inner radius is $r_l = r - \frac{258}{2}$ and the outer radius is $r_r = r + \frac{258}{2}$. Given these two

values the measurement d_{rot} can be converted to degrees α_{od} from odometry according to:

$$\begin{aligned} d_{\text{rot}} &= \frac{d_r - d_l}{2} \\ &= \frac{\left(r + \frac{258}{2}\right)\alpha_{\text{od}}\left(\frac{\pi}{180}\right) - \left(r - \frac{258}{2}\right)\alpha_{\text{od}}\left(\frac{\pi}{180}\right)}{2} \\ &= \frac{258\pi\alpha_{\text{od}}}{360} \\ \alpha_{\text{od}} &= \frac{360 d_{\text{rot}}}{258\pi} \end{aligned} \tag{7.7}$$

In the following experiments, we tested the accuracy of Roomba using our graphical animation tool (see Chapter 6.4). We validated empirically how much the values of the odometry sensors differ from the real world position, by physically measuring the distance the robot has traveled with a meter. In first trial sessions we found that not the whole range for the velocity is used but that the robot reaches its maximum velocity already at about velocity value of 450 mm/s. It also only started to move above a threshold of about 15 mm/s. We took these initial boundaries into account and constructed an experiment to sample the command parameter space and evaluate the actual result of these control commands. We run this experiment on even floor on which the rubber wheels of Roomba had good grip to minimize random errors and measured the distance Roomba traveled with a meter. We sampled the command space used to steer Roomba at [1800, 1500, 1000, 500, -500, -1000, -1500, -1800] for the radius and [400, 200, 100, 50, -100, -200, -400] for the velocity, resulting in $N = 56$ samples, and measured its real world position after the command execution. A sample pair therefore contained the output of the odometry for travel distance and turning angle, and the measured xy-position and turning angle of robot after the run. For comparison, we converted the odometry values in xy-coordinates according to the following formulas as presented in equations 7.5 and 7.6. After that, we calculated an average error for both, position and turning angle according to:

$$\begin{aligned} err_{\text{pos}} &= \frac{1}{N}\sum \sqrt{(x_{\text{od}} - x_{\text{m}})^2 + (y_{\text{od}} - y_{\text{m}})^2} \\ err_{\text{angle}} &= \frac{1}{N}\sum |\alpha_{\text{od}} - \alpha_{\text{m}}\| \end{aligned} \tag{7.8}$$

As result, we got an average error for the position of 20 mm and an average error of 1 ° for the turning angle, which we considered as sufficiently

accurate to use the odometry information as initial reference system for the calibration. Of course, odometry sensor information can later be exchanged by sensor readings from more accurate position sensors such as laser based velocity sensors or external localization system, as soon as they become cheaply available. Additionally, in literature several odometry calibration methods have been proposed [25, 55, 246]. In the next step we evaluated path tracking performance in relation to the given command.

Command validation

After validating the odometry readings, we compared how these values relate to the commands that were sent to Roomba. The drive command suggests a high accuracy by requiring the commands to be given in mm. However, already in the more or less ideal scenario of the above experiment, we found an average error of 40 mm for the position with respect to a theoretically calculated position based on the velocity and radius given in a command. The average error for the turning angle yielded a value of 3 °. We want to point out that these errors occur after the execution of a single command. Especially the error in the turning angle accumulates quickly in consecutive drive commands, resulting in unpredictable configurations after only a few commands. Admittedly, in Roomba's original functionality as a random walker, a higher precision for movements was not required. However, it is interesting to note that Roomba already has sensors on board that theoretically allow for more accurate navigation. The question that immediately arises is how this information can be utilized to calibrate the drive commands.

In order to answer this question, we had at first a closer look at how the given commands relate to errors that are being made in comparison to the expected configuration. Similar to the first experiment, we recorded a sample table using values in the range of -1900 and 1900 in steps of 200 for the radius and values in the range of 60 to 460 in steps of 50 for the velocity. We concentrated only on forward motion, because we considered forward motion to be the average use case and because we experienced a symmetric behavior between backward and forward motion. We recorded the command values and the resulting odometry readings for the 200 sam-

ples and calculated which command would have caused the readings in an ideal controller according to:

$$\begin{aligned} r_{\text{ideal}} &= \frac{dist_{\text{ideal od}} \cdot 180}{\alpha_{\text{ideal od}} \cdot \pi} \\ v_{\text{ideal}} &= \frac{dist_{\text{ideal od}}}{t_{\text{ideal od}}} \end{aligned} \tag{7.9}$$

The results suggest a non-linear relationship between the absolute values of the commands and the accuracy with which they are executed by the micro controller of Roomba. The averages of the absolute errors are depicted in Fig. 7.16. The graphs A and B depict the average velocity error with respect to the values of radius and velocity of the command. Similarly, the graphs C and D depict the average radius error with respect to the issued command. For the velocity the graphs suggest that the smaller the radius error is, the higher the resulting velocity error will be and that the inaccuracy of the velocity grows exponentially with the issued command. Accordingly, it can be concluded that the radius error grows with an increasing radius, but that the velocity has almost no impact on the radius inaccuracy. The last spike in the graph D can be explained by the hardware limitations of the wheel motors that reach their maximum speed, because in order to drive a curve with a specific radius, one wheel has to turn faster than the other. If, however, the wheels turn already with maximum velocity, the required difference in velocity to turn with a specific radius cannot be achieved and the robot will almost drive straight.

Calibration

Our approach to calibrate the drive command of Roomba is to feed back the information from earlier runs in order to update a predictive control model that calibrates the commands in a way that they match the expected drive behavior. The setup is depicted in Fig. 7.17. A path generator translates an input path to control commands. These are calibrated using a neural calibration method before fed to an embedded controller. After each command execution the resulting error is fed back to update the calibration. This calibration architecture can be added on top of existing embedded hardware controllers. García-Córdova et al. motivate a similar approach biologically from the observation that also animals have to learn the consequences for their own actions in order to be able to deal with unknown environments [79]. In [118], Jang et al. calibrate an industrial robot by

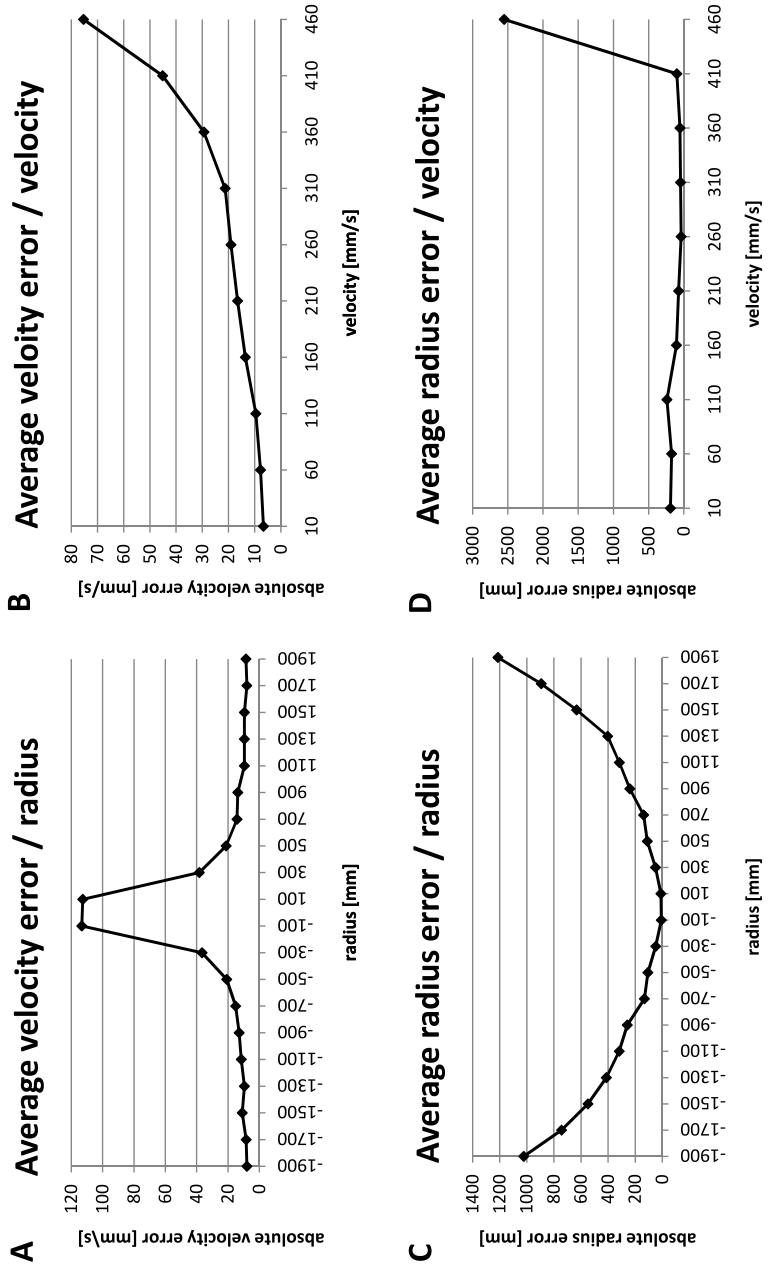


Figure 7.16: Roomba robot command validation

dividing the workspace in several regions and used a positioning sensor system to measure the robot position. The error parameters for every region were used to train a radial basis function neural network. They apply this method to a six degree of freedom industrial welding robot. We follow a similar approach by learning a predictive motion model based on past experiences.
The approach can mathematically be modeled as finding the inverse function to the micro controller and the robot. We chose as reference system the geometrical travel distance, so that the micro controller and physical drive behavior corresponds to a two-dimensional function M that receives a tuple of velocity and radius from the command (cmd) and returns a tuple describing the drive behavior of the robot (measured by the odometry (od)):

$$(v_{\text{od}}, r_{\text{od}}) = M(v_{\text{cmd}}, r_{\text{cmd}}) \tag{7.10}$$

In an ideal situation, the micro controller resembles the identity function, meaning that the command is executed as specified. In order to re-establish an identity function, we are searching for an inverse M^{-1} of the function M to accommodate for the transformation:

$$(v_{\text{ideal}}, r_{\text{ideal}}) = M^{-1}(M(v_{\text{cmd}}, r_{\text{cmd}})) \tag{7.11}$$

The first thing that we verified for pursuing this approach is that the response of the controller is sufficiently deterministic to train a model. We randomly picked 100 different control tuples consisting of values for radius and velocity and repeated each command 10 times on the robot. We then calculated an overall standard deviation of the resulting positioning and received a value of $SD = 2.68mm$, which we considered to be sufficiently small to meaningfully train a model. Of course there will be always a small margin for example due to random or rounding errors.
Given the non-linear nature of the dependency between the issued command and the resulting driving behavior, we found a neural network approach to be the most flexible solution. It has been shown that any continuous function can be approximated by superpositions of sigmoidal function [43]. Applying this principle in a Multilayer Perceptron (MLP), it has been proven that one hidden layer is sufficient to approximate a function, given an appropriate number of nodes [77]. Furthermore, a MLP can easily be updated using a backpropagation algorithm as soon as new data samples become available. Ozaki et al. applied a similar approach to calibrate a robotic manipulator [180]. Fierro and Lewis used a neural network based torque controller for a car-like mobile robot and tested their method for

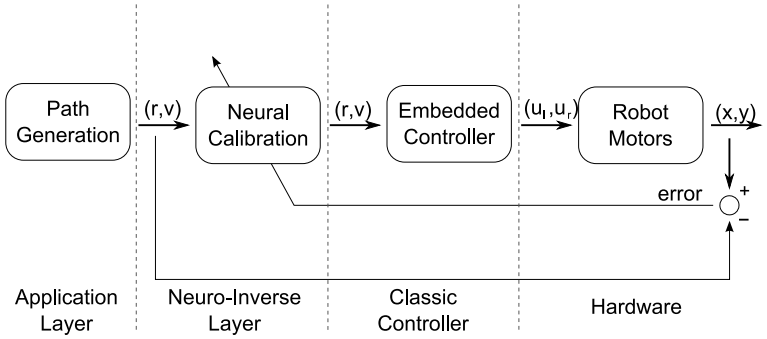


Figure 7.17: Calibration setup: A path generator generates control command commands with radius r and velocity v. An embedded controller translates these commands to wheel speeds u_l and u_r which result in new x,y configurations of the robot.

trajectory tracking [71]. However, they focus on high precision control, by also taking the kinematics into account, which is computationally more costly than taking a geometrical approach that we pursuit with Roomba.

In an initial experiment, we implemented a MLP with an input layer consisting out of two inputs, one hidden layer H with 16 neurons and one output layer with two output neurons. At this point, the number of neurons was chosen arbitrarily. We connected the input neurons directly to the odometry readings for radius and velocity, both scaled to a range between -1 and 1. The activation of a hidden neuron y_i was determined by a weighed sum over the inputs and scaled by the sigmoidal Fermi-function Θ with bias b to compute the output of neuron i in layer λ:

$$\begin{aligned} y_i &= \Theta\left(\sum_{j=1}^{\dim\lambda-1} w_{i,j}^{\lambda} y_j^{\lambda-1}, b\right) \\ \Theta(x,b) &= \frac{1}{1+e^{-(x-b)}} \end{aligned} \tag{7.12}$$

The output of the output layer was computed analogously, but using a linear activation function $\Theta(x,b) = x - b$. Additionally, both input and hidden layer were extended with a bias neuron whose output is always 1. This means in total $n_w = 5 \dim H + 2$ weights and $n_b = dimH + 2$ activation function bias had to be estimated. We used the backpropagation algorithm

[206] using a square error measure between the output of the network y and the given command sample s:

$$E = \frac{1}{2}\sum_i (\vec{y}_i - \vec{s}_i)^2 \tag{7.13}$$

The functional animation approach directly supports this feedback mechanism also on-line, because it naturally creates a training sample through its interval based approach. A sample consists of the command values that are sent at the beginning of an interval and the configuration that is measured at the end of an interval.

We tested the qualitative learning capability of the neural network by trying to approximate the sample that we took for the command validation. The training set therefore contained 200 training samples from which we chose for every training cycle one sample at random and updated the parameters according the the backpropagation algorithm. We found that the network was able to approximate the qualitative shape of the function, but we observed that especially the control values around zero and at the borders of the control intervals raised the output of the functions too high. The reason for this is that, in the MLP approach, every neuron contributes to the overall output of a specific value of the input space.

Therefore, before evaluating the performance of the neural network approach, we chose to implement a second neuronal network architecture based on local influence of the neurons. For comparison, we found Radial Basis Functions (RBF) a suitable candidate due to its similarity to the MLP approach [182]. Comparable to MLP, RBF consist out of one hidden layer of neurons that receive a weighted sum from the input layer. The crucial difference is that RBF uses locally restricted activation functions instead of sigmoid activation functions. This architecture has the advantage that every neuron is responsible for only a specific interval of the input space. This also makes it more fault tolerant, because the overall network response is preserved, even if a single neuron produces erroneous outputs. The disadvantage is that more neurons are needed to sufficiently sample the input space. This becomes intractable for high dimensional input spaces, but is well possible for our problem domain with well defined and limited two dimensional input space.

The learning step of a RBF network is performed in two steps. In the first step, appropriate centers and the radius of influence for the neurons are calculated to approximate the input space. The output of a neuron n_k is

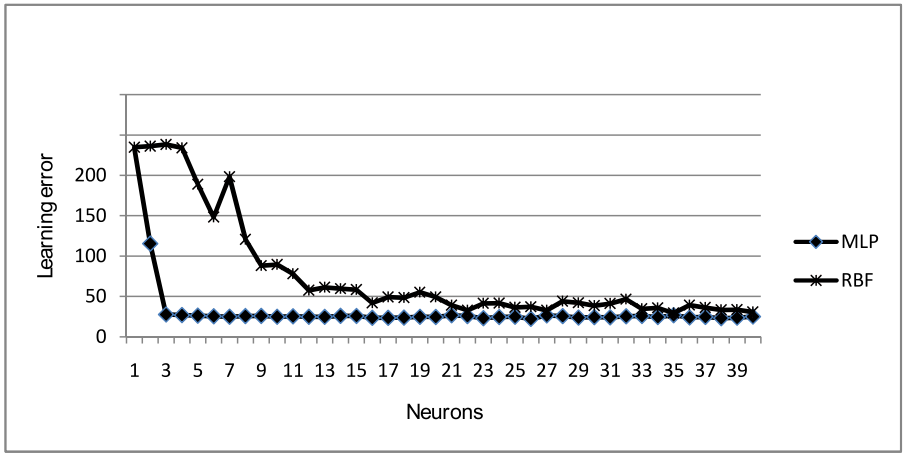


Figure 7.18: Uncalibrated error

calculated according to the following function, where $\vec{w_k}$ is the center of the neuron in input space and r_k determines the radius of influence:

$$n_k = e^{-\frac{1}{2}\left(\frac{\|\vec{x}-\vec{w_k}\|}{2r_k}\right)^2} \tag{7.14}$$

We used vector quantization (VQ) to calculate appropriate centers for a given set of input samples and set the radius to be the distance to the closest neighbor. The overall network response for an output dimension i is then calculated as a weighted sum of all neurons with an output weight a_{ik}

$$y_i = \sum_k a_{ik} \cdot n_k \tag{7.15}$$

The parameters for the weights a are calculated using a gradient decent method with square error measure analogous to the MLP architecture in equation 7.13. In the following we took both implementations and evaluated for both architectures their calibration performance.

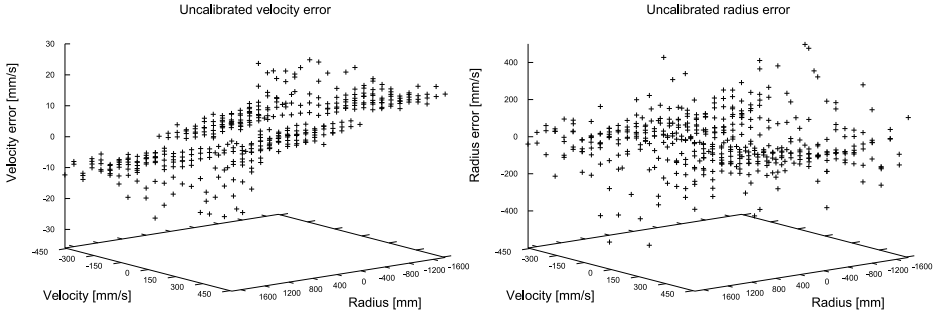
7.4.4 Evaluation

For the evaluation of the calibration, we sampled the whole range of the command space by steps of 50 for velocity and by steps of 150 for radius, resulting in 480 samples. We used this training set to learn an initial calibration for both networks.

In the first step to evaluate the calibration performance of both networks, we first searched for a suitable number of neurons. With too few neurons the functions cannot be learned correctly, but with too many neurons the computational complexity becomes impracticable for low cost platforms and additionally poses the risk of over fitting. We therefore trained both networks with the training set for an increasing number of neurons. For each training we used a fixed number of 10.000 training cycles and calculated the average approximation error of the network. The results for approximating velocity are depicted in Fig. 7.18. It can be seen that the error of the MLP quickly drops, indicating a linear component as a major factor in the error. As expected, the RBF network only approaches the performance of the MLP later, as soon as enough neurons are available to sample the input space. From the results and for ease of implementation, we used a number of 24 hidden neurons for both networks for the remainder of our evaluation.

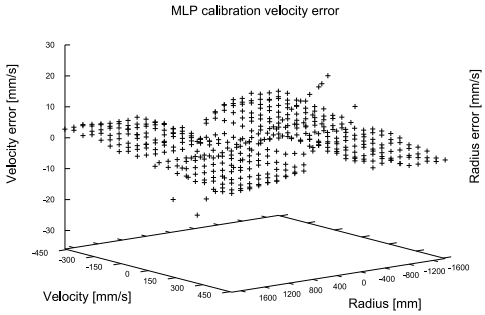
We trained both networks with the above training set until the approximation error saturated. For the MLP this resulted in 1000 training cycles and for the RBF in 4000 cycles, respectively. We then used the networks to transform the steering commands before they were sent to the controller and collected the error for velocity and radius. Fig. 7.19 shows the performance of the calibration procedures. The plot of the velocity error in the uncalibrated condition unveils the linear component that we already suspected earlier, when we determined the number of neurons for the networks. That means the higher the velocity, the greater the error the robot makes. The plot for the radius error, however, has a more complex wave like structure that increases at the extremes. It can also be seen that the error around zero velocity is exceptionally high due to the fact that the motors only start turning above a certain threshold.

From the results of the MLP calibration it can be seen that the MLP was able to qualitatively train an inverse function to the micro controller that restores the desired identity control function. However, the graph shows a high error around zero for the radius. This can be explained with the hardware limitations of the motors. The error increases with growing velocity, indicating that the necessary difference for the wheel speeds cannot be met any more. A small value for the radius means that the robot drives on a very small circle with a high difference between the wheel speeds. Another fact that can be observed is that the major plain is not located at zero but shifted upwards. As indicated above, this can be explained by the fact that the extreme values at the borders of the command space have

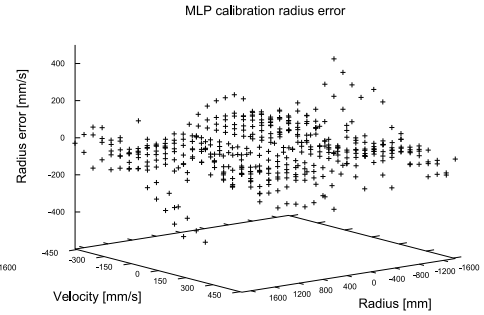


(a) Uncalibrated error velocity

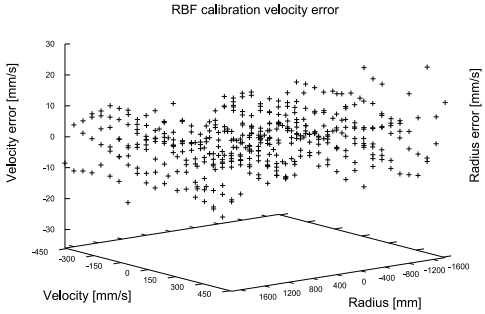
(b) Uncalibrated error radius



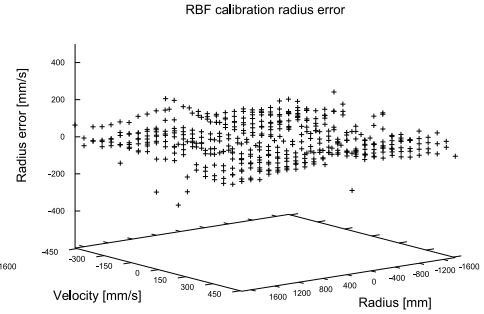
(c) MLP error velocity



(d) MLP error radius



(e) RBF error velocity



(f) RBF error radius

Figure 7.19: Calibration performance

an impact on all neurons, pulling the answer of the network too far in one direction. Nevertheless, the network improves the overall path tracking performance. While the qualitative structure of the velocity calibration function can be learned reasonably well, the network has more difficulties in approximating the function for the radius. Despite the calibration, the error plain still shows the wave like structure that should be canceled out to zero. This also can be explained by the structure of the network. In the training phase, every sample pulls the response of the network in the direction to minimize the error, with the result that opposite error corrections in the structure cancel each other out.

The RBF calibration, in contrast to the MLP calibration, performs better at the calibration of radius. It appears that the wave like structure can efficiently be approximated by superpositions of the chosen basis functions. The linear component of the velocity error, on the other hand, poses a bigger challenge with the chosen number of neurons and values for the radii. If the control space is not sampled sufficiently, the response of the network is artificially dragged to zero in underrepresented areas, because no neuron has a sufficiently large influence. Increasing the radius of influence would help in this case, but pose a problem if more fine grained features have to be learned. Another option is to increase the number of neurons. Reconsidering the graph on the average training error with respect to the number of neurons, it can be seen that graph still shows a downwards trend for more than 40 neurons. The disadvantage is that the computational complexity increases with a growing number of neurons.

At last we performed a statistical analysis on the collected data to verify the significance of the results. Therefore, we performed an analysis of variance (ANOVA) in which the calibration was the independent factor and the absolute errors in velocity and radius were the dependent variables. The calibration factor consisted of three conditions: no calibration, RBF calibration and MLP calibration. Table 7.1 shows the mean and standard deviation for all three conditions. As expected from the graphs, we found that the calibration method had a significant influence on both errors, velocity ($F(2,1405) = 22.05$, $p<0.01$) and radius ($F(2,1358) = 7.20$, $p < 0.01$). Post-hoc t-tests with Bonferroni corrected alpha showed that the error was significantly ($p < 0.03$) lower in the MLP condition ($M = 13.07$ $SD = 24.24$) for velocity compared to the uncalibrated condition ($M = 19.47$, $SD = 29.70$), but significantly higher ($p < 0.01$) for radius ($M = 151.88$, $SD = 173.12$) compared to the uncalibrated condition ($M = 116.01$, $SD = 229.60$). The RBF calibration showed exactly the opposite behavior. The

Condition	velocity		radius	
	mean	SD	mean	SD
uncalibrated	19.47	29.70	116.01	229.60
MLP	13.07	24.24	151.88	173.12
RBF	29.33	52.78	85.34	122.38

Table 7.1: Mean and standard deviation of the error for the uncalibrated, MLP and RBF condition

post-hoc t-tests with Bonferroni corrected alpha showed that the velocity was significantly ($p < 0.01$) higher in the RBF condition ($M = 29.33$, $SD = 52.78$) but significantly lower ($p < 0.03$) for the radius ($M = 85.34$, $SD = 122.38$). From this analysis it can be seen that the MLP calibration improved the path tracking performance for velocity by 33% and that the RBF calibration improved the path tracking performance for radius by 24%. However, even though there seems to be more structure in the error plots, the overall performance worsened with calibration for the radius with the MLP calibration and the velocity in the RBF calibration. These behaviors can be explained as outlined above with the characteristics of both network architectures. Therefore, the network architectures have to be chosen with care to achieve a desired improvement in the path tracking behavior.

The results show that we were able to achieve a significant improvement in the path tracking behavior, if we apply an MLP to calibrate the velocity and an RBF for the radius. The question remains, why our calibration procedure was not able to restore the desired identity function completely. One of our constraints was to develop a calibration procedure, without modifying the existing hard or software. Therefore, the achievable improvements in the path tracking performance are naturally limited by the capabilities of the existing hard and software. In our example with Roomba we found that rounding errors can become a major obstacle, especially if they cause that multiple input values are mapped to the same output values. This behavior is visualized in Fig 7.20. We performed a complete sampling of the command space in which we varied the radius parameter continuously from 100 to 1800 and kept the velocity constant. The quantization to integer values results in the typical stair shape of the graph. That means that especially in the areas of a high radius, our calibration approach fails, because sending a different input value will have no effect, or even worsens

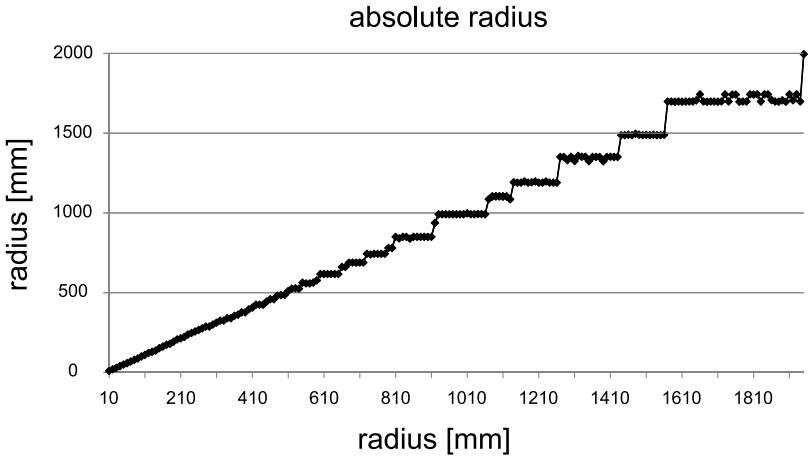


Figure 7.20: Full sampling of radius between 100 and 1800 with fixed velocity of 400.

because an input command is calibrated to a value out of range for the control command.

7.4.5 Discussion of calibration procedure

With our study we aimed to develop a calibration procedure to increase local path tracking accuracy for low-cost consumer mobile robots. We motivated the necessity for consumer robots to be able to show expressive motion paths, and derived from the application domain three constraints for our research (1) not to violate the cost-constraints (2) applicability to closed systems and (3) improving local path tracking performance with a focus on how the robot is perceived. Based on these constraints, we followed a model predictive control approach and applied and compared two different neural network architectures to calibrate the control commands of the robotic vacuum cleaner Roomba. The chosen approach satisfies all three constraints. First of all, it does not violate the cost-constraints, for example by adding new hardware, or using numerical methods that are impracticable for embedded systems. Secondly, we successfully demonstrated that we are able to apply the calibration method to an existing, closed system domestic robot and third, by applying a model predictive approach we avoided to introduce additional correction movements that could have an undesired effect on the perception of the robot.

Furthermore, we have shown that the general approach of building on top of a closed system by using a neural network calibration approach improves the path tracking accuracy. Even though the expected results, that a neural network calibration will always improve the tracking performance could not be confirmed, we have demonstrated that by carefully choosing the network architecture an improvement can be achieved. From our comparison of a MLP and a RBF architecture for calibrating Roomba, we concluded that the best performance can be achieved by applying a MLP to calibrate the velocity parameter and a RBF architecture to calibrate the radius parameter. Both networks perform with a sufficiently small amount of neurons to be implementable on an embedded hardware. The obtained results could theoretically be improved further, for example by adding more domain knowledge in terms of choosing basis functions that best represent the qualitative structure of a calibration function, but naturally this approach conflicts with a general applicability, also to other robotic platforms, of the calibration algorithm. Another advantage of our approach is that we did not assume any specific drive kinematics of the robot, but let the network automatically learn the non-linear relationship between control signals and resulting change in the robot configuration.

However, we also have shown that there are limitations to this approach, especially due to the constraint of working with a closed system. The calibration method cannot compensate for errors that are made due to internal rounding errors. Nevertheless, we believe that a post-hoc calibration is useful especially for the design of strictly modularized systems. Furthermore, our calibration method is flexible to on-line update the calibration, for example if the low-cost servos wear out over time, or to accommodate and predict changing terrains.

7.5 Discussion of SRD architecture

The presented SRD framework identifies and addresses central tasks in the process of creating an application for an expressive robotic interface. While in the gaming industry a similar multidisciplinary approach has become a standard [113], it is still new for creating robotic applications.

For the modeling of the SRD framework, several components of existing programming environments for robots have been adapted to generate a unified architecture. For example, a particular strength of the Microsoft Robotic Studio® is its flexibility due to its service orientated middleware.

One of the advantages is that the robot hardware can for example be controlled using high level web interfaces. However, an asynchronous component model can also result in non-deterministic system behavior, which complicates the design process. It has been motivated that during application design a separate environment with a synchronous communication model benefits the design process. Also the Lego® NXT development environment makes a clear separation between development and run-time environment. The designer has first to finalize a complete application that in turn can be compiled and uploaded to the hardware. The advantage of this setup is that development and execution are decoupled from each other, but it makes it very difficult to debug certain aspects of the application. The final behavior can only be assessed during execution of the application. The SRD framework overcomes this shortcoming by generalizing over software components that relate to an application artifact.

Furthermore, experiences with the OPPR framework served as an important input for the design of the SRD architecture. For example, animation technology is one of the key technologies for creating expressive robotic interfaces. The OPRR framework applied keyframe animation to the development of robotic behavior. Within the SRD framework, the principle of keyframe animations has been extended by the functional animation principle. Functional animations combine the control over the expressive behavior with the high level control of scripted behaviors.

Another central component of OPPR is the animation module that controls the robotic hardware. The concept of the animation module has been generalized by an artifact executor, which is decoupled from a view of on the application. Within the OPPR framework, visualization, development and execution are coupled, which make it difficult to extend the framework by new types of editors or views. For example, next to a 3D simulation of the posture another conceivable view might display the emotion that is currently expressed by the character.

The SRD architecture introduced a general component interface, which allows third parties to extend the framework by new components. A synchronous communication model has been proposed which uses a blackboard system to exchange information. A scripting engine was used to allow computational hierarchies, which are also used by the functional animations. An event based communication protocol has been defined, which allows third party component developers to provide new components for the SRD framework. Based on this component model, several software components such as the general editor interface and the view facilities have been devel-

oped. The editor interface generalized the concept of an editor and defines a general editor interface to insure coherent editing functionality (e.g., cut-copy-paste).
Finally, the reemerging object definition design pattern has been reused on several levels. For example, it has been used to provide meta information about the individual components and to unify the interface to control and compare different embodiments.

7.6 Summary

This chapter has introduced the Execution-Environment of the SRD architecture. It complements the Development-Environment and with components that are required for the execution of the application on the deployment hardware.
The Execution-Environment shares with the Development-Environment software components that are related to application artifacts, which ensures consistent rendering of the application between the two environments. The presented architecture minimizes maintaining effort for the software by efficiently reusing software components form the Development-Environment. Furthermore, the component structure of the Development-Environment has been generalized and extended by a component structure with an asynchronous communication model. It has been motivated that an asynchronous communication model is required to satisfy the demands of typical robotic hardware. The components have been modeled based on a service oriented reference architecture. A central dispatcher manages components, but data communication has been decentralized to avoid bottlenecks. A publish-subscribe mechanism has been developed with a type based selection mechanism to query and filter data.
With this setup, the Execution-Environment explicitly targets at the specific demands of the deployment environment. In particular, it addresses the management of sensor input and proposes a calibration method to improve the drive accuracy of rendering animations on low-cost platforms. For handling of sensor data, a situation awareness component is introduced, which generalizes sensor input of a robotic embodiment and provides a high level interfaces to react to external stimuli. It hides the details of signal processing from the designer by automatically constructing a network of recognition modules and providing the recognition results to an application. Also the proposed calibration procedure hides hardware details from the designer. For example, it assumes that the designer has no access to

the internal controller of the robot, but requires that the hardware platform can reliably be controlled. The calibration procedure has been used to improve drive accuracy of a Roomba robot for the case study presented in Chapter 9.

Part III

Evaluation and application

Chapter 8

Evaluation of the architecture

The main goal for the evaluation of a given software architecture is to predict quality attributes before a software is implemented [36]. Already in 1972, Parnas proposed modularization and high level design descriptions as a method for improving software quality, in particular in terms of flexibility and comprehensibility [183]. Starting from those initial observations, the notion of a software architecture has emerged in the field of software engineering to provide an appropriate balance between abstraction and concrete design to be evaluated for quality attributes [16].
Software architectures can be evaluated in terms of functional and non-functional attributes. In the next section first an overview of commonly evaluated software architecture attributes is given before these are applied for the SRD architecture.

8.1 Architecture quality attributes

Several software engineering communities have proposed their own software design methods for the development of a specific type of system, including reusable systems [126], real-time systems [149] and high-performance systems [226]. However, these methods only focus on a single attribute of the architecture for a particular application domain. The challenge is that it is not possible to directly measure quality attributes based on an architecture, because this would imply that implementation is a deterministic mapping of the architecture [36]. Instead, architecture evaluation methods

focus on evaluating the *potential* of a particular architecture to meet software quality attributes. This is of great importance, especially in business environments, because the outcome of the evaluation methods can be used to cost effectively refine a system before it is implemented.
A software architecture has a number of characteristics that determine the quality of the software, but are not directly related to the functionality of the software. A software quality attribute is therefore a non-functional characteristic. Most commonly a non-functional quality attribute cannot directly be linked to a single software component, but is an emerging feature from the overall architecture and collaboration of components. A definition for such a software quality metric is given by the IEEE standard 1061 [109], which describes a methodology to establish and analyse software qualities. In a different standard ISO/IEC9126 [115] a list of six concrete characteristics, including sub-characteristics is given. These qualities are listed in Table 8.1. Sommerville restricts the list to four essential categories, which are described in table 8.2. However, the definitions of software qualities are not independent dimensions. Several definitions of strongly related constructs exist. For example Dobrica and Niemelä compare definitions of 'maintainability', 'modifiability' and 'flexibility' and conclude that even though the wording is different the semantics are similar [54].

8.1.1 Software architecture evaluation

In the literature several architecture evaluation methods have been proposed. A survey of most commonly applied methods is given in the surveys [54] and [203].
Dobrica and Niemelä defined a framework to compare software architecture evaluation methods. They compared the goals of the evaluation method, the employed techniques, which quality attributes are evaluated, stakeholder involvement, performed activities, reuse of existing knowledge and method validation. With this framework they evaluated the scenario-based architecture analysis method (SAAM), and three of its derivatives (SAAMCS, ESAAMI, SAAMER), the architecture trade-off analysis method (ATAM), scenario-based architecture engineering (SBAR), architecture level prediction of software maintenance (ALPSM) and the software architecture evaluation model (SEAM). An in depth discussion of ATAM, ARID and SAAM is given in [36].
Most notably, the various evaluation methods differ in terms of which quality attributes are evaluated. While for example, SAAM and derivatives mainly focus on a single quality attribute, ATAM and SBAR are able

Functionality	suitability accuracy interoperability security functional compliance
Reliability	maturity fault tolerance recoverability reliability compliance
Usability	understandability learnability operability attractiveness usability compliance
Efficiency	time behavior resource utilization efficiency compliance
Maintainability	analysability changeability stability testability maintainability compliance
Portability	adaptability installability co-existence replaceability portability compliance

Table 8.1: Software quality characteristics according to standard ISO/IEC 9226-1

Maintainability	Software should be written in such a way that it may evolve to meet the changing needs of customers. This is a critical attribute, because software change is an inevitable consequence of a changing business environment.
Dependability	Software dependability has a range of characteristics, including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc.
Usability	Software must be usable, without undue effort, by the type of user for whom it is designed. This means that it should have an appropriate user interface and adequate documentation.

Table 8.2: Essential characteristics of well-designed software systems according to Sommervile ([227] p. 13)

to evaluate multiple attributes. Almost all of the architecture evaluation methods include the usage of scenarios as an appropriate method for architecture evaluation. It is checked whether the scenarios are covered by the architecture and a metric is defined to quantize the amount of changes needed to adapt the architecture in case the scenario is not covered.

Based on these results, a combination of ATAM and SBAR was chosen to evaluate the SRD architecture. These two methods are introduced in the following.

8.1.2 Scenario-based architecture engineering (SBAR)

In its original presentation, Bengtsson and Bosch established SBAR as a method for reengineering existing software architectures with the goal to improve architecture quality attributes. Due to its applicability during the development process, SBAR is in particular well suited for an iterative software development process.

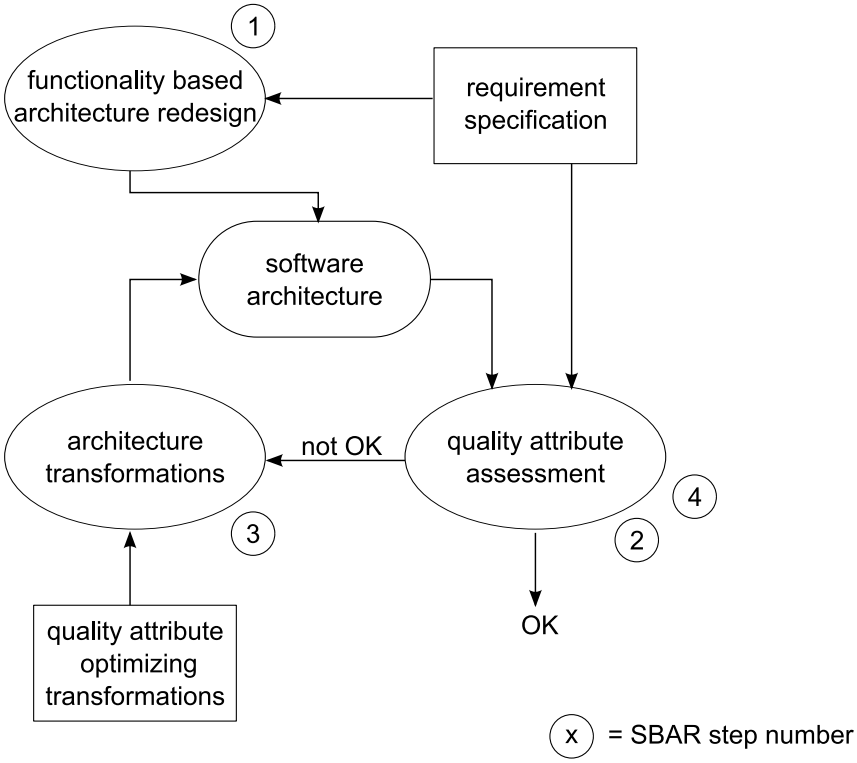


Figure 8.1: Graphical overview of the SBAR architecture evaluation method

The input for the SBAR method consists of the requirements specification and existing software architecture. The output of the evaluation is an analysis of the architecture and potential improvements on the architecture. SBAR includes four steps as depicted in Fig. 8.1. The four steps are 1) Incorporate functional requirements, 2) Software quality assessment, 3) Architecture transformation and 4) Software quality assessment. SBAR assumes that new requirements are added to an existing system design. However, the SBAR method is sufficiently generic so that it does distinguish between old and new requirements. Quality requirements that have not been stated explicitly during the initial functional requirement engineering phase can be added during this first step. Along with the requirements, the existing architecture is assessed. SBAR offers four different strategies for evaluation. The main part of the SBAR evaluation method bases on the construction of scenarios. However, also most of the critique

on the SBAR method targets the employed scenario approach, because of assumptions that are made about the scenarios [54]. SBAR proposes two strategies, a complete and statistical approach for the definition of scenarios. In the complete approach a full set of scenarios for an architecture has to be given. This is practically impossible. In the statistical approach a representative set has to be found. However, it is not well defined which requirements adequately represent the whole space of requirements. Next to the scenarios, SBAR proposes evaluation of the architecture with simulation, mathematical modeling and experience based reasoning. Based on the results of this evaluation the architecture is transformed by imposing an architectural style (e.g. a layered architecture, client-server model etc.), applying design patterns, converting quality attributes to functional requirements and distributing a quality attribute of the overall architecture to subsystems within the architecture.

The challenge for the overall procedure is to decide when a requirement has been achieved. While functional requirements can be evaluated by tracing requirements within the architecture, this is not possible for general quality attributes, because of their emerging nature. Instead, the quality attributes have to be embodied in a scenario that explicitly requires these attributes.

8.1.3 Architecture trade-off analysis method (ATAM)

One of the major advantages of the ATAM method over the SAAM and SBAR is that ATAM explicitly considers interactions among quality attributes and provides a method to prioritize among them [36]. The ATAM method consists of 9 activities that can be categorized in the four major groups of 1) Presentation, 2) Investigation and analysis, 3) Testing and 4) Reporting. An overview of the process for the analysis is shown if Fig. 8.2. One of the central tools within the ATAM method is the development of a quality attribute utility tree. The utility tree prioritizes quality attributes and maps them on concrete scenarios. For evaluating a particular design decision, ATAM captures the quality attributes of a particular solution with the attribute-based architectural style framework (ABAS) [36]. The ABAS has four parts: 1) Problem description, 2) Simuli/responses, 3) Architectural style, 4) Analysis. The first point addresses what particular problem is solved by a given architectural structure. The second describes the stimuli in terms of which quality attributes have to met in the response of the system. The third part contains a concrete description of the architecture in terms of components and interactions and the

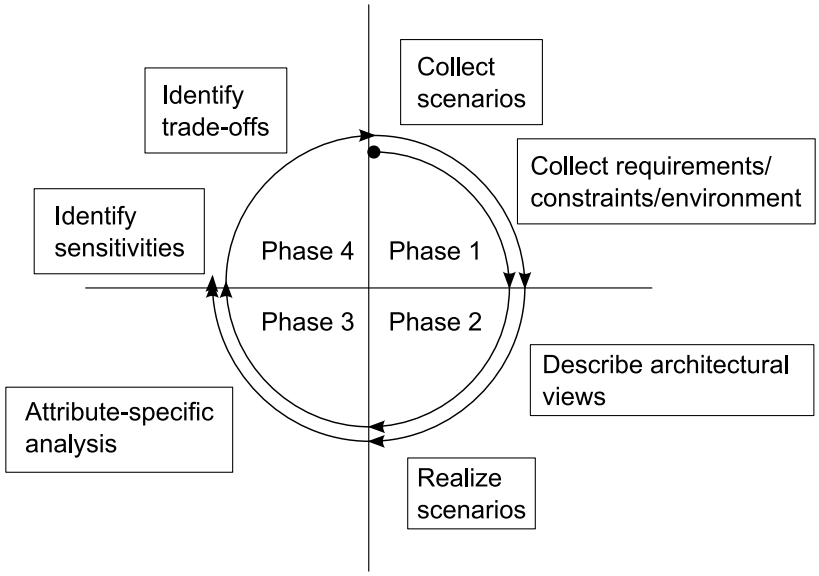


Figure 8.2: Analysis activities in the four phases of the ATAM (Source: [54] p.645)

fourth part provides the reasoning about decomposition and interaction of the components.

Based on a comparison of the different architecture evaluation methods, a combination of the SBAR and the ATAM method was regarded as the most appropriate method to evaluate the SRD architecture. Also Drobica and Niemelä recommended that a combination of methods that is adjusted for a concrete analysis case would improve the results [54]. The SBAR evaluation method is best suited for an iterative software design cycle, such as the one followed in this technological design. Following the SBAR method, quality requirements are imposed on a functionality based architecture design as described above. However, one of the major shortcoming of the SABR approach is the handling of scenarios, in particular for an open ended development platform as the SRD framework. Using the ATAM utility tree, scenarios can be grouped along quality attributes and thereupon prioritized and analyzed. Quality attributes are reflected by applied architectural styles in the architecture. Already during the design process,

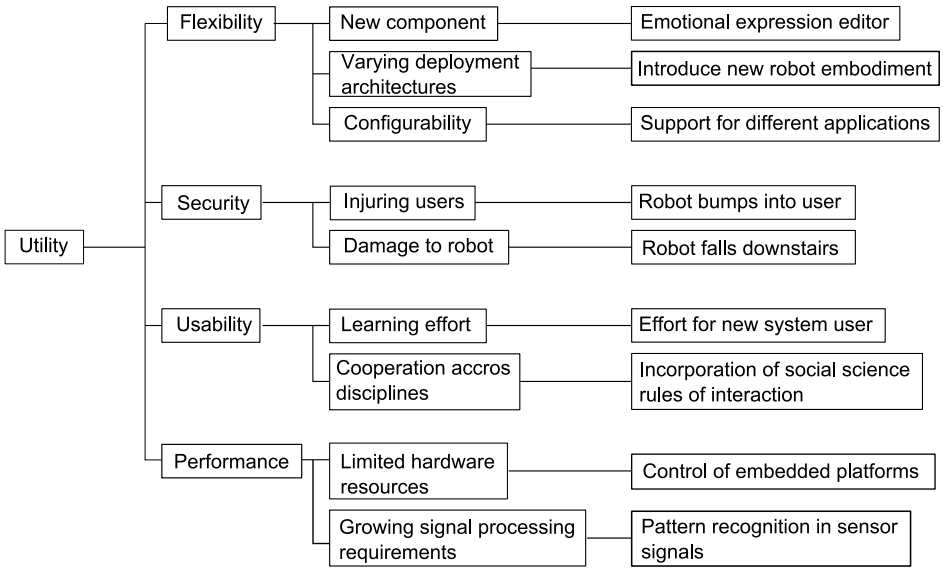


Figure 8.3:]
Utility tree to with scenarios to test the SRD framework

several architectural patterns have been followed. These design decisions can be analyzed using the methods as described above.

8.2 Utility tree

Following the SBAR analysis method, first the functional based architectural design is investigated. Traces of the functional requirements are provided and discussed in concrete scenarios. However, in some cases, the functional requirements are already linked to general qualities such as flexibility or performance. Therefore, functional requirements and non-functional requirements cannot strictly be separated for the SRD architecture. In fact, part of the proposed analysis methods of the SBAR evaluation method is to explicitly convert non-function requirements to functional requirements. In order to cover these requirements, this section develops a utility tree that creates concrete scenarios that represent particular quality attributes. The utility tree according the ATAM for the SRD framework is shown in Fig. 8.3. The most important quality for the SRD framework is a flexibility quality. The architecture needs to be sufficiently flexible to incorporate new domain knowledge as soon as it becomes available.

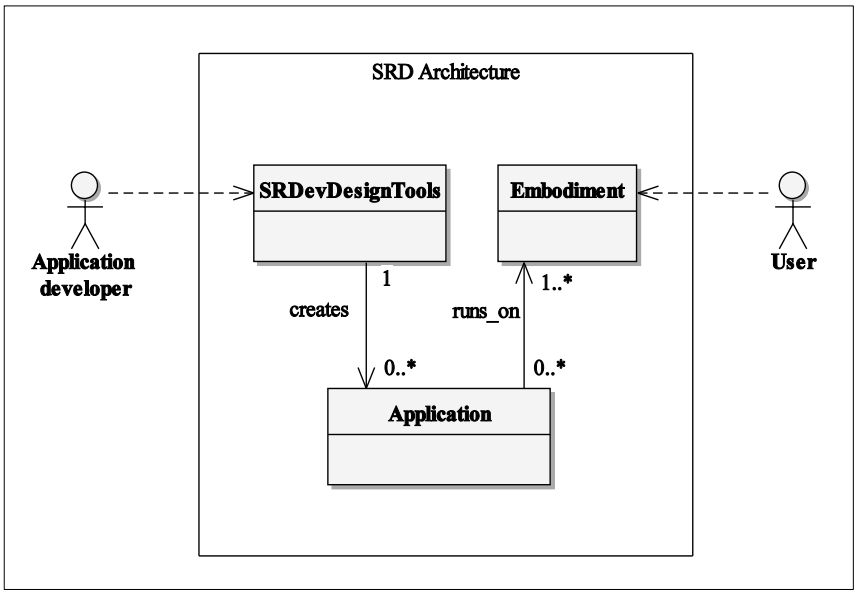


Figure 8.4: Boundary of SRD architecture

Next to flexibility, important quality attributes for the architecture are security, usability and performance. In order to test these quality attributes, in the following scenarios and examples are developed that explicitly require these attributes.

8.2.1 Flexibility

The flexibility quality is positioned at the top in the utility graph and has been embodied by three anticipated scenarios 1) New component, 2) Varying deployment architectures and 3) Configurability. Every of these scenarios is illustrated with a concrete example. These three scenarios have been derived by varying major building block within the SRD architecture as depicted in Fig. 8.4.

The first scenario considers the efforts necessary to introduce a new component. A new component becomes available for example when a new editor concept can be defined. In Chapter 6.4 three basic editors have been provided to satisfy basic needs of different application design disciplines. For realizing more complex scenarios also higher levels of abstractions might be necessary in order to keep the development of interactive applications

maintainable. For example, while for traditional animations it is affordable to hire a team of animators that optimize animations for a movie of a predefined length. This approach is not possible for interactive robots. The interaction is not limited by time or context, resulting in an unlimited amount of animations to cover every concrete situation. Therefore, one of the anticipated extensions is to automate emotional expressions with a high level emotion editor (see also discussion on affective perception of robot motion in Chapter 9).
Human-robot interaction is investigated on a variety of custom made robotic platforms [72, 28, 238, 9]. This variety is likely to increase as more application areas for interactive robots become feasible. However, the concrete hardware setup of future robots is impossible to predict. While in the functional requirements it was demanded that the development tools are independent from the deployment environment, no particular statement was made about the effort to introduce a new embodiment. However, already during the design of the architecture, multiple platforms have been considered, the iCat robot and the Roomba embodiment. In order to test flexibility, this scenario considers to introduce a new robotic embodiment to the architecture.
The third scenario considers configurability. Even though concrete robot hardware platforms are often carved to serve a particular functionality, multiple applications are conceivable using the same hardware. For example, the main functionality of the iCat robot is to serve as an interface. This interface can be applied in a variety of scenarios, including a game-buddy [143], TV assistant [161] and waiter application [208]. In order to facilitate a variety of applications, the architecture must be sufficiently configurable. For example, some applications might require image processing in order to identify objects, while others need Internet connectivity in order to download new content. Therefore, it must be possible to configure an application to only suit the needs of the concrete application.

8.2.2 Security

The second general quality attribute of the SRD architecture considers security. Most likely, security will become the most important software requirement as soon as applications are brought to the market. However, the main focus for the current architecture lies on research to develop the architecture. Nevertheless, also during development, there are a few security issues that have to be considered for the development of the SRD architecture. The two categories are (1) Insuring of the user and (2) Damaging of

the robot. Especially during a development process, several implementation mistakes can be made that lead to unpredictable behavior of the robot. Therefore the architecture has to provide mechanism to reduce potential damage.

In the first example, a mobile robot is considered that bumps into a user. In order to avoid insuring the user, for example in the case the user tries to pick up the robot while the wheels are still rotating, a security mechanism has to stop the robot. This is an important general software quality that has not been considered during the requirement engineering phase as this scenario describes malfunction of the software and not the regular use case.

The second example considers a case in which the robot damages itself. While for some cases the security of robot manipulators can already be handled on driver level, e.g., to avoid that a robotic arm breaks by pushing into its own embodiment, others cases need external control. The given example therefore tests how security can be realized on top of the basic driver hardware.

Most often a third scenario is considered, in which the robot damages the environment, e.g., by bumping into an object. However, from a technical perspective this last case can be handled by the first two and is therefore not considered in the utility tree.

8.2.3 Usability

Usability is commonly analyzed in the context of a user interface that is presented to a user. For a graphical user interface the graphical layout and the design play an important role. In the usability context also available documentation and learning effort for a user are evaluated [16, 227]. However, the concrete graphical layout is not considered to be architectural, but belonging to visual design. Only the type of control that is presented to the user is considered to be part of the architecture. [36, 227]. Therefore, the utility tree tests only for usability that is directly connected to the architecture.

In the first scenario learning effort has been considered to discuss usability of the architecture. This is motivated by the general positioning of the SRD architecture to offer tools for application developers with a variety of different backgrounds. In order to avoid that this demand leads to an increased learning effort, e.g., that the designer has to understand all tools before being able to utilize a particular one, this is included in the utility tree.

In the second scenario, the selection of an appropriate type of tools is considered, which are offered to the designer. In terms of developing applications for robots, the level of abstraction of the provided tools is essential. In order to test for this quality, a scenario has been constructed to incorporate tools for a new discipline with a different level of abstraction. As an example, the discipline of social science has been selected. From social science several interaction patterns can be derived that are valid for the design of robot applications [124]. How these can be incorporated in the overall architecture is discussed in relation with this requirement.

8.2.4 Performance

The last general quality attribute on the utility graph is performance. Performance has been considered, because of the special requirements of robotic hardware platforms. In particular, two cases are considered.
The first case tests the architecture in terms of control possibilities of embedded hardware. Embedded hardware has usually only very limited computational resources. The scenario tests for performance of the architecture in terms of how software components are deployed on concrete hardware platforms.
The second performance related scenario discusses how the architecture handles an anticipated growing need for computational resources for signal processing. For example, speech and gestures are important modalities for interaction for which considerable computational effort is necessary to analyze the semantics. It is discussed, how the architecture can handle the distribution of computationally heavy algorithms.

8.2.5 Functionality, reliability, efficiency, portability

Not considered in the utility tree have been the general qualities of functionality, reliability, efficiency and portability (see Table 8.1 on page 209). First of all, functionality is covered by the functional requirements, which is analyzed using requirement traces. Reliability and efficiency are usually measured in terms of numeric metrics, rather than evaluated using scenarios. For example efficiency can be evaluated for the communication architecture in terms of how many bytes are sent. As described during the design of the communication architecture of the Execution-Environment in Chapter 7.2.3, data is only sent if it is requested by a component. Similarly, for the Development-Environment the overhead of storing data on the blackboard system versus keeping the data locally could be evaluated.

In both cases, however, the major influencing factor is how a concrete application makes use of these facilities. In terms of the ATAM evaluation method this means that efficiency interacts with usability of which usability was in this context prioritized. Likewise, reliability has not been explicitly measured because of the research focus. Nevertheless, the provided prototypes and tools have proven to be sufficiently stable to be applied for the development in concrete applications as demonstrated in Chapter 9. Lastly, portability was not included in the usability tree, because it already has been captured by the functional requirements. Requirement 4.4 on page 94 demanded that different platforms must be supported. During the design process, this has been adhered by selecting only libraries that are available for multiple platforms.

8.3 Functional evaluation

In the following the list of requirements as developed in Chapter 5.3 is analyzed and concrete scenarios are discussed in combination with traces of the requirements on the architecture. Four different requirement categories has been defined: 1) Application designer requirements, 2) Development process requirements, 3) Domain requirements, 4) Framework requirements. From an overall architecture perspective, these requirements are grouped in categories defined by the origin of the requirements and not in terms of related constructs within the architecture. A single architectural feature can therefore respond to requirements of different categories.

8.3.1 Environment decomposition

To begin with, the overall domain decomposition already allows to trace multiple requirements as shown in Fig. 8.5. During the design process, an overall separation between Development-Environment and Execution-Environment was introduced. This general design decision adheres to a layered architectural style that groups related components on a the same layer. The connection between the layers is channeled by the concept of an application.
This general setup responds to requirement 1.1, which demanded that the level of abstraction is increased. A layered architecture suits this requirement as details of lower layers are only accessible through a specified interface. The layered approach is consequently adhered also in the definition of editors and operators to create applications. For example, using functional animations the designer describes basic movement patterns. On the

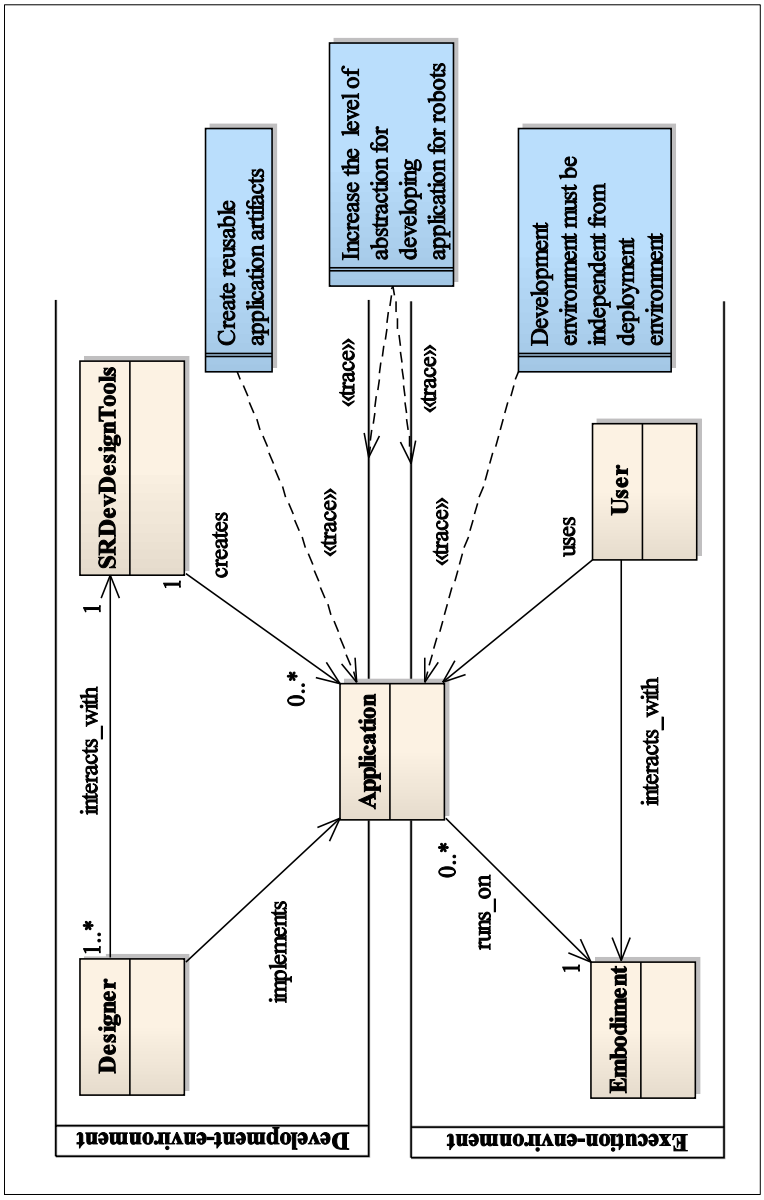


Figure 8.5: Trace of functional requirements along domain decomposition of environments.

next higher level these movement patterns are controlled through a set of parameters. These parameters, in turn can be calculated using parameter models. Ultimately, multiple application artifacts are combined to an application by an application logic.

Furthermore, the layered architecture fulfills the demands of requirements 2.5 and 2.4. The former stressed the necessity of reusing application artifacts. The layered approach decouples the application from the hardware, application artifacts can potentially run on different hardware platforms. The latter explicitly required the Development-Environment to be independent from the Execution-Environment. The architecture meets this requirement by defining an interface between development and deployment environment that decouples both environments from each other.

A particular concern is how a synchronized communication from the Development-Environment can be mapped on an asynchronous communication as defined for the Execution-Environment. For example, if in the development environment a relationship between a bumper sensor and actuator is established, it can be shown that the signal will reach the actuator in the next abstract instance of time. Therefore, it can be mathematically modeled that the robot shows the correct behavior in the presence of a particular stimulus. This is not possible for an asynchronous architecture, because no assumption can be made about the delay of communication. However, a delay metric can be computed that can be used to set boundaries for execution cycles.

Four basic communication scenarios between components are considered. The scenarios are depicted in Fig. 8.6. Three independent components are indicated by the letters A, B and C, respectively. A communication event between these processes is indicated by an arrow, which is annotated with a message that flows between them in the direction of the arrow.

First of all, all of these communication events are non-blocking, which means that the producer of the signal can continue as soon as the signal is sent. A message is buffered on the receiver side until the receiver discards it. In the first scenario, a producer A creates a message x, but no receiver is defined. In the Development-Environment this message is stored on the blackboard, but no other component reads the value. This scenario maps in the execution-environment to the case that no component has registered to the event. Therefore, the message is simply dropped and no synchronization delay is introduced.

In the second scenario, a component A sends a message x to component B. For the Development-Environment that means that in cycle c_1 the message

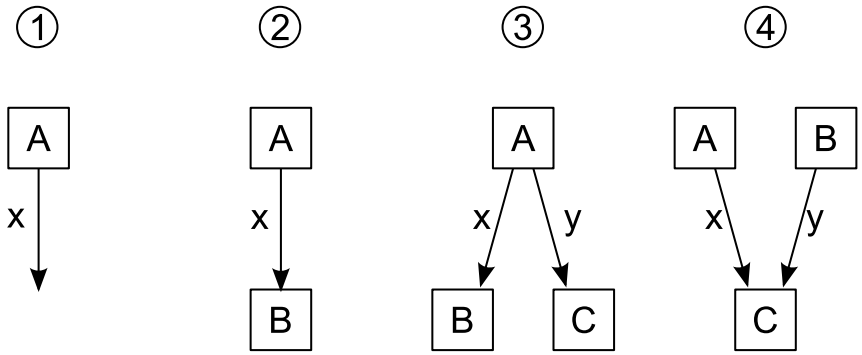


Figure 8.6: Basic communication scenarios sending messages x, y between processes A, B, C

is triggered and ready to be read by component B in the very next cycle c_2. In the Execution-Environment, both components may run with different clocks, which means that starting from the cycle c_1 in which the message is triggered, component A may perform c_A execution cycles and component B may perform c_B execution cycles until the message arrives. For example, from the sending of a bumper event, a motor controller may have updated the motor speeds 10 times and the bumper controller may have read the sensor value 5 times. This means that the message has from the point of view of the receiver component a delay of 10 cycles and from the sender component a delay of 5 cycles. From these values a synchronization rating s can be computed according to

$$s = \frac{1}{1 + c_A + c_B} \tag{8.1}$$

which gives an indication of the communication delay between the components. In the case the message is transported instantaneously, both components complete zero execution cycles which results in a rating of 1. The smaller the value s the more communication delay the message has.
In the second scenario a component A sends two messages x and y at cycle c_1. In this setup components A, B and C perform c_A, c_B and c_C, execution cycles, respectively. The synchronization rating is then computed by:

$$s = \frac{1}{1 + c_A + c_B + c_C} \tag{8.2}$$

Similarly in scenario 4, component C performs execution cycles until both messages from components A and B are available. The synchronization rating is therefore also computed as for case three.

In the Development-Environment, communication is handled by the blackboard system and in the Execution-Environment it is handled by localized point-to-point connections. These communication events can directly be translated from the Development-Environment to the Execution-Environment. The above synchronization rating can be used for example for program verification. A constraint could be defined that the synchronization rating may never be below 0.3, which means a bumper event may never arrive later than with three execution cycles delay.
Having evaluated the general architectural split between Execution- and Development-Environment the following sections discuss the requirements of every functional requirement category.

8.3.2 Trace application designer requirements

A trace of how the requirements from the application designer category map on the architecture is shown in Fig. 8.7. From the requirements trace it can be seen that all of the application designer requirements are met within the Development-Environment. As discussed above, requirement 1.1 is answered by a general layered model that is continued up to the operator and the editor level. Furthermore, the general editor concept responds to requirements 1.2 and 1.3. First of all requirement 1.2 denoted that the editor framework has to provide editing tools that agree with background knowledge and design approach from a particular field of expertise. Multiple editors have been introduced to the architecture resulting in a generalization of an editor interface that allows to extend the architecture with new editors. Requirement 1.3 is related to two concepts. First of all it is related to the editor concept, because editors naturally also present information to the user. The more explicit representation of this requirement is realized by the concept of a `DesignView`. In contrast to an editor, a design view is a purely passive element with regards to application development.
The architecture allows to present multiple views that have to be synchronized as stated in requirement 1.4. The problem of synchronization has been solved by the synchronous communication architecture. The synchronous architecture guarantees that all modifications are executed and visualized in the same virtual instance of time. In contrast to an external scheduling, this architecture insures a deterministic execution and therefore reproducible results.
Lastly, requirement 1.5 addressed that the different views must be easily accessible for the designer. This requirement has been realized within the visualization of the central engine of the development environment. Design

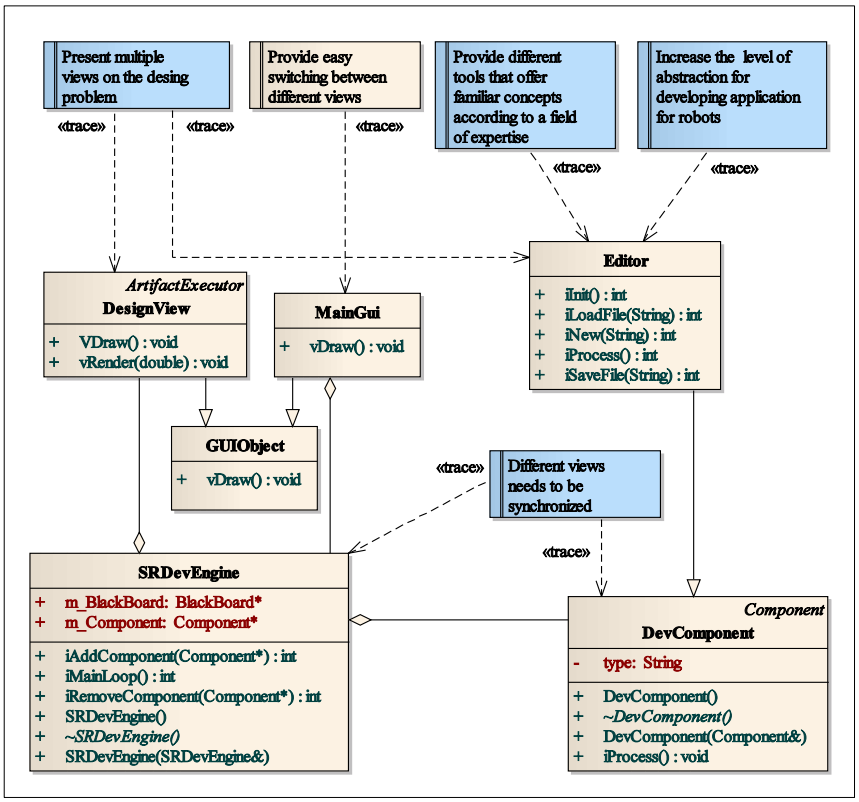


Figure 8.7: Trace of designer requirements on the architecture

views and editors are components within the architecture of which the central engine keeps a reference. Within the visualization of the current implementation, components may be freely organized.

8.3.3 Trace development process requirements

Requirements 2.4 and 2.5 have already been discussed in the context of the separation of the architecture in Development-Environment and Execution-Environment. A trace of the remaining requirements is depicted in the requirement trace in Fig. 8.8. The main requirement is to allow for rapid prototyping (RQ 2.1). This requirement has been realized by the `Editor` and `ApplicationArtifact` concepts. An editor allows to create new application artifacts. Therefore, an application can be created by reusing existing artifacts and trying new behaviors in combination with

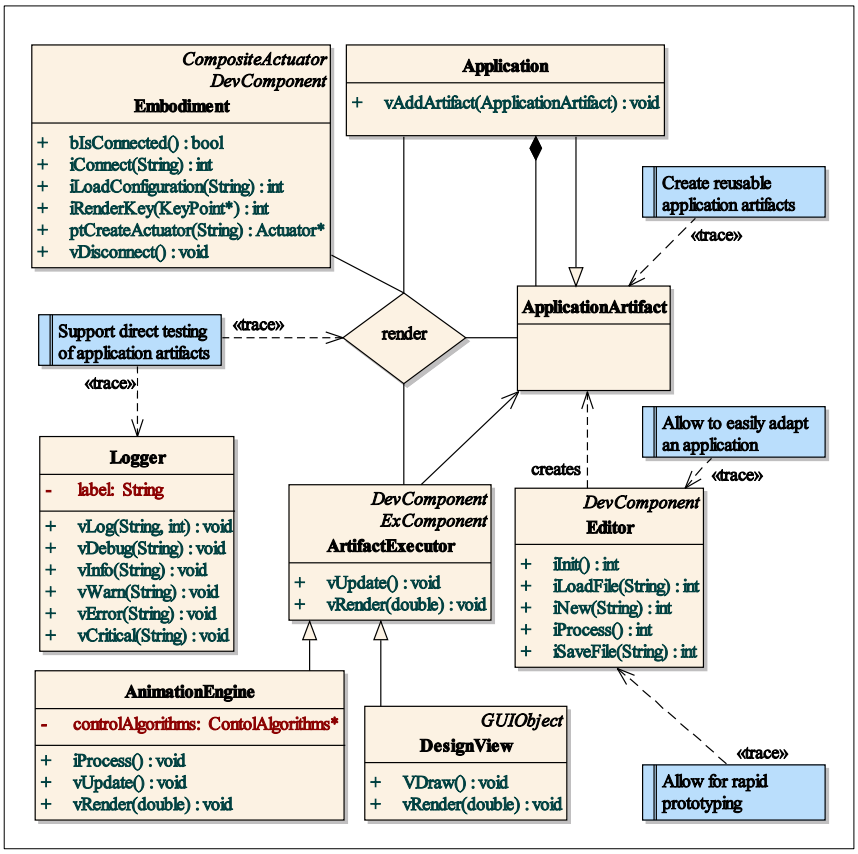


Figure 8.8: Trace of the development process requirements

.

an according `ArtifactExecutor`. Therefore, the `Editor` concept simultaneously fulfills requirement 2.2 to allow for easy adaptation of existing application artifacts.

Finally, requirement 2.3 asked for direct testing and debugging facilities. This requirement has been realized on multiple levels. First of all, the overall communication architectures of both the Development- and Execution-Environment allow to trace communication messages. Secondly, the synchronous communication protocol of the development environment insures for deterministic execution, which eases tracing of errors that have been logged with the logging mechanism. Lastly, the preview facilities provide a direct testing possibility before an application artifact is employed in the context of an application.

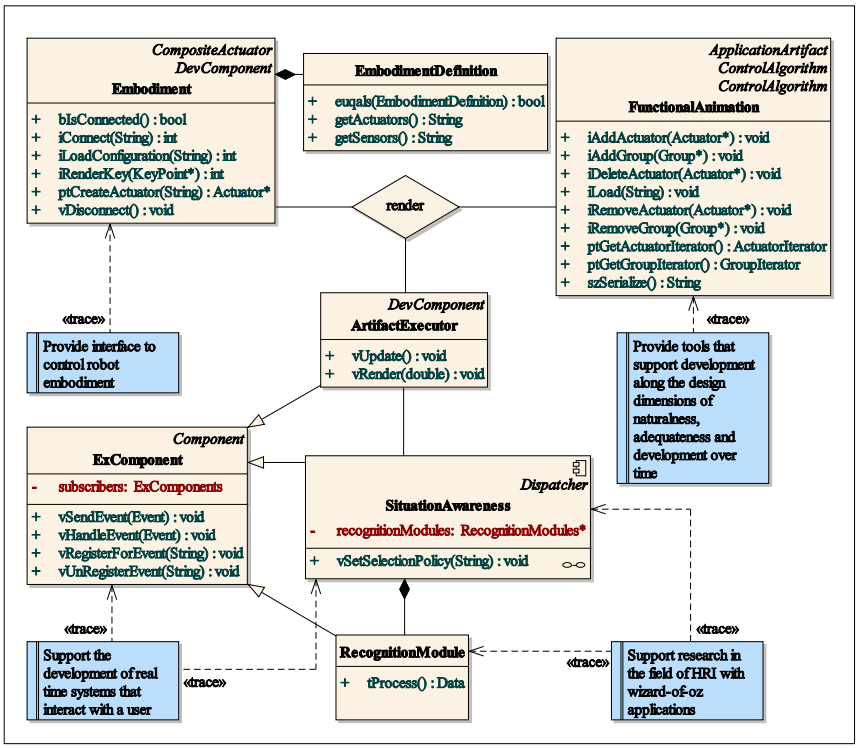


Figure 8.9: Trace of domain requirements

8.3.4 Trace domain requirements

A trace of the domain requirements is depicted in Fig. 8.9. Domain requirements originated from the general application domain of developing interactive applications for robotic interfaces. The first requirement (RQ 3.1) denoted the inherent coupling with a robotic embodiment. Due to the fact that there is only few standardization for robotic hardware available, a generic interface can only be defined on an application layer. The general embodiment class, in combination with the concept of an embodiment definition, generalize an embodiment as a set of sensors and actuators. The embodiment definition carries a type definition that is used to compare different embodiments. For example, the creation of a functional animation requires the knowledge of available actuators for the embodiment.

Functional animations in turn fulfill the basic requirement to be able to develop applications that adhere to the three design dimensions naturalness, adequateness and development over time as demanded by requirement

3.2. First, functional animations offer the control over expressiveness of the embodiment using the same principles as movie animation tools like Maya [4]. Secondly, they combine the power of dynamic scripting with keyframe animation technology using parameterized functional equations to adapt a behavior to the current situation. Lastly, these parameters can be transformed to different coordinate systems and manipulated during run-time, resulting in changing behavior over time.
In order to be able to react on the environment during run-time the system must be able to sense the environment (RQ 3.3). Furthermore, this interaction is subject to real-time constraints. This requirement has been realized by the asynchronous communication model of the Execution-Environment in combination with the situation awareness component.
Lastly, requirement 3.4 to support Wizard-of-Oz applications can uniformly be realized using the situation awareness concept. As described in Chapter 7.3.2, a recognition module can also be controlled using a control interface from the wizard. In this model, the wizard input is treated just like other sensor input. The component definitions carry the respective information of which data is provided by a particular recognition module.

8.3.5 Trace framework requirements

The category of framework requirements is mainly realized by the central component decomposition of the overall architecture. The requirement trace is depicted in Fig. 8.10. A central component interface has been generalized over the component classes of the Development-Environment and Execution-Environment. Most notably, the general component class includes a component definition that carries meta information about the component. The derivatives add a communication protocol between the components. This modularized specification is the major source of flexibility within the architecture, which also allows to dynamically extend the architecture (RQ 4.1). Furthermore, it allows third party vendors to contribute new components with specialized functionality (RQ 4.2). It therefore also facilitates the integration of different expertise as required by requirement 4.3. For example, a software company or research group with extensive knowledge on computer vision, can provide this expertize encapsulated in a recognition module. This allows an application designer to make use of the functionality without knowing the details of the particular component.
The last two requirements, to align with existing development platforms (RQ 4.4) and to integrate existing OPPR tools, (4.5) address the integration

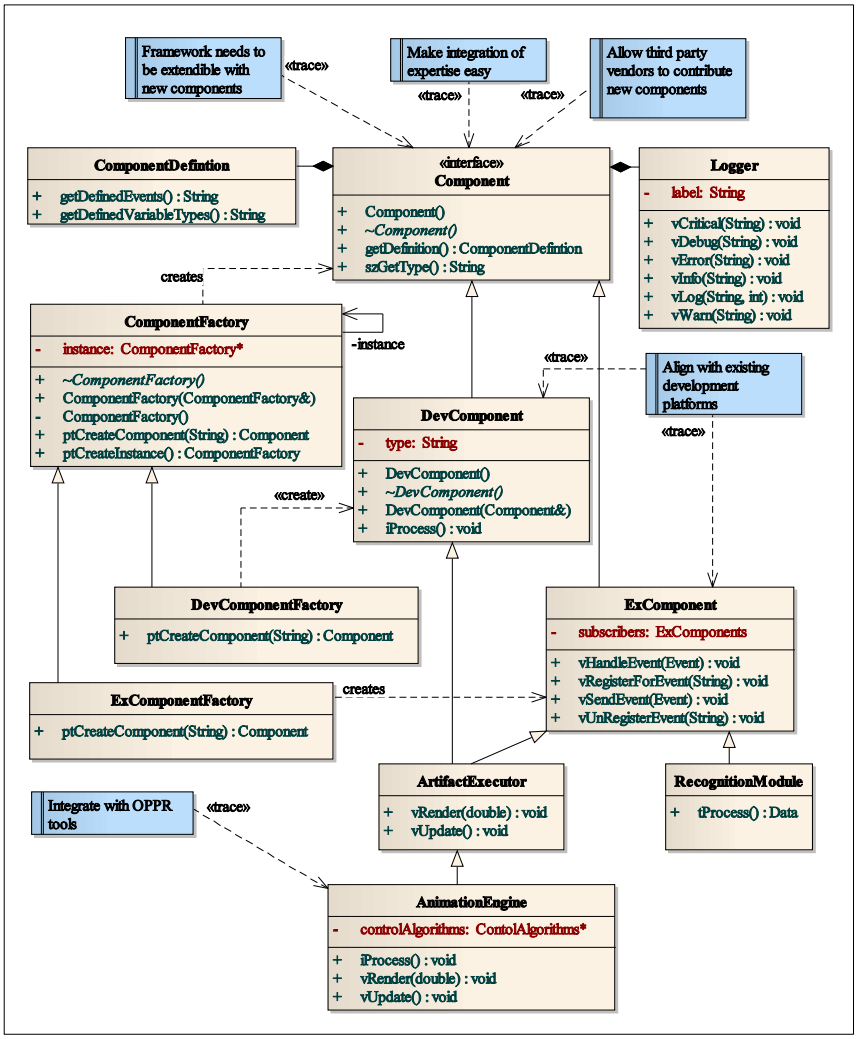

Framework needs to be extendible with new components
«trace»
Make integration of expertise easy
«trace»
«trace»
Allow third party vendors to contribute new components
ComponentDefintion
+ getDefinedEvents() : String
+ getDefinedVariableTypes() : String
«interface»
Component
+ Component()
+ ~Component()
+ getDefinition() : ComponentDefintion
+ szGetType() : String
Logger
- label: String
+ vCritical(String) : void
+ vDebug(String) : void
+ vError(String) : void
+ vInfo(String) : void
+ vLog(String, int) : void
+ vWarn(String) : void
creates
ComponentFactory
- instance: ComponentFactory*
+ ~ComponentFactory()
+ ComponentFactory(ComponentFactory&)
- ComponentFactory()
+ ptCreateComponent(String) : Component
+ ptCreateInstance() : ComponentFactory
-instance
«trace»
Align with existing development platforms
«trace»
DevComponent
- type: String
+ DevComponent()
+ ~DevComponent()
+ DevComponent(Component&)
+ iProcess() : void
«create»
DevComponentFactory
+ ptCreateComponent(String) : Component
ExComponent
- subscribers: ExComponents
+ vHandleEvent(Event) : void
+ vRegisterForEvent(String) : void
+ vSendEvent(Event) : void
+ vUnRegisterEvent(String) : void
ExComponentFactory
+ ptCreateComponent(String) : Component
creates
ArtifactExecutor
+ vRender(double) : void
+ vUpdate() : void
RecognitionModule
+ tProcess() : Data
Integrate with OPPR tools
«trace»
AnimationEngine
- controlAlgorithms: ContolAlgorithms*
+ iProcess() : void
+ vRender(double) : void
+ vUpdate() : void

Figure 8.10

of legacy systems. In the most general case, the communication interface of the communication architecture can be used. Both, the development- and Execution-Environment define a general communication protocol using human readable XML messages. More specifically, the animation engine of the OPPR framework has directly been integrated using the interface of an artifact executor.
Having mapped the functional requirements, the next evaluation section targets at stressing the architecture for more general quality attributes using scenarios and examples as defined in the utility tree.

8.4 Quality attribute evaluation

In this section it is discussed how the quality attributes are met by the design of the architecture.

8.4.1 Flexibility

Flexibility addresses how flexible an architecture is to accommodate changes. Usually it is measured by the amount of steps necessary to adjust the system. To a great extent, flexibility is defined by how the functionality of the system is broken down into components. In the literature three typical strategies are followed to achieve flexibility, which are indirection, encapsulation and separation [36]. Indirection decouples source and target of an operation, usually by involving a mediator. An example of this approach was followed in the design of the communication architectures. Encapsulation bases on the principle of information hiding by defining a concrete interface and leaving the implementation open. Therefore, interfaces become fixed points within the architecture with the result that a change of an implementation will not transcend the encapsulating interface. The last, separation, isolates not related constructs from each other. An example of this separation is how the graphical representation was decoupled from the underlying components. The graphical representation should be able to change independently from the implementation of underlying objects.
In the following, the three scenarios of the utility tree are discussed in terms of how flexibility is achieved within the architecture.
The fist scenario has been realized by encapsulation. A concrete interface has been defined for a component of the modular architecture, which hides the internal implementation. Furthermore, interfaces have been defined on different levels of abstraction. For this reason the architecture can be extended on top level by adding a new component or more concretely by

adhering to the interface of an editor. The concrete case has the advantage that the new editor can reuse predefined functionality such as the undo-manager and the cut-copy-paste facility.
Also the second scenario is met through encapsulation. The central interface to encapsulate an embodiment consists of the embodiment class and a corresponding embodiment definition. Furthermore, indirection has been used through the factory design pattern that handles the instantiation of an embodiment. Therefore, specific initialization parameters of the hardware can be handled without affecting the implementation of the development engine which keeps track of the components in the Development-Environment.
The last scenario of the utility tree targets the configurability of the architecture. This requirement is met by strict separation of orthogonal constructs. For example, the situation awareness component handles the sensory perception of an application while an artifact executor handles the control over the actuators. Modularization breaks the architecture down to independent small components that can be assembled in any combination. Within the situation awareness every signal processing algorithm is encapsulated within its own module. By defining input and output of the modules, they can freely be arranged to form a recognition network. Configuration is a matter of selecting and arranging predefined components. Within the SRD architecture, all configuration files have been consistently handled using a human readable XML syntax.

8.4.2 Security

The major difficulty for assessing system security is that security requirements describe what should not happen instead of what should happen [227]. In the context of software, security mostly concerns the management of rights, that is to assign limited access to users. In case these access rights are violated, for example if a party gains access beyond these rights, system security is compromised.
Sommerville [227] lists four different approaches for security checking that are experience-based validation, tool-based validation, tiger teams and formal verification. In the first approach the system is manually analyzed against a checklist of known security problems. In the second approach this analysis is supported by special tools, for example brute-force attacks on passwords. The third approach assigns a dedicated team the task to breach the security in order to solve discovered issues. The last approach requires that a formal specification of the security can be given against

which the security can be checked. Within the SRD architecture, currently no rights management is performed. However, access rights can for example be given on component level. A component is an identifiable entity that participates in the architecture using the communication interfaces. As a first step, these communications can be encrypted and only accepted from trusted components. In both defined environments these management can be placed in the central communication component that is either the blackboard or the dispatcher. Nevertheless, due to the research nature of the architecture these features have not been implemented yet and still pose a risk on the architecture.

Additionally to these traditional software security issues the application of robots introduces new security issues, because robots are able to act in the physical world. From an application development perspective, these security issues pose a problem even if no intruder tries to circumvent system security. Security in the context of human robot interaction has for example be discussed in the context of search and rescue and system safety [32, 105]. System safety can for many cases be addressed on driver level by restricting torque commands to a robotic actuator. However, also on application level security can be implemented. For this reason, two examples of this security class have been added to the utility tree. The first addresses that a robot injures a user and the second that the robot damages itself. In both examples, from a system point of view, the security issue originates from interaction with the environment. Therefore, security is at risk if certain signals from the environment are lost, e.g., the signals from a bumper or cliff sensor.

The SRD architecture defines an asynchronous execution model to meet the real-time constraints of interactions with the environment. Additionally, it defines point-to-point communication channels in which it can be verified if a message has reached a receiver. In the current implementation, the communication architecture uses the TCP protocol. However, based on the variety of different hardware platforms, no further generalization can be made. Therefore it is up to the application designer to configure a connection between bumper sensor and actuator. Additionally, the application can be configured to react to control events, e.g., an event that informs about a lost connection. In particular, a lost connection is also signaled locally to the component, so that appropriate measures can be taken.

8.4.3 Usability

In the literature, usability has received extensive attention, for example in the context of user-centered design, human-factors and ergonomics [18, 218, 66]. One of the key challenges for usability is to find appropriate interface metaphors that are easy to understand for a targeted user group. Bass et al. define usability by how easy it is for a user to achieve a desired result [16]. In this definition, they identify 5 areas: (1) learning effort for an unfamiliar user, (2) efficiency of an operation, (3) minimizing the impact of an error, (4) adapting to user needs and (5) increasing confidence and satisfaction.

From an architecture perspective, usability is analyzed in terms of how the chosen architectural style supports usability. For example, the first scenario in the utility tree raises the issue of learning effort for a new user. To some extent this usability quality is reflected in the functional requirement 1.2, which demanded to present familiar tools to the application designer. However, this requirement does not quantify usability over the whole architecture. The question to answer is if a global knowledge of all components is required before a particular function can be used.

Within the SRD architecture modularization keeps independent components separate from each other. Therefore, the user needs only to learn the interface of a particular component. In order to create a functional animation, the user needs to be familiar with animation terminology such as keypoint and interpolation. Therefore, this component is independent from an application logic editor. Furthermore, the architecture builds on common editor techniques such as cut-copy-paste, which on the one hand require the user to be familiar with these constructs, but on the other hand also insure consistency and minimize the effort of learning new constructs and interface gestures.

The second scenario stresses the architecture to introduce a new application designer with a social science background. In the SRD architecture such an addition is realized by providing a new editor concept and application artifact. For example, social rules might be given in terms of templates [124]. A new editor can support the development process of such templates and store them as a new application artifact. The important point in terms of usability is that the architecture allows several disciplines to contribute knowledge to the application development with specialized tools, which minimizes the integration effort between the domains.

8.4.4 Performance

Like usability, also performance includes architectural and non-architectural aspects. For example, specification of communication between components or allocation of components to hardware belongs to the architecture description. However, the concrete implementation does not belong to an architecture [16].

From an architecture perspective, performance describes the ability of a system to allocate computational resources [36]. In this context, Abbott and Garcia-Molina present a performance evaluation and ranking of concurrent control strategies [1]. The SRD architecture defines two different environments with two different control strategies. For the Development-Environment a desktop computer environment is assumed that is able to meet the demands of visualization of the graphical interfaces. The complexity of computation scales with the number of components that are loaded with the main development engine. As described in Chapter 6.7 on page 138 the communication architecture allows to distribute components over several processing nodes. However, the major bottleneck for the architecture is the synchronized communication approach, because the execution of components is sequential. Therefore, this design decision treats performance for maintainability so that no additional synchronization code need to be maintained for scheduling of the components.

The major performance requirements for SRD framework arise from the hardware limitations of the robotic deployment environment. In order to evaluate the architecture for these constraints, the utility tree defines two scenarios that target at the execution of applications.

The first one relates to the control of embedded hardware platforms. In order to execute an application two approaches can be followed. In the first approach, the artifact executors and required recognition modules have to be ported to the embedded platform. The advantage of this approach is that every of these components can be highly optimized for the available processing unit. Application artifacts may still be developed on a desktop environment, because the hardware access is delegated to an embodiment definition. The second approach treats the robotic hardware as a closed system that is controlled through a defined interface. For example the iCat robot and the Roomba robot have been controlled using this approach. An embodiment definition encapsulates the hardware interface and makes it available within the SRD architecture.

In both approaches, components may be distributed among processing nodes using the asynchronous communication architecture. This is an im-

portant quality in particular, because it facilitates to balance the working load. Additionally, components can be controlled during run-time using control events. For example within the situation awareness, components may be temporarily disabled, if their output is not required. However, the responsibility for allocation lies fully on the application configuration and application logic. That means that in the current implementation no automatic load balancing is performed. However, automatic load balancing also introduces new overhead which increases the requirements on the hardware.
In summary, the above analysis has mapped the functional requirements on the architecture and illustrated which constructs are responsible to realize a certain function. Furthermore, quality attributes have been discussed in to concrete scenarios, highlighting how certain architectural styles contribute to achieve a quality requirement.
However, a fundamental problem of a design tool is to prove its basic applicability to given design domain based on the produced results. The provided design framework merely serves as a facilitator. The final result depends on the one hand on the skills of the designer and on the other hand on the quality of the provided tools. For this reason, the SRD architecture has also been evaluated at hand of two case studies. The first study develops a tutoring application for the iCat robot and the second study investigates how robot motion is perceived by a user in terms of affective content. The latter one is reported in the next chapter.

Chapter 9

Case study: Emotional messages in motion

The SRD framework has been applied in two case studies. In the first, it has been used to vary the degree of social supportive feedback in a tutoring scenario. The full study is reported in [211]. Education literature suggests a relationship between the degree of social behavior of a robotic interface in a tutoring application and its effectiveness in terms of learning performance of the student [234]. The challenge for an application designer is to control the degree of social behavior of the character. As interface the expressive Philips iCat robot was chosen. The SRD framework has been used to modify the behavior of the robot along five dimensions: 1) role model 2) non-verbal feedback, 3) attention guiding, 4) empathy and 5) communicativeness. Expressive behaviors and emotional expressions have been created using the functional animation editor. These animations were triggered by an application script that has been developed using the Robot Interaction and Behavior Markup Language (RIBML). Furthermore, the animations were parameterized and linked to the sensory input from the situation awareness component.

The application has been tested with 16 children of the age of 10 to 11. An example interaction of the tutoring application is depicted in Fig. 9.1. Even though no qualitative judgment can be made about the effectiveness of the design tools, the experiences from this case study have demonstrated that the SRD framework can successfully be applied to create a social robotic interface.

In total, four persons worked with the tools, of which two had no formal training in software engineering. They reported that especially the

Figure 9.1: Example interaction between participant and iCat robot in a tutoring application.

graphical editing methods and the preview facilities helped to create desired expressions for the robot. Furthermore, the graphical user interface was sufficiently responsive to test various parameterizations without recognizable lag. Also when executing the final application on the robotic embodiment, the animations were rendered smoothly. For this reason, and because the focus of this design thesis is on architecture level rather than on performance optimizing, no further performance analysis with regards to execution time or memory usage was performed. Nevertheless, based on the experiences it can be stated that the framework actively supported the development of the social robotic interface.

One thing that the tools missed, however, was the possibility to control the emotional expression of iCat on a high level of abstraction. The designers had to rely on their intuitive interpretation of iCat's behavior from the preview facility and observing the actual robot. Just by varying the motion patterns of iCat, without changing the facial expression, could give very different emotional impressions.

People's interpretation of robot motion has been investigated in a second case study of the SRD framework, which is reported in the following. This case study is of particular relevance, because the findings might impact the development of the framework itself. That is, it gives an outlook on

next generation design tools. The study presented in this chapter has been submitted for publication at the 5th ACM/IEEE International Conference on Human-Robot Interaction (HRI 2010).

Abstract – Nonverbal behaviors serve as a rich source of information in inter human communication. In particular, motion cues can reveal details on a person's current physical and mental state. Research has shown that people do not only interpret motion cues of humans in these terms, but also the motion of animals and inanimate devices such as robots. In order to successfully integrate mobile robots in domestic environments, designers have therefore to take into account how the device will be perceived by the user.

In this study we analyzed the relationship between motion characteristics of a robot and perceived affect. Based on a literature study we selected two motion characteristics, namely acceleration and curvature, which appear to be most influential for how motion is perceived. We systematically varied these motion parameters and recorded participants interpretations in terms of affective content. Our results suggest a strong relation between motion parameters and attribution of affect, while the type of embodiment had no effect. Furthermore, we found that the level of acceleration can be used to predict perceived arousal and that valence information is at least partly encoded in an interaction between acceleration and curvature. These findings are important for the design of behaviors for future autonomous household robots.

Non-verbal communication such as symbolic gestures and motion patterns play an important role in human communication as emotions are almost exclusively conveyed non-verbally. For this reason it is investigated to what extend robots are able to display similar expressions. Robots such as Kismet [28] and iCat [238] are especially designed for social and emotional interaction with humans. However, currently available household robots such as Roomba are not equipped with special expressive degrees of freedom for communication, but nevertheless, people are ascribing emotions and respond to Roomba's behavior in social terms [74].
Designing expressive movements for robots aims at using motion as modality to convey information, for example about the status of a device. Furthermore, expressive behaviors improve the overall perception of the device.

The research of Takashi and Hiroshi showed that specific subtle motion cues can have a measurable effect on how the robot is perceived [167]. They analyzed motion diversity in the context of a reach gesture and found that there is a dependency between the context and the retract motion of the gesture. Furthermore, they showed that the overall impression of the robot is improved when these subtle differences are modeled in the behavior.
The design challenge is to create motion patterns that give a desired impression. For designing expressive and communicative behaviors it is important to know what features cause the interpretation of intentions and emotions [82]. The interpretation of motion in terms of affect or emotions is particularly relevant for our research.

9.1 Perception of animacy

Several psychological, social, biological and evolutionary models and theories for explaining the ability of people to interpret motion of objects have been developed. For example, one theory for peoples' ability to interpret motion suggests that the visual system is sensitive to energy violation [215]. This means that whenever people perceive a change of the current motion that is not caused by external events following the Newtonian laws, these objects are perceived as animated. However, even though this rule seems to explain most of the cases for perception of animacy, it has also been shown that the orientation of an object with otherwise identical motion trajectories is enough to influence the perception of animacy [215].
Blythe et al. give an evolutionary explanation for the perception and interpretation of motion features [22]. They state that being able to categorize the behavior of a predator, e.g., if it is searching for food or exhibits mating behavior could make the difference for the survival of the prey. Also for gregarious animals the ability to express a state or intentions to be recognized by other members of the group might hold the key for the survival of the group. Blythe argues that at least the benefits of such movements must have exceeded the energy costs, otherwise they would not have evolved.
Another model for interpretation of motion patterns is that social reasoning helps to make sense of an observation. If objects change direction without an obvious reason, people tend to use their social reasoning to explain the phenomenon, i.e., by internal drives and needs. That is, people use a "it is like me" reasoning to explain the observations [59].
A multitude of studies in psychology have analyzed the phenomenon of ascribing lifelike attributes based on motion patterns. One aim in this

research is to isolate the underlying mechanisms and their development throughout childhood. For example, Boon and Cunnigham give evidence that emotion decoding based on expressive movement is acquired in an early developmental stage at the age of 4–5 years [24]. The work of Leslie gives evidence that infants of six month are already able to perceive causality in motion patterns [144]. Rakison and Poulin-Dubois report that with six month infants already begin to attribute characteristics of animacy to people [194]. However, the question if the perceptual processing of causality and animacy is innate and where perceptual processing and cognitive processes link has not been resolved yet [215]. There has been no agreement if the perception of animacy is solely based on stimulus or if higher level cognitive processes are involved. For example, it has been argued that the perception of animacy requires the knowledge of causal relationships [44]. In contrast there is also evidence that simple motion cues are sufficient to judge animacy and intentions [22].

Overall, there seem to exist specific motion patterns that stimulate attribution of animacy and are therefore important for the design of expressive communicative behaviors. For example, Dittrich and Lea conducted an experiment where participants had to detect a biological meaningful motion of a moving character within a group of distractors [53]. They found that more direct motion is perceived as more intentional. The more interesting result was that even if the target for the intentional motion was removed, participants were able to detect the intention. Furthermore, the difference in appearance, resembled by difference of brightness in the experiment, did not lead to a difference in interpretation of the motion. Dasser et al. explored the ability to attribute intentionality to motion patterns in an experiment in which they showed movement patterns of two balls to preschool children [44]. In their experiment physical causality was interpreted with psychological motivation such as A hit B, because B made A angry. They argued that this type of interpretation requires a representation of the internal state of the object and showed that the children were able to interpret the motion without making a distinction between animate and inanimate motion, which was kept constant in the conditions.

The above results suggest that there exists a relationship between motion and perceived animate characteristics such as intentions and emotions. While motion can be observed and parameterized directly this is not possible with affect. Therefore, we discuss in the following section models of emotions that have been proposed in literature and introduce the affect assessment methods used in this study.

9.1.1 Emotional model

The ability of mechanical rendered faces to express facial emotions has received plenty of attention in literature [9, 28]. In this study we are interested in the perception of how robotic movement patterns are interpreted by a human observer in terms of emotions.
Until now, a number of different psychological models for the cognitive structure of emotions have been proposed. An extensive discussion on emotional models and the experience of emotions can be found in [147, 7, 35, 38]. In general, two models have found wide acceptance and are supported by empirical evidence.
The first describes emotions as a combination of basic emotions. Ekman found that facial expressions for the six emotions of anger, surprise, disgust, happy, sad and fear are universally recognized [61]. Each of these basic emotions describes an unipolar dimension containing the activation of a particular basic emotion. However, it is not clear which emotions make the basic set out of which all other emotions can be constructed [108].
The second model represents experiences of emotions as points in a continuous two dimensional space. Russell found that most of the variance of emotional perception can be accounted for in a two dimensional space with the axis of arousal and valance. This model is referred to as *circumplex model of affect* [207]. The results of Russell have been repeated in several other studies that found the same axis or rotational variants and resulted in the development of multiple scales to measure different degrees of affect in this two dimensional space [39, 164]. Some studies have extended the model, for example by a third dimension representing dominance [139].

9.1.2 Assessing affect

For our research, we adopted a similar approach as Pollick et al. [192] and Lee et al. [142] and measured emotion according to a two dimensional parameterized model of emotion. Pollick et al. found that emotions perceived from arm motion can be clustered in a space with the two main axis of valence and arousal similar to Russell's circumplex model of affect [192]. We also followed Pollick's argumentation that similar measurements can be used both for measuring one's own experiences of emotion as well as assessing the emotional state of someone else. Literature gives evidence that the same, or at least very similar processes are involved when assessing one's own experience of affect and recognizing affect in others. An overview of assessment methods for affect can be found in [108].

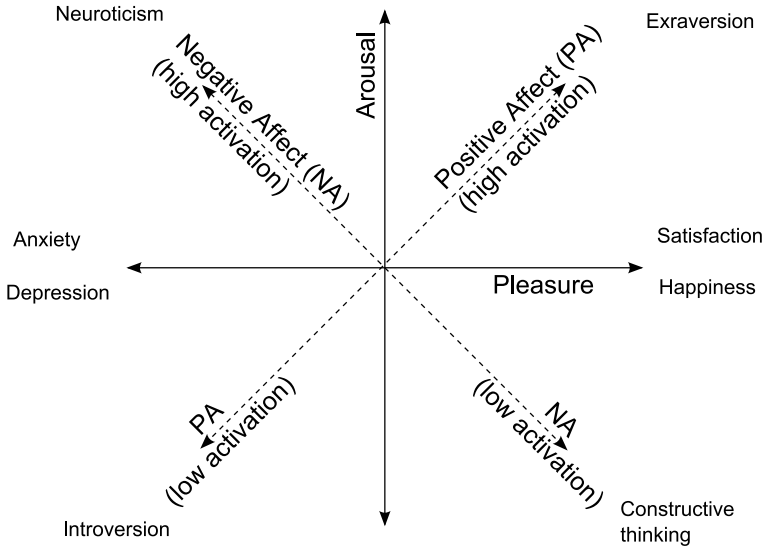


Figure 9.2: Simplified version of a two dimensional space of affect(derived from Larsen and Diener [141])

We selected the PANAS [247] scale to fit best our needs. First of all, plenty of studies have been reported using the PANAS and the results showed high validity and reliability ($Cronbach's\alpha = 0.89$) for a general population [39]. The scale and rating instructions are freely available and are quick to administer. Furthermore, the PANAS has also been administered to rate the affective state of other persons, not only to assess the emotional experiences of oneself. For example, it has been successfully administered to mothers to assess affect of their children [49]. The PANAS scale measures a dimensional model of emotions, which allows to parameterize an emotional state by a coordinate in a two dimensional space. It measures *positive affect* (PA) and *negative affect* (NA) which consist of 10 items for every of the two constructs.

Additionally to the PANAS scale, we administered a second scale based on a slightly different model of affect, comprising the three independent dimensions *pleasure*, *arousal* and *dominance* (PAD) [163]. In literature these dimensions are also sometimes referred to as valence or pleasantness for pleasure or as activity for arousal. Both the PANAS and the PAD models are rotational variants in the same two dimensional space [164]. A simplified version of the space can be found in Fig. 9.2.

However, when comparing both models, Mehrabian found the PAD to be superior to the PANAS model, because certain affective dimensions such as 'dependent-disdainful' (e.g., amazed, fascinated, impressed, loved versus indifferent, selfish-uninterested, uncaring) and 'docile-hostile' (e.g., consoled, protected, reverent, sleepy, tranquilized versus angry catty, defiant, insolent, nasty), had no counterparts in the PANAS(p. 350) [164]. Another advantage for the PAD is that it can be administered using pictographic representations of the three dimensions, in literature referred to as 'Self-Assessment Manikins' (SAM) [139, 26]. Bradley and Young showed that the SAM are highly correlated to the Semantic Differential Scale (developed by Mehrabian and Russell [165]), but are faster to administer and are not subject to language misinterpretations [26]. We expected that an iconic representation of emotions might be easier to understand and apply to inanimate beings such as a robot. In this sense, rating the affective state of the robot could be compared to rating the state of a cartoon character. In both cases the participant is consciously aware that the character is not alive and does not have 'real' emotions. Additionally, it has been shown that anthropomorphic characteristics of the embodiment are not a requirement for the attribution of an affective state. For example, Disney has demonstrated that a sack of flour can be animated in a way so that it expresses various types of emotion [231].
Nevertheless, we figured that the PANAS is still useful for our case, especially because its validity has also been demonstrated for assessing the affective state of others. To the author's knowledge, the SAM have not generally been administered to assess affective states of others. For our case this is essential, because we are interested in what affective state participants perceive in the robot motion.

9.2 Selection of motion features

In literature, several studies have been published that analyze the perception of particular motion. In order to determine which motion features are most influential for the perception of emotion we analyzed the main effects of several studies that investigated the perception of motion. For example, Tremoulet and Feldman have shown that already two cues are enough to give an impression of animacy: 1) change in speed and 2) change in direction [235]. In the following we give an overview of the field.
Camurri et al. aimed to automate the recognition of emotional content of expressive gestures made by dance performances [31]. They asked actors

to perform a dance with four different emotional expressions: anger, fear, grief and joy and computed motion features derived from Laban Movement Analyis: overall duration, contraction index, quantity of motion, and motion fluency. From these dimensions they composed a feature vector that contained measurements of the full body motion trajectory as well as relative motion of body and limbs and used it as input for training a classifier. They found main effects for example for duration and quantity of motion, which are related to changes in speed and trajectory.

Similarly, Gaur et al. aimed at automating the recognition of animate and inanimate characteristics solely based on motion features [82]. They represented the motion data as a vector of motion angle, which gives the direction of motion, and the moved distance between breaks. From this raw representation they calculated more specific motion features, including mean distance, mean rotation, range of distance, range of rotation, variance of distance, variance of rotation, spline coefficients representing the sharpness, and an energy metric that calculates the energy that the objects gains to give the impression of being animated. Beside these continuous variables, they also calculated binary flags indicating if the object is static and if it moves on an exact straight line. With these input data they trained a Bayes classifier with a human annotated training set and performed a knockout analysis with every feature, to investigate which features contributed the most to the perception of animacy. They found that a combination of spline coefficients, change in velocity and direction together with the energy feature hold the most information for classifying a motion as either animate or inanimate. The absolute values seemed to be of less importance.

Bethel and Murphy reviewed different methods for affective expressions, among others using motion [21]. They found that depression is connected with slow and hesitating movements while elation is connected to fast expansive motions. Blythe et al. analyzed motion patterns for basic behavior patterns of pursue, evade, fight, court, be courted and play [22]. In their experiments they let participants control an ant on the screen and asked them to move according to one of the predefined basic behaviors. They validated the resulting trajectories in a confusion matrix and found that participants were able to categorize the motion trajectory as it was intended. An interesting result was that participants were even able to judge the behavior when the target, i.e., the second ant was taken away. These results stress the informational content of motion that is independent from the context. Analyzing specific motion features in detail, Blythe et al. tested seven cues relevant for the categorization: 1) relative distance, 2) relative angle, 3) rel-

ative heading, 4) absolute velocity, 5) relative velocity, 6) absolute vorticity and 7) relative vorticity. To find out which motion parameters carried the most information for the categorization task they trained a Categorization by Estimation (CBE) algorithm. They found the following order of importance of the features: 1) absolute velocity, 2) relative angle, 3) relative velocity, 4) relative heading, 5) relative vorticity, 6) absolute vorticity and 7) relative distance. Additionally, they quantified the information that the context holds for correctly interpreting a particular behavior. Therefore, they measured the performance drop that was observed when the context was taken away.

Pollick et al. calculated from a point light display of human body motion [192] the following movement features: 1) wrist kinematics, 2) average velocity, 3) peak velocity, 4) peak acceleration, 5) peak deceleration and 6) jerk index. They found that kinematic features correlated with the activation dimension of arousal and valance. Energetic motions were positively correlated with shorter duration, acceleration, jerk, greater magnitudes of average velocity, and peak velocity. The authors explained that the arousal could more robustly be decoded by the redundancy of motion information, because all of the kinematic measures were related to the arousal axis. Furthermore, they conjecture that information on the pleasantness axis is encoded in the phase relations between the different limb segments.

Lee et al. present a relational framework between motion features and emotions [142] for which they used a two dimensional emotional model with the axis of valance and arousal. They generated multiple motion trajectories for which they varied the degree smoothness, speed and openness. They found a positive correlation between velocity and activation axis and a positive correlation between pleasantness and smoothness, but they could not find an effect for openness.

From the above results it appears that especially relative motion features hold important information for categorizing a motion trajectory. In all studies changes of speed and changes of direction had an effect, while the absolute values seemed to be of less importance. Only the study of Blythe et al. found absolute velocity to carry most information, but they also found that the absolute ordering of the features was of less importance. The study of Gaur et al. found the absolute values explicitly of less importance [82]. Based on these results we chose to focus our study on the motion parameters of acceleration (representing differences in speed) and curvature (representing differences in direction).

9.3 Measuring perception of motion

From our discussion above we selected to investigate the parameters acceleration and curvature and to focus on the perception of motion of a robot with a physical embodiment. Multiple studies have shown that the embodiment of a robot has an effect on how it is perceived [15, 128, 13, 193, 122, 106, 153]. Furthermore, depending on the embodiment two different types of motion are possible, to which we refer to as *external motion* and *internal motion*. With external motion we refer to motion 'external' to the embodiment, i.e., movement of an object from location x to location y in a defined space. With internal motion we refer to motion internal to the embodiment, i.e., movement of the limbs.

Therefore, we chose for our study two embodiments with two different types of motion, namely the iCat robot for implementing the internal motion and the Roomba robot for implementing the external motion.

9.3.1 Robotic embodiments

For this study we chose two robotic embodiments, namely the iCat robot and the Roomba robot (see Chapter 2.3). The iCat robot is a non mobile robotic research platform developed by Philips Research for human machine interactions. The robot is depicted in Fig. 9.4. The iCat robot has the shape of a cat and is approximately 40cm tall. It has an animated mechanical face with 13 degrees of freedom to express basic emotions, such as, happiness, sadness or disgust. Furthermore, iCat is equipped with a camera, a microphone, a distance sensor and four touch sensors. For our experiment we focused on the pan and tilt degrees of freedom of the head only. The expression of the face was kept neutral in order to avoid an interpretation of iCat based on the symbolic expression of the face rather than the impression of the movement patterns.

The Roomba robot is a commercially available vacuum cleaning robot developed by iRobot. The robot is depicted in Fig. 9.5. Roomba has a circular shape with a radius of approximately 15cm. It has a differential drive system, consisting out of two velocity controlled wheels that can be controlled via a serial interface. The interface defines two parameters, namely radius and velocity to control the motion. Roomba has a state based controller, which means that it keeps driving on a circle with given radius and velocity until a new command arrives. More complex trajectories can be approximated from small arc segments. Additionally, the interface defines special

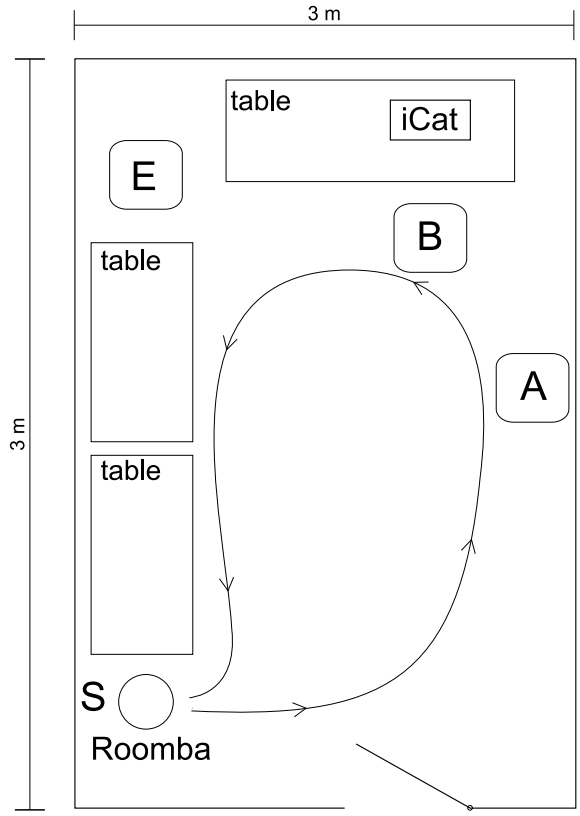


Figure 9.3: Experimental setup of the room indicating the movement of Roomba, starting from position S. In the Roomba condition participants were placed in position A and in the iCat condition in position B. The experimenter was placed at position E.

case parameters for driving straight or rotating on the spot. We tested and evaluated the drive accuracy of the robot as described in Chapter 7.4.

9.3.2 Motion pattern generation

The goal of this study was to systematically vary certain motion features. Therefore, we used the graphical animation tools as described in [212] to create behaviors. The editor for creating animations for iCat is depicted in Fig. 6.15 on page 127 and the editor for creating a path for Roomba is depicted in Fig. 6.18 on page 130.

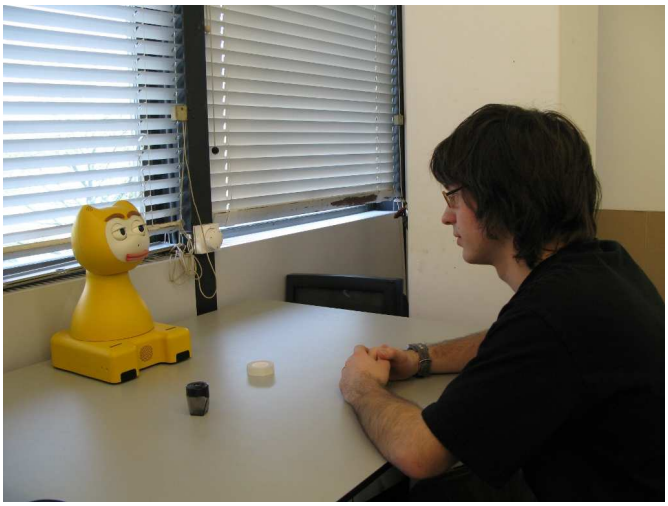


Figure 9.4: Sample interaction in the iCat condition.

Three different values for curvature and acceleration were created for both embodiments, resulting in 9 animations for every embodiment. For creating the animations both robots were assigned a simple task. For Roomba, we defined a circular trajectory through the room as depicted in Fig. 9.3 so that the robot would start from a defined home position S, drives through the room on the indicated trajectory and returns to the home position afterwards. On this route, Roomba passes the participant at position A. An example picture with Roomba is depicted in Fig. 9.5.
In the iCat condition we placed two objects in front of iCat and defined an animation to look at both objects. We realized the animation with the iCat editor by moving the head so that the robot would start from a central position, look first at the left object then at the right object and finally return to a central position. The participants were seated with an approximate distance of 80 cm to iCat, in Fig. 9.3 marked as position B. A sample picture from the iCat condition is shown in Fig. 9.4.
These two basic animations were used to create animations for the different levels of acceleration and curvature by repositioning the keypoints and adjusting interpolation parameters of the functional animations. Afterwards, we calculated the values for acceleration and curvature separately for both embodiments as follows. For acceleration, we first approximated the first and second derivatives based on the motion trajectories from the editors. In the current version, the iCat robot updates the motor positions with 10 frames per second. Therefore, the velocity v of an actuator can be approx-

Figure 9.5: Sample interaction in the Roomba condition.

imated by the difference in position of two consecutive frames and in an analogous manner the acceleration a can be approximated as the difference of two consecutive velocities:

$$v = \dot{s} = \frac{s_{i+1} - s_i}{t} \tag{9.1}$$

$$a = \ddot{s} = \dot{v} = \frac{v_{i+1} - v_i}{t} \tag{9.2}$$

For an animation the average acceleration $\bar{a}$ was calculated over the number of frames F as:

$$\bar{a} = \frac{1}{F} \sum_{i=1}^{F} a_i \tag{9.3}$$

In the same manner also the average velocity of the Roomba robot was calculated, but with the difference that the velocity didn't have to be approximated, because the serial interface directly accepts a target velocity as a parameter, which was used for the calculation.

As a measurement for curvature we calculated the extrinsic curvature κ. If the radius is known, the curvature can directly be calculated by:

$$\kappa = \frac{1}{r} \tag{9.4}$$

For the Roomba robot we could directly apply this definition, because it always moves in circular segments. For the iCat robot, however, the radius of a line segment cannot directly be derived from the control signals. In order to calculate the curvature of the movements of iCat, we analyzed the path in space of the center of iCat's face as it moved the head to accomplish the task. The center is given by the tip of the nose and moves on an ellipsoid surface, which is defined by the radius for the pan and tilt axes. The shape of the ellipsoid is parameterized by the equatorial radii a, b and c along the axes of the coordinate system

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1 \tag{9.5}$$

The parameters are given by the iCat embodiment as $a = b = 10.5$cm and $c = 12.5$cm, assuming that z is the vertical axis and the x-y plane is parallel to the table. Furthermore, the control signal of the motors can be directly converted to viewing angles. The pan angle Φ is controlled by the value that is send to the actuator labeled 'body' and the tilt angle Θ is controlled by the actuator labeled 'neck'. The maximum viewing angles for Φ and Θ are in the range of $-45 \leq \Phi \leq 45$ and $-25 \leq \Theta \leq 25$, respectively.

$$\Phi = \text{body}\tfrac{45}{100} \tag{9.6}$$

$$\Theta = \tfrac{\text{neck}-50}{2} \tag{9.7}$$

With these parameters the three dimensional path is parameterized by

$$x = a \sin(\Phi) \cos(\Theta) \tag{9.8}$$

$$y = b \sin(\Phi) \sin(\Theta) \tag{9.9}$$

$$z = c \cos(\Phi) \tag{9.10}$$

The curvature of a parameterized curve in a three dimensional space is given by:

$$\kappa = \sqrt{\frac{(\dot{x}^2 + \dot{y}^2 + \dot{z}^2)(\ddot{x}^2 + \ddot{y}^2 + \ddot{z}^2) - (\dot{x}\ddot{x} + \dot{y}\ddot{y} + \dot{z}\ddot{z})^2}{(\dot{x}^2 + \dot{y}^2 + \dot{z}^2)^3}} \tag{9.11}$$

After designing the animations for the two embodiments, we calculated the actual values for curvature and acceleration to make sure that they indeed

Robot	mean acceleration			mean curvature		
Condition	low	medium	high	low	medium	high
iCat	0.33	0.81	1.19	0.46	0.56	1.41
Roomba	0.18	0.60	1.10	0.85	1.50	2.20

Table 9.1: Values of acceleration and curvature for the three levels low, medium and high.

represent three levels for the two motion characteristics. The resulting values are summarized in Table 9.1.

9.3.3 Participants

We recruited participants through the J. F. Schouten School participant database [73]. The database contains people of all age groups who are interested to participate in scientific research experiments. 42 healthy adult participants, aged 20-45 years, were randomly selected and recruited from this database. All participants were reimbursed for their participation in the experiment. Three participants participated in the initial pilot study. The other 39 participants, 12 female, 17 male, participated in the final experiments.

9.3.4 Procedure

The experiment took approximately 45 min and consisted out of three parts: 1) Intake ($\tilde{5}$ min), 2) Rating of stimuli ($\tilde{3}5$ min) and 3) Final interview ($\tilde{5}$min). When the participants entered the room, they were greeted by the experimenter who introduced himself and gave an outline of the activities during the experiment. After this introduction the participants were given an informed consent form to sign before the experiment started.
The participants were randomly assigned to see either the iCat or the Roomba embodiment first. Every robot performed nine animations, one for every combination of acceleration and curvature for the values of low, medium and high, respectively. The order of the nine animations was randomized. After every behavior the participants filled in the PANAS and SAM questionnaires. Most of the participants gave spontaneous comments in a think-out-loud fashion, which were also noted down. After the completion of all stimuli, a semi-structured interview was performed in which the participants were asked: 1) to give their general impressions on the behaviors, 2) to describe the differences and similarities of the behaviors,

3) to indicate a preference for behaviors and to elaborate why and 4) to compare the observed behaviors with behaviors they would expect from a commercial product. At last the participants received a small reimbursement according to the guidelines of the J.F. Shouten participant database.

9.4 Results

9.4.1 Missing values

One participant had to be excluded from the data, because she misunderstood the task and responded to all items of the PANAS and SAM questionnaires with the same value, arguing that a machine with wires does not have emotions. However, during the final interview she had a clear preference for certain behaviors of both robots. In fact she used the same emotional words to describe these behaviors as the other participants, i.e., "*... this one was too aggressive. It seemed to be very angry about something.*"(participant 26) or "*I liked this the most, because it seemed very happy and confident with the task*" (participant 26).
Some participants couldn't finish answering the questionnaires in the scheduled time. For this reason, we had to leave out some of the animations for eleven participants in order to minimize scheduling conflicts. All animations for the Roomba embodiment were included. A fixed number of animations for the iCat embodiment were excluded. The main reason for leaving out these iCat animations was to record all responses to the Roomba embodiment, which is presumably a more abstract stimulus than the iCat. In particular, the four animations in which one of the independent variables was set to medium were not shown. This allowed us to get a stable representation of the affective space at the borders of the motion parameter space, but reduced the resolution for in-between values. Using this procedure, for every missing value close neighbors were available in the data set.

9.4.2 Gender effects

First of all, we tested whether gender had an effect. We performed an analysis of variance with the sex as independent variable and tested if there is a significant effect on any combination of acceleration and curvature for the Roomba and the iCat condition. All combinations summed up to a total of 90 measurements (three levels for acceleration, three levels for curvature,

two levels for embodiment and five measurements for valence, arousal, dominance, positive affect and negative effect). Neither significant main effects nor significant interactions were found for gender. Hence gender could be excluded from the following measurements.

9.4.3 Perception of affect

In order to test whether the motion features had an impact on the perception of affect we performed a repeated measure analysis with the independent variables curvature c, acceleration a, embodiment e and the dependent variables valence (V), arousal (A), dominance (D), positive affect (PA) and negative affect (NA). As input we used only the levels of acceleration and curvature. We first only tested the 18 participants for which we had all data available. The mean and standard deviation for the measurements for Roomba are summarized in Table 9.2 and the according values for iCat are summarized in Table 9.3. The significance levels and partial eta square effect sizes are reported in Tables 9.5 and 9.6. Mauchly's test indicated that sphericity was for none of the cases violated, therefore degrees of freedom did not have to be corrected. After this first test we also repeated the test with all 38 participants, replacing the missing values with the means over the combination of stimulus and measurement. We found that none of the results changed significance. That is, the measurements that showed a significant difference for 18 participants also stayed significant with 38 participants and all non-significant values remained to be non-significant. In fact, the values changed only marginally. Therefore, we report in the following only on the cases where full data was available.

From Table 9.5 it can be seen that the embodiment had no main effect on the measurements, but that in general acceleration and curvature showed significant effects. However, there are a few exceptions. For example, the acceleration did not have a main effect on valence.

Comparing the results from the PANAS and SAM scales we found both to be similarly responsive to the manipulations of the independent variables. In our repeated measure design, we calculated for every experimental condition a correlation table for the five measurements, resulting in 18 tables with 5x5 entries. We calculated a mean correlation table by averaging over the factors. The mean correlation values are reported in Table 9.4. The highest absolute value was found between valence and positive affect. However, comparing the significance values, the PAD model indicated that acceleration has no effect on the perceived valence, which was not visible

acc.	cur.	valence	arousal	dominance	positive affect	negative affect
low	low	4.89/0.32	6.00/0.20	4.33/0.18	21.00/1.10	15.50/1.00
low	med.	4.11/0.29	6.06/0.30	4.11/0.24	20.61/1.35	14.50/0.85
low	high	5.44/0.28	5.17/0.32	3.94/0.24	21.56/1.57	17.72/0.89
med.	low	5.67/0.31	5.28/0.29	3.78/0.28	20.78/1.26	17.06/0.85
med.	med.	4.56/0.27	5.17/0.22	4.00/0.34	21.00/1.61	14.56/0.83
med.	high	3.89/0.25	4.33/0.23	4.11/0.27	24.06/1.89	16.00/1.15
high	low	5.39/0.37	3.11/0.20	5.22/0.29	24.44/1.57	19.50/1.14
high	med.	3.67/0.26	3.61/0.28	5.11/0.35	26.56/1.54	16.78/1.04
high	high	5.00/0.28	2.67/0.29	4.11/0.14	27.11/1.37	20.28/1.51
Average		4.73/0.29	4.60/0.26	4.30/0.26	23.01/1.47	16.88/1.03

Table 9.2: Mean and standard deviation for Roomba (N=18). Reported in the format: mean/std. dev. (acc. = acceleration, cur. = curvature)

acc.	cur.	valence	arousal	dominance	positive affect	negative affect
low	low	5.39/0.29	6.39/0.27	3.89/0.18	16.11/1.26	16.67/1.13
low	med.	3.83/0.34	5.72/0.21	4.17/0.19	22.89/1.18	15.17/0.70
low	high	5.06/0.30	5.61/0.29	3.94/0.25	18.61/2.03	14.89/1.08
med.	low	5.17/0.22	5.22/0.25	4.22/0.24	20.33/0.71	14.67/0.63
med.	med.	3.94/0.24	5.17/0.22	4.17/0.28	22.50/1.64	13.89/1.00
med.	high	4.39/0.29	3.56/0.29	4.11/0.36	26.89/1.04	17.28/1.00
high	low	4.44/0.37	3.28/0.21	5.44/0.35	28.67/1.01	13.94/1.00
high	med.	4.17/0.25	3.83/0.22	4.61/0.18	25.44/1.08	19.33/0.96
high	high	4.78/0.31	3.11/0.27	3.89/0.29	25.17/1.23	17.89/1.37
Average		4.57/0.29	4.65/0.25	4.27/0.26	22.96/1.24	15.97/0.99

Table 9.3: Mean and standard deviation for iCat (N=18). Reported in the format: mean/std. dev. (acc. = acceleration, cur. = curvature)

	valence	arousal	dominance	PA	NA
V	1.00	0.11	-0.17	-0.38	0.30
A	0.11	1.00	-0.10	-0.28	0.05
D	-0.17	-0.10	1.00	0.28	-0.19
PA	-0.38	-0.28	0.28	1.00	0.02
NA	0.30	0.05	-0.19	0.02	1.00

Table 9.4: Mean correlation values of the PAD and PA-NA space. (PA = positive affect, NA = negative affect)

when expressed in the positive and negative affect dimensions. Additionally, there was some discrepancy in the interaction between acceleration and curvature. While in the PAD space all dimensions showed a significant interaction, this could not be reported on the negative affect axis. Furthermore, in the PAD space there was no interaction between curvature and embodiment visible, but the negative affect measurement reported a value below the significance threshold of 0.05. A similar observation can be made when analyzing the three way interaction of acceleration, curvature and embodiment. Only along the valence axis this interaction was significant, while both, the positive and negative dimensions showed significance.

9.4.4 Relational of motion features to perceived affect

In order to estimate a relation between the motion and affective space, we performed linear regression analysis. As model parameters we used the linear and squared terms of acceleration and curvature as well as the linear interaction between those two. We did not include the embodiment, because it did not have a main effect in the previous analysis. Furthermore, we only analyzed a relationship dependent on the levels of acceleration and curvature in order to be able to compare between the different embodiments. From the resulting five parameters, we searched for the best predictors using a stepwise selection in a linear regression. That is, in every step the predictor that contributed the most to minimize the residual error was chosen. The results are summarized in Table 9.7. The first column gives the order of the predictors, the second column gives the quality of the approximation, i.e., how much of the variance is accounted for by the model the third column gives the corresponding ANOVA results for testing the approximation. First, from these values it can be seen that most of the information for perceived arousal is carried in the acceleration parameter of the motion. Secondly, even if all ANOVAs report significant results, only small percentages of the variance could be predicted with these simple models. This is most evident for the PA-NA space in which the variance of the negative affect dimension cannot conclusive be explained with the calculated models.
During the final interview, all but one participant reported that they had the impression that the robots clearly had different emotions or were in particular moods. We will summarize these subjective impressions in the following. All participants consistently attributed animacy and almost all participants perceived some type of personality. While some reported that they had the impression to face every time a completely new robot with a

	acceleration			curvature			embodiment		
	F	p	$_p.\eta^2$	F	p	$_p.\eta^2$	F	p	$_p.\eta^2$
V	.755	.478	.043	15.726	.000	.481	2.018	.174	.106
A	114.112	.000	.870	19.546	.000	.535	.230	.638	.013
D	11.255	.000	.398	4.687	.016	.216	.084	.776	.005
PA	28.061	.000	.623	4.336	.021	.203	.014	.909	.001
NA	9.457	.001	.357	3.857	.031	.031	3.871	.066	.185

Table 9.5: Significance values for the main effects of the independent variables on the measurements and partial η^2 effect sizes. (N=18)

	acc.*cur.			acc.*emb.			cur.*emb.			acc.*cur.*emb.		
	F	p	$_p.\eta^2$	F	p	$_p.\eta^2$	F	p	$_p.\eta^2$	F	p	$_p.\eta^2$
V	4.331	.004	.203	.146	.865	.009	.381	.686	.022	4.215	.004	.199
A	3.109	.021	.155	1.708	.196	.091	.704	.502	.040	1.708	.158	.091
D	4.157	.005	.196	.727	.491	.041	.173	.842	.010	1.019	.404	.057
PA	2.843	.031	.143	2.077	.141	.109	1.027	.369	.057	3.747	.008	.181
NA	1.990	.106	.105	1.033	.367	.057	4.005	.027	.191	5.202	.001	.234

Table 9.6: Significance values for the interaction effect of the independent variables on the measurements and and partial η^2 effect sizes. (N=18)

different personality, others had the impression of observing the same character in different moods. When asked to describe the observed behaviors, all participants used emotional adjectives to describe the robots' behavior, e.g., "... this one was a little moody. It seemed to be not very happy with what he was doing."(participant 6). Interestingly, some participants attributed male and female characteristics depending on the current condition and addressed the robots either with "he" or with "she". It was our impression that high levels of curvature were associated with female characteristics while high levels of acceleration were associated with male characteristics. When asked for a preference, some participants preferred seemingly neutral behaviors with medium levels of acceleration and arousal, while others liked the expressiveness of the robots in the high and low conditions. Especially the younger participants liked more the expressive behaviors, while older participants stated that neutral behaviors are more appropriate for a robotic device. However, almost all participants stated that they rather would buy a robot that is friendly and helpful, instead of a robot that is very aggressive or moody.

	predictors	R^2	ANOVA
V	(1) $a * c$	0.10	F(1,616)=7.26, $p = 0.07$
A	(1) a^2(2) c^2(3) $a * c$	0.471	F(3,614)=184.03, $p < 0.001$
D	(1) a^2(2) $a * c$(3) c	0.10	F(3,614)=23.86, $p < 0.001$
PA	(1) a^2(2) c	0.106	F(2,615)=37.70, $p < 0.001$
NA	(1) a^2(2) a(3) c^2(4) c	0.063	F(4,613)=11.41, $p < 0.001$

Table 9.7: Stepwise linear regression results

9.5 Discussion

The results show that acceleration is correlated with the perceived arousal. However, no direct relationship between acceleration or curvature and valence could be found. A significant interaction between acceleration and curvature was found, suggesting that these parameters are not perceived independently from each other. Even though the dimensions of acceleration and curvature are independent in movement space and the dimensions valance, arousal and dominance are independent in affect space, they interfere in the cognitive process that transforms between the two spaces. For example, in the high curvature condition of Roomba, it depended on the level of acceleration how the behavior is interpreted. In the low acceleration condition participants interpreted the behavior as "careful"(participant 27), "moving like a cat that wants attention"(participant 6), "not determined wandering around" (participant 9). In contrast, for the same value of curvature, participants interpreted the behavior for a high level of acceleration as "stressed"(participant 4), "aggressive; guarding an area"(participant 25) or even "very proud, exhibiting a macho kind of behavior" (participant 7). From Table 9.6 it can also be seen that the way the two dimensions interact with each other is influenced by the embodiment. The same manipulation from low to high acceleration in the iCat condition, keeping the curvature constant, resulted in a change from "falling asleep" (participant 4) and "calm and relaxed" (participant 19) to "nervously searching" (participant 16) and "very chaotic and unorganized" (participant 6).

Pollick et al. estimated that most of the information on the valence axis is encoded in the frequency relations between limb movements [192]. However, in our experiment the Roomba robot did not possess limbs, but participants were still able to perceive different levels of affect. Our results suggest that the valence information is at least partly encoded in the interaction between acceleration and curvature. However, this model did not explain

a sufficient amount of the variance to be conclusive. Analysis of further motion features and models is required to isolate the valence information from motion signals.

It remains to be tested if discrepancies between the PAD space and PA-NA space can be explained by being a rotational variant of the same space. For example, Mehrabian claimed that the PANAS model lacks validity, because it does not capture certain aspects of the affective space [164]. When analyzing dance moments, Pollick et al. stated that activity or arousal could be predicted from the motion parameters, but no direct relation between movement parameters and valence was found [192]. Our results give evidence that the perceived level of arousal is highly correlated with the level of acceleration. However, there seems to be no independent model that connects curvature to valence, but that most of the information about valence is carried in the interaction between acceleration and curvature. Furthermore, some participants reported that the words on the PANAS scale did not really match their impression of the robot. They reported to miss words such as "happy", "tiered", "moody", "confused" or "disinterested" and therefore rated the other items lower, because they did not seem them fit their impression. A specialized questionnaire would have to be developed that captures possible interpretations of motion patterns better than the PANAS. In contrast there were no problems with the SAM scale.

Some people reported that at the beginning they were confused, because iCat kept a neutral face during the conditions. Therefore, they perceived a mismatch between expression in the motion and in iCat's mechanical face. However, they also stated that they quickly got used to it, interpreting that iCat did not want to reveal her true emotions but kept a Poker-face. In general, there was no clear preference for an embodiment. Some reported it was easier to "see" emotions in iCat, others reported that the task was easier for the Roomba condition.

During the interview, all participants articulated a clear preference for certain types of behavior. Based on these responses we can recommend to avoid combinations of high level or low levels for curvature and arousal, because generally these conditions were most disliked. However, participants liked the expressiveness of the character when low levels in one dimension were combined with high levels of the other dimensions. In these cases the robot was perceived as most emotional, for example very active or aggressive. Medium levels for both conditions gave a rather neutral impression that was generally accepted.

9.6 Conclusions

In this study we investigated the relation between robot motion and the perceived affective state of the robot. From literature we derived two motion characteristics that seemed to be most influential for the perceived affective state, namely acceleration and curvature. We systematically varied both conditions and tested the perceived affect with two embodiments. For assessing affect we selected the PANAS and SAM scales, which are supposed to be rotational variants of the same space. With our variations we were able to trigger the perception of different emotions.
We found that both parameters, acceleration and curvature, have a significant effect on the perceived affective state. However, there were slight differences between the two emotional models that were difficult to explain by being a rotational variant. In general, we found the SAM to be more appropriate, because all participants were able to report their general impression according to the pictographic representation of the self-assessment manikins.
Furthermore, we found that the embodiment had no significant main effect on the perceived affective state, stressing the importance for carefully designed robot behaviors. Analyzing the relationship in more detail, we found that acceleration carries most of the information for perceived arousal. However, no such simple relationship could be found for the dimensions of valence and dominance or for the dimensions of positive affect and negative affect. Our results indicate that the information for valence is at least partly carried by a linear relation between curvature and arousal.
From these results we can derive design knowledge for the design of movement behaviors of social robotic interfaces. If the designer wants to convey different levels of arousal he can adjust the acceleration parameter of the animation accordingly. Motion can therefore be used as a design modality to induce a desired perception. However, further research is needed to investigate such a model for valence. Especially if a designer intends to convey a positive or negative emotion it has to be analyzed what motion features carry this valence information in order to be able to predict user responses.

Chapter 10

Discussion

This technological design has analyzed the design process of applications for social robotic interfaces. Social interaction technology facilitates seamless user interaction, by utilizing people's natural interaction capabilities to interact with other social communication partners.
Socially interactive interfaces offer a range of opportunities. For example, social interactive interfaces might serve as enabler for a variety of products that require human-machine cooperation. A potential advantage could be that people tend to be more forgiving when the system doesn't behave as they expected. During a pilot for the robot motion experiment (see Chapter 9) Roomba got stuck at a table. The participant was immediately willing to pick it up and place it at a free spot where it could continue. Typical reactions to a failure of a technical device such as a mobile phone are frustration and anger. Instead, users appeared to feel empathy when dealing with a social interface. Several researchers in the field of human-robot interaction have made similar observations [9, 214]. This phenomenon can help to enable applications in which user support is required. Especially in the field of home robotics, technology is not yet mature enough to handle all situations that can occur in unstructured environments.

10.1 Design challenges

The design of social interactive interfaces poses new design challenges on the designer. In comparison to traditional interfaces, robotic user interfaces do not only provide a set of interface elements that offer the control over a given set of functions, but the sum of all elements is perceived as one entity. That is, these interface elements induce the perception of dealing

with a character. However, already subtle flaws in the design can destroy this perception, which has a negative effect on the interaction, potentially leading to frustration and a loss of trust of the user.

Throughout this work, design challenges for the application design for social robotic interfaces have been analyzed at three levels: 1) application design, 2) interaction design and 3) behavior design level.

Analyzing the design challenge at application design level, it was shown that incorporating social interaction technology in the interface design does not necessarily enhance interaction. The challenge is to find applications which benefit from social interaction, but take limitations of currently available technology into account. To support the designer, a balancing framework has been proposed that requires to balance between the four elements of 1) application, 2) user, 3) interface and 4) technology. Multiple applications have been analyzed. It was shown that carefully choosing a restricted application domain is a crucial requirement for applying social interaction technology that is often overlooked.

Analyzing the design challenges at the second, interaction design level, it was found that social robotic interfaces provide means to offer complex control over device autonomy. Most notably, social interaction technology has the potential to provide interface artifacts that fall in the category of programming. For example, in order to instruct a robotic vacuum cleaner to execute a customized cleaning plan, the user essentially needs to program the device's behavior. In traditional interface design, a trade-off between level of control and ease-of-use has to be made. Traditional configuration interfaces only allow to select from a predefined set of options. Programming interfaces, on the other hand, allow to define new device behavior. The challenge for the designer is to find an interface metaphor that is familiar to the user, but allows programming. One option that is offered in the context of social interaction is a learning and teaching metaphor [232]. If the users perceives a device as a character, the user most commonly also attributes human-like capabilities. However, there are not necessarily all human capabilities such as natural language or symbolic reasoning required. The example of training a pet shows that knowledge transfer can be realized on a very basic level of communication. A promising approach in this context is to teach by demonstration. The important point from a design perspective is to maintain the impression of a life-like character.

Inducing this perception has been addressed at behavior design level. In this context, the concept of believability [145, 201, 158, 17, 112, 8] becomes important. For example, the work of Reeves and Nass [198] has shown

that people apply social interaction rules even if they are consciously aware of dealing with a machine. This phenomenon can also be observed in the field of movie and film animation. Even though the audience is consciously aware that the characters on the screen (e.g., cartoon characters) do not really exist, they attribute a rich set of life-like characteristics including desires and emotions.
However, there seems to exist a fine line between which attributes are perceived as life-like and which induce the perception of a dull machine. The field of animation has shown that the illusion of life can be destroyed already by subtle flaws in the animation [231, 253]. In game character design, one of the most challenging behaviors to design is idle behavior [113]. This fine line makes it increasingly challenging to create the appropriate interface for the user. Throughout this thesis it has been argued that animation technology provides adequate means for designing believable behaviors for social robotic interfaces.

10.2 Animation Technology

Believability is one the core requirements for the design of an interactive character [17]. The user will only engage in social natural interaction if he receives appropriate feedback from the device. For example, for many users it would be awkward to start talking to a VCR. However, if the interface presents itself as a character, people immediately attribute life-like attributes and start using their natural interaction capabilities [161]. Furthermore, the degree of socialness has an effect on the effectiveness of the interaction [211].
Design knowledge for the design of social robotic user interfaces can be derived from the field of animation. The movie and animation industry have impressively demonstrated that expressive behaviors can evoke emotional responses, even without special expressive degrees of freedom [231]. Film characters can engage the audience over the entire length of a movie. However, there are several important differences between movie or game character animation and robotic character animation. First of all, in a movie script the whole scene is predefined. For example, a cartoon designer has control over camera perspective, characters in the scene and their interactions down to a pixel level. This level of control is not possible for robotic applications as they need to react dynamically to the environment. For game characters, this has mainly been solved by defining a set of basic animations and modifying them using signal processing techniques

[30, 85, 113]. Also scripting technology and automatic equation solvers have been used in case of well defined application domains [186, 84]. However, for game character design the actions of the user are limited by the controls of the game. All information about the environment are digitally accessible to game characters. This rich source of information is not available for physical robots, but the advantage is that users can physically interact with the device.
In order to facilitate the behavior design process, three basic design dimensions of robotic behavior design have been identified: 1) Naturalness, 2) Adequateness, and 3) Development over time. For adhering to these design dimensions, new tools are required that support the development process. This thesis proposed functional animations as a step to close this gap. Functional animations generalize traditional animations by adding the dynamic power of describing animations by equations and combine the expressive power of keyframe animation techniques with the dynamic control of scripted behaviors. For example, using keyframe animations, the designer has direct control over a particular expression, but lacks high level control to adapt the animation to a particular context. Scripted animations offer highly dynamic behaviors but are limited in expressivity. Functional animations offer the possibility to adjust an animation in a formally defined way using parameter models. An important difference to motion editing methods is that the allowed adjustments of an animation can be parameterized. Motion editing techniques such as blending often destroy an internal structure of the animation and produce behaviors that are not believable or even impossible to execute for a given embodiment [84, 253].
Furthermore, functional animations support the design process by defining an interface between trajectory designers and interaction designers. Using parameter models, animation designers can define a parameter space that is allowed to modify the animation. For example, they can define a parameter space to let the same animation appear happy or sad. Well defined interfaces are especially important for development of more complex applications in which designers of multiple disciplines need to collaborate. These requirements have been reflected in the overall design of a social robotic development framework.

10.3 Requirements

The results from the analysis of the design challenges have been used to derive a set of requirements for a software framework that supports the

development process of social robotic interfaces. Requirements have been collected in an iterative design process in which basic features have been prototyped, tested and refined. Another important source of input has been the OPPR user community [188] and published results of user studies with the iCat research platform.
Based on these sources and general design knowledge derived from literature, four major sources of requirements have been defined: 1) Application designer requirements, 2) Development process requirements, 3) Domain requirements, and 4) Framework requirements. The requirements in these categories served as the main guideline for the implementation of the SRD architecture. Additionally, they denote design knowledge that was gained throughout this design project. In future iterations, the list of requirements might be supplemented with additional demands as the architecture increases in level of abstraction. For example, one envisioned feature for one of the next generations is to extend the framework with an emotion design tool. In fact, one of the core qualities of the framework is to be sufficiently flexible to accommodate changing requirements. This has been achieved by identifying major building blocks and architectural decoupling to restrict changes to a single software component.

10.4 Architecture

Based on the requirements, two different environments have been introduced, a 1) Development-Environment and an 2) Execution-Environment for the Social Robot Development architecture. A similar approach has been followed for example by the Lego® Mindstorms platform as it allows to develop an application and later execute it on the Lego-Brick independently from the development environment. However, the Lego® architecture does not allow to directly test application artifacts. In other development environments for robotic applications, these two cases are intertwined [168], which makes it very difficult to maintain development and deployment independently from each other.
The separation introduced for the SRD architecture, has several advantages. First of all, it allows different communication models. For the Development-Environment a synchronous communication model was chosen, which responds to the designer requirement of being able to fully test and verify correctness of the application logic as it allows deterministic execution of the program. For the Execution-Environment, this is not a feasible communication structure, due to the hardware setup of many systems and

real-time constraints. However, a synchronization index has been defined that indicates the difference between a synchronous and asynchronous execution. In concrete situations, this index can be used as quality criteria to insure correct program execution.
Another quality attribute of the proposed architecture is that several design patterns have been applied throughout the design process. These design patterns impact mainly the non-functional attributes of the architecture. Flexibility is one of the main non-functional attributes that must be guaranteed. In the SRD architecture, flexibility has been achieved by modularization and separation of independent software components. However, as soon as social robot applications become available for customers, security might become the most important requirement. During the evaluation of the non-functional requirements, it has been shown that the SRD architecture is already prepared to satisfy basic security needs, but these are still missing in the current implementation.
In the course of the architecture design, several editors have been defined to support the design task on several levels of abstraction. The tools have been successfully applied in two case studies. They offer not only operators to design application artifacts, but also define interfaces to communicate between different design disciplines.

10.5 Case studies

The software tools of the SRD architecture have been used in two case studies. In the first study, a tutoring application was developed to test the effects of social supportive behavior on the learning performance in a tutoring scenario. The conclusions and the research goals of the case study are not part of the documentation of this technological design, but it was an opportunity to see the software architecture in action. Roughly speaking, the case study confirmed that the design tools could be used to vary the degree of social behavior of the robot.
In the second case study, the software tools were used to investigate people's perception of robot motion in terms of perceived emotions. The functional animations were used to vary systematically the motion of two robots with different embodiments. The results of the experiment showed that the motion patterns have an influence on how the robot is perceived. In particular, the acceleration parameter of the motion appears to be directly related to the perceived emotion. No such relationship could be found to explain the perceived valence. One important finding in terms of design knowledge for

social robotic interfaces was that no main effect between the two embodiments could be found. This result suggests that the motion design can be generalized across embodiments. This means that design tools can be created that transcends a particular embodiment. However, more experiments are necessary to identify relevant motion features to be able to predict the emotion that is perceived from a motion feature.

10.6 Options for future research

One of the biggest challenges for social robotic interfaces is to find applications in which the interface is balanced with the application. Designing for social interaction does not necessarily mean to engage in social relationships, but to behave appropriately in social environments and communicate using naturally understandable modalities. Many promising application domains have been proposed, including tutoring, game buddy [60, 143] and general interface to ambient intelligent environments [9]. However, none of the applications has yet reached a critical mass and wide acceptance. Instead, only few social signs are incorporated in interface design, such as the Apple^{TM} desktop login screen that indicates a shaking of a head to signal that access was denied.

Currently, the biggest problem is the adequateness design dimension, because adequateness requires to appropriately react to a given situation. Technology for correctly assessing a social situation or emotions of the interaction partner are still immature. Furthermore, interaction rules needs to be defined that match a given situation.

Further research is required to extend available interaction design knowledge. This knowledge can in turn be used to create even higher level editors, such as an emotion editor. This level of abstraction would allow a designer to modify a basic action such as “look at the object” with an emotional parameter such as “look at the object angrily”. Such high level of abstractions are needed for creating believable characters.

Chapter 11

Conclusions

This thesis has investigated the design process of creating social interactive interfaces using interface robots. Social interaction technology provides a powerful interface metaphor to ease the interaction with technological devices. It has been shown that, for enabling social interaction, application designers face a new set of design challenges and requirements. Most notably, social interfaces need to create and maintain the impression of a believable character.
Four research questions have been addressed:

① **Design challenges:** What are the design challenges in the process to develop an application for a social robotic interface?

② **Tools:** How can the design process be supported by design tools?

③ **Requirements:** What are the requirements for a software architecture that unifies the design process?

④ **Architecture:** What architecture fulfills the requirements for a robot application design framework?

To answer the first question, a top-down analysis was performed and the design challenges were investigated on application, interaction and behavior design level. The analysis on application design level has shown that not every application benefits from social interaction technology. A balancing framework was conceived that supports the design process and serves as a predictor for the success of an application in an early design phase.
On the next level, the design process of social interactive interfaces has been analyzed with regards to the interaction design. It was found that

current interface technology can be grouped in two classes, configuration and programming. Most of the current interfaces only use elements of the configuration class. However, these interfaces do not scale up and fail to provide high level control over complex autonomous devices. Instead, programming interfaces must be used.

On the behavior design level, the design process has been analyzed with regards to behavior generation of the interface character. Three basic design dimensions have been identified: 1) Naturalness, 2) Adequateness and 3) Development over time. Existing animation technology was found to provide an adequate abstraction for satisfying the first dimension. However, for the other two dimensions, interactivity had to be added. This demand has been solved by introducing the functional animation principle. A particular advantage of functional animation is that they combine the expressiveness of keyframed animations with the flexibility of scripted behaviors.

In order to answer the second research question, an iterative design approach was adhered throughout this thesis for the development of application design tools. Multiple tools have been developed, including the functional animations editors to animate the iCat robot and the Roomba robot and an interaction design editor. It was found that the design task can best be supported by providing specialized tools for the different levels of abstraction. A posture editor provided an adequate level of abstraction to control the iCat robot. For the Roomba robot, it was demonstrated that a path editor on the level of path segments was a more appropriate level of abstraction. For both editors, the functional animation principle was applied, which provided high level, parameterized control over expressive keyframe animation. On the level of interaction design, a specialized domain language was provided, which allowed high level commands such as 'look-at-user' or 'wait-for-reply'.

To answer the third research question, the requirements for a unified software architecture that supports the design process have been analyzed. Four basic categories of requirements have been identified: 1) Application designer requirements 2) Development process requirements, 3) Domain requirements and 4) Framework requirements. Several requirements of every domain have been motivated and subsequently addressed in the design of unified software architecture, which answers the last research question. A separate Development-Environment and Execution-Environment have been introduced, which allowed a modular decomposition with different communication models. It has been motivated that synchronous

communication is appropriate for the Development-Environment, while the Execution-Environment benefits from asynchronous communication. Both environments share software components that are related to the construct of an application.
The architecture has been evaluated with regards to functional and non-functional requirements. For functional requirements, traces have been analyzed that map the individual requirements on the architecture. The non-functional requirements have been evaluated using a scenario based architecture evaluation, based on the established ATAM and SBAR evaluation methods. Additionally, two case studies have been performed, one of which is reported in Chapter 9. In this case study, the design tools have been used to control the behavior of two robot embodiments in order to investigate how different motion patterns are perceived by a user. The results revealed a relationship between perceived arousal and acceleration parameters of the behavior. No such direct relationship could be found that conclusively explains the perceived valence. Instead, it seems that the interaction of acceleration and curvature encodes parts of the valence information. Further research is necessary to isolate the features that elicit the perception of different degrees of valence in order to create a model that allows automatic high level emotional direction of a character. Emotional direction provides a high level of abstraction that in turn can be integrated in the presented design framework to develop a next generation of social robotic interfaces.

References

[1] R. K. Abbott and H. García-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, 1992.

[2] M. Aicardi, G. Casalino, A. Bicchi, and A. Balestrino. Closed loop steering of unicycle like vehicles via Lyapunov techniques. *Robotics & Automation Magazine, IEEE*, 2(1):27–35, 1995.

[3] A. L. Ananda, B. H. Tay, and E. K. Koh. A survey of asynchronous remote procedure calls. *SIGOPS Operating System Review*, 26(2):92–109, 1992.

[4] Autodesk. Maya 8.5. http://usa.autodesk.com, 2007.

[5] J. Baeten and W. Weijland. *Process algebra*. Cambridge University Press New York, 1990.

[6] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.

[7] L. F. Barrett, B. Mesquita, K. N. Ochsner, and J. J. Gross. The experience of emotion. In *Anual Review of Psychology*, volume 58, pages 373–403, 2007.

[8] C. Bartneck. How convincing is Mr. Data's smile: Affective expressions of machines. *User Modeling and User-Adapted Interaction*, 11(4):279–295, 2001.

[9] C. Bartneck. *eMuu - An Embodied Emotional Character for the Ambient Intelligent Home*. PhD thesis, Technische Universiteit Eindhoven, 2002.

[10] C. Bartneck and J. Forlizzi. A design-centred framework for social human-robot interaction. In *13th IEEE International Workshop on Robot and Human Interactive Communication, 2004. ROMAN 2004*, pages 591–594, 2004.

[11] C. Bartneck and J. Forlizzi. Shaping human-robot interaction: Understanding the social aspects of intelligent robotic products. In *CHI '04 extended abstracts on Human factors in computing systems*, pages 1731–1732. ACM Press, 2004.

[12] C. Bartneck and H. Jun. Rapid prototyping for interactive robots. In *Proceedings of the 8th Conference on Intelligent Autonomous Systems (IAS-8)*, pages 136–145, 2004.

[13] C. Bartneck, T. Kanda, H. Ishiguro, and N. Hagita. Is the Uncanny Valley an Uncanny Cliff? In *16th IEEE International Conference on Robot and Human Interactive Communication ROMAN*, pages 368–373. IEEE, 2007.

[14] C. Bartneck, T. Kanda, O. Mubin, and A. A. Mahmud. Does the design of a robot influence its animacy and perceived intelligence? *International Journal of Social Robotics*, 1(2):195–204, 2009.

[15] C. Bartneck, J. Reichenbach, and A. van Breemen. In your face, robot! The influence of a character's embodiment on how users perceive its emotional expressions. In *Proceedings of the Design and Emotion Conference, Ankara, Turkey,*, pages 32–51, 2004.

[16] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[17] J. Bates. The role of emotion in believable agents. *Communications of the ACM*, 37(7):122–125, 1994.

[18] D. Benyon, P. Turner, and S. Turner. *Designing interactive systems*. Addison-Wesley New York, 2004.

[19] J. Bergstra and J. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[20] J. Bergstra, J. Klop, and J. Tucker. Process algebra with asynchronous communication mechanisms. In *Seminar on Concurrency: Carnegie-Mellon University Pittsburgh, PA, July 9-11, 1984*, page 76. Springer, 1985.

[21] C. Bethel and R. Murphy. Survey of Non-facial/Non-verbal Affective Expressions for Appearance-Constrained Robots. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 38(1):83–92, 2008.

[22] P. W. Blythe, P. M. Todd, and G. F. Miller. *Simple Heuristics That Make Us Smart*, chapter How Motion Reveals Intention: Categorizing Social Interactions, pages 257–285. Oxford University Press, 1999.

[23] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[24] R. T. Boone and J. G. Cunningham. Children's decoding of emotion in expressive body movement: The development of cue attunement. *Developmental Psychology*, 34(5):1007–1016, 1998.

[25] J. Borenstein and L. Feng. Measurement and correction of systematic odometry errors in mobile robots. *Robotics and Automation, IEEE Transactions on*, 12(6):869–880, 1996.

[26] M. Bradley and P. Lang. Measuring emotion: the Self-Assessment Manikin and the Semantic Differential. *Journal of Behavior Therapy and Experimental Psychiatry*, 25(1):49–59, 1994.

[27] C. Breazeal. Regulation and entrainment in human-robot interaction. In *ISER '00: Experimental Robotics VII*, pages 61–70. Springer-Verlag, 2001.

[28] C. Breazeal. *Designing Sociable Robots*. MIT Press, 2002.

[29] R. A. Brooks. *Architectures for Intelligence*, chapter How to build complete creatures rather than isolated cognitive simulators, pages 225–239. Lawrence Erlbaum Assosiates, Hillsdale, NJ, 1991.

[30] A. Bruderlin and L. Williams. Motion signal processing. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 97–104. ACM Press, 1995.

[31] A. Camurri, I. Lagerlöf, and G. Volpe. Recognizing emotion from dance movement: Comparison of spectator recognition and automated techniques. *International Journal of Human Computer Studies*, 59(1-2):213–225, 2003.

[32] J. Casper and R. Murphy. Human-robot interactions during the robot-assisted urban search and rescue response at the world trade center. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 33(3):367–385, 2003.

[33] D. Chen and J. Yang. *Advances in Neural Networks - ISNN 2005*, volume 3498/2005 of *Lecture Notes in Computer Science*, chapter Stability Analysis and Performance Evaluation of an Adaptive Neural Controller, pages 42–47. Springer Berlin, 2005.

[34] J. Clark, S. DeRose, et al. XML path language (XPath) version 1.0. *W3C Recommendation*, 16, 1999.

[35] A. Clay, N. Couture, and L. Nigay. Emotion capture based on body postures and movements. *ArXiv e-prints*, 710, 2007.

[36] P. Clements, R. Kazman, and M. Klein. *Evaluating software architectures: Methods and Case Studies*. Addison-Wesley, 2006.

[37] G. Cockton. Revisiting usability's three key principles. In *CHI '08 extended abstracts on Human factors in computing systems*, pages 2473–2484. ACM Press, 2008.

[38] R. Cowie, E. Douglas-Cowie, N. Tsapatsoulis, G. Votsis, S. Kollias, W. Fellenz, and J. Taylor. Emotion recognition in human-computer interaction. *IEEE Signal processing magazine*, 18(1):32–80, 2001.

[39] J. R. Crawford and J. D. Henry. The Positve and Negative Affect Schedule (PANAS): Construct validity, measurement properties and normative data in a large non-clinical sample. *British Journal of Clinical Psychology*, 43:245–265, 2004.

[40] J. Crowley. Situation models for observing human activity. *ACM Queue Magazine*, 2006.

[41] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complexdistributed systems. In *Software Engineering, 1998. Proceedings of the 1998 International Conference on*, pages 261–270, 1998.

[42] M. Cummings and S. Guerlain. Developing operator capacity estimates for supervisory control of autonomous vehicles. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 49(1):1–15, 2007.

[43] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathmatics of Control, Signals, and Systems*, 2(4):303–314, 1998.

[44] V. Dasser, I. Ulbaek, and D. Premack. The perception of intention. *Science*, 243(4889):365–367, 1989.

[45] K. Dautenhahn. I could be you: The phenomenological dimension of social understanding. *Cybernetics and Systems*, 25(8):417–453, 1997.

[46] K. Dautenhahn. The art of designing socially intelligent agents – Science, fiction and the human in the loop. *Applied Artificial Intelligence*, 12(8–9):573–617, 1998.

[47] K. Dautenhahn. Socially intelligent robots: Dimensions of human-robot interaction. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 362(1480):679–704, 2007.

[48] S. Dekker and M. Lützhöft. Correspondence, cognition and sensemaking: A radical empiricist view of situation awareness. *A cognitive approach to situation awareness: Theory and application*, pages 22–41, 2004.

[49] S. Denham, T. Wyatt, H. Bassett, D. Echeverria, and S. Knox. Assessing social-emotional development in children from a longitudinal perspective. *Journal of Epidemiology and Community Health*, 63(Suppl 1):i37–i52, 2009.

[50] P. Desmet. *Funology: from usability to enjoyment*, chapter Measuring Emotions: Development and application of an instrument to measure emotional responses to products, pages 111–123. Kluwer Academic Publishers, 2004.

[51] A. Dey and G. Abowd. The context toolkit: Aiding the development of context-aware applications. In *Workshop on Software Engineering for wearable and pervasive computing*, 2000.

[52] C. F. DiSalvo, F. Gemperle, J. Forlizzi, and S. Kiesler. All robots are not created equal: The design and perception of humanoid robot heads. In *DIS '02: Proceedings of the conference on designing interactive systems*, pages 321–326. ACM Press, 2002.

[53] W. Dittrich and S. Lea. Visual perception of intentional motion. *Perception*, 23(3):253–268, 1994.

[54] L. Dobrica and E. Niemelä. A survey on software architecture analysis methods. *Software Engineering, IEEE Transactions on*, 28(7):638–653, 2002.

[55] N. Doh, H. Choset, and W. K. Chung. Accurate relative localization using odometry. *Robotics and Automation. Proceedings. ICRA '03. IEEE International Conference on*, 2:1606–1612, 2003.

[56] K. Dorer. The magmaFreiburg Soccer Team. *Lecture notes in computer science: RoboCup-99 Robot Soccer World Cup III*, 3:600–603, 1999.

[57] E. Driscoll. Will Lucky the Dinosaur transform Disneyland into Jurassic Park? *Servo Magazine*, pages 8–12, 2004.

[58] D. C. Dryer and L. M. Horowitz. When do opposites attract? interpersonal complementarity versus similarity. *Journal of Personality and Social Psychology*, 72(3):592–603, 1997.

[59] B. R. Duffy. Anthropomorphism and the social robot. *Robotics and Autonomous Systems*, 42(3–4):177–190, 2003.

[60] B. Eggen, L. Feijs, M. Graaf, and P. Peters. Breaking the flow-intervention in computer game play through physical and on-screen interaction. In *The 'Level Up' Digital Games Research Conference,Copier, M. and Raessens, J.(editors) Uetrecht Univeristy*, 2003.

[61] P. Ekman and W. Friesen. *Unmasking the face: A guide to recognizing emotions from facial clues*. Prentice-Hall, Englewood Cliffs, NJ, 1975.

[62] A. I. Eliazar and R. Parr. Learning probabilistic motion models for mobile robots. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, page 32. ACM, 2004.

[63] M. Endsley. Design and evaluation for situation awareness enhancement. In *Design and evaluation for situation awareness enhancement*, volume 32, pages 97–101, 1988.

[64] M. Endsley. Measurement of situation awareness in dynamic systems. *Human Factors*, 37(1):65–84, 1995.

[65] M. Endsley. Toward a theory of situation awareness in dynamic systems: Situation awareness. *Human factors*, 37(1):32–64, 1995.

[66] M. Endsley, B. Bolte, and D. Jones. *Designing for situation awareness: An approach to user-centered design*. Taylor & Francis, 2003.

[67] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[68] European i2010 initiative on e-Inclusion. To be part of the information society. Technical Report COM(2007) 694 final; [SEC(2007) 1469], Commision of the European Communities, 2007.

[69] L. M. G. Feijs and Y. Qian. Component algebra. *Science of Computer Programming*, 42(2-3):173–228, 2002.

[70] A. D. Few, C. M. Roman, D. J. Bruemmer, and W. D. Smart. What does it do? : HRI Studies with the general public. In *16th IEEE International Conference on Robot and Human Interactive Communication ROMAN*, pages 744–749. IEEE, 2007.

[71] R. Fierro and F. Lewis. Control of a nonholonomic mobile robot using neural networks. *Neural Networks, IEEE Transactions on*, 9(4):589–600, 1998.

[72] T. Fong, I. Nourbakhsh, and K. Dautenhahn. A survey of socially interactive robots. *Robotics and Autonomous Systems*, 42(3-4):143–166, 2003.

[73] J. S. S. for User System Interaction Research. Participant database: http://ppdb.tm.tue.nl/. online, 2009.

[74] J. Forlizzi and C. DiSalvo. Service robots in the domestic environment: A study of the roomba vacuum in the home. In *HRI '06: Proceedings of the 1st ACM SIGCHI/SIGART conference on human-robot interaction*, pages 258–265. ACM, 2006.

[75] R. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-driven development using uml 2.0: promises and pitfalls. *Computer*, 39(2):59–66, 2006.

[76] N. Frijda. *The Emotions*. Cambridge University Press., 1986.

[77] K. Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural Networks*, 2(3):183–192, 1989.

[78] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software (34th printing Mar. 2007)*. Addison-Wesley Reading, MA, 1995.

[79] F. García-Córdova, A. Guerrero-González, and F. Marín-García. *Computational and Ambient Intelligence*, volume 4507/2007 of *Lecture Notes in Computer Science*, chapter Neuronal Architecture for Reactive and Adaptive Navigation of a Mobile Robot, pages 830–838. Springer Berlin / Heidelberg, 2007.

[80] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, 1:1–40, 1993.

[81] B. Gates. A robot in every home. *Scientific American*, Jan/2007, 2006.

[82] V. Gaur and B. Scassellati. Which motion features induce the perception of animacy? In *International Conference for Developmental Learning, Bloomington, Indiana*, 2006.

[83] F. Gee, W. Browne, and K. Kawamura. Uncanny valley revisited. In *Robot and Human Interactive Communication, 2005. ROMAN 2005. IEEE International Workshop on*, pages 151–157, 2005.

[84] M. Gleicher. Motion editing with spacetime constraints. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 139–149. ACM Press, 1997.

[85] M. Gleicher. Retargetting motion to new characters. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 33–42. ACM Press, 1998.

[86] M. Gleicher. Comparing constraint-based motion editing methods. *Graph. Models*, 63(2):107–134, 2001.

[87] R. Gockley, A. Bruce, J. Forlizzi, M. Michalowski, A. Mundell, S. Rosenthal, B. Sellner, R. Simmons, K. Snipes, A. Schultz, and J. Wang. Designing robots for long-term social interaction. In *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 1338–1343. IEEE, 2005.

[88] J. Goetz, S. Kiesler, and A. Powers. Matching robot appearance and behavior to tasks to improve human-robot cooperation. In *Robot and Human Interactive Communication, 2003. Proceedings. ROMAN 2003. The 12th IEEE International Workshop on*, pages 55–60. IEEE, 2003.

[89] M. A. Goodrich and A. C. Schultz. Human-robot interaction: A survey. *Found. Trends Hum.-Comput. Interact.*, 1(3):203–275, 2007.

[90] J. D. Gould and C. Lewis. Designing for usability: Key principles and what designers think. *Communications of ACM*, 28(3):300–311, 1985.

[91] P. C. M. Govers, P. Hekkert, and J. P. L. Schoormans. *Design and emotion. The design of everyday things*. CRC Press, 2002.

[92] B. Graf, C. Parlitz, and M. Hägele. Robotic Home Assistant Care-O-bot® 3 Product Vision and Innovation Platform. In *Human-Computer Interaction. Novel Interaction Methods and Techniques: 13th International Conference, HCI International*, page 312. Springer, 2009.

[93] A. Gray. *Interprocess Communication in Linux*. Prentice Hall Professional Technical Reference, 2002.

[94] J. Grefenstette and A. Schultz. An evolutionary approach to learning in robots. In *Proceedings of the Machine Learning Workshop on Robot Learning, Eleventh International Conference on Machine Learning*. Naval Research Lab Washington DC., 1994.

[95] A. Grizard and C. Lisetti. Generation of facial emotional expressions based on psychological theory. In *1rst Workshop on Emotion and Computing at KI 2006, 29th Annual Conference on Artificial Intelligence*, pages 14–19, 2006.

[96] A. Grizard and C. L. Lisetti. Generation of facial emotional expressions based on psychological theory. In *1rst Workshop on Emotion and Computing at KI 2006, 29th Annual Conference on Artificial Intelligence, June, 14-19, 2006, Bremen, Germany*, 2006.

[97] D. Gu and H. Hu. Neural predictive control for a car-like mobile robot. *International Journal of Robotics and Autonomous Systems*, 39(2):73–86, 2002.

[98] J. P. Gunderson and L. Gunderson. Intelligence ≠ autonomy ≠ capability. In E. Messina and A. Meystel, editors, *Performance Metrics for Intelligent Systems (PerMIS '04), Issues in Developing Adaptive Intelligence I*, 2004.

[99] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1992.

[100] M. Hans, B. Graf, and R. Schraft. Robotic home assistant care-o-bot: Past–present–future. In *Proceedings of 11th International Workshop on Robot and Human Interactive Communication (IEEE Roman 2002)*, pages 380–385, 2002.

[101] M. A. Hearst. Mixed-initiative interaction. *IEEE Intelligent Systems*, 14(5):14–23, 1999.

[102] M. Heerink, B. Kröse, V. Evers, and B. Wielinga. The influence of a robot's social abilities on acceptance by elderly user. *International Journal of Assistive Robotics and Mechatronics*, 7, 2006.

[103] F. Hegel, S. Krach, T. Kircher, B. Wrede, and G. Sagerer. Theory of mind (ToM) on robots: a functional neuroimaging study. In *HRI '08: Proceedings of the 3rd ACM/IEEE international conference on Human robot interaction*, pages 335–342. ACM, 2008.

[104] F. Heider and M. Simmel. An experimental study of apparent behavior. *Journal of Psychology*, 57:243–249, 1944.

[105] J. Heinzmann and A. Zelinsky. Quantitative safety guarantees for physical human-robot interaction. *The International Journal of Robotics Research*, 22(7-8):479, 2003.

[106] C.-C. Ho, K. F. MacDorman, and Z. A. D. D. Pramono. Human emotion and the uncanny valley: a GLM, MDS, and Isomap analysis of robot video ratings. In *HRI '08: Proceedings of the 3rd ACM/IEEE international conference on Human robot interaction*, pages 169–176. ACM, 2008.

[107] K. Höök. *From Brows to Trust: Evaluating embodied conversational agents*, volume 7 of *Human-Computer Interaction Series*, chapter User-Centred Design and Evaluation of Affective Interfaces: A Two-tiered Model, pages 127–160. Springer Netherlands, 2004.

[108] J. Humrichouse, M. Chmielewski, E. A. McDade-Montez, and D. Watson. *Emotion and Psychopathology*, chapter Affect assessment through self-report methods, pages 13–34. American Psychological Association (APA), 2007.

[109] IEEE. Standard for a software quality metrics methodology. *IEEE Std 1061-1992*, 1993.

[110] R. Ierusalimschy, L. de Figueiredo, and W. Celes. *Lua 5.1 Reference Manual*. Lua. org, 2006.

[111] iRobot. *iRobot Roomba® Serial Command Interface (SCI) Specification*. iRobot®, 2005.

[112] K. Isbister. Perceived intelligence and the design of computer characters. Master's thesis, Dept. of Communication, Standford University, 1995.

[113] K. Isbister. *Better game characters by design: A psychological approach*. Morgan Kaufmann Pub, 2006.

[114] H. Ishiguro. Interactive humanoids and androids as ideal interfaces for humans. In *IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces*, pages 2–9. ACM Press, 2006.

[115] ISO/IEC 9126-1. Software engineering–Product quality. Technical Report ISO/IEC 9126-1, International Organization of Standardisation and International Electrotechnical Commission, 2001.

[116] E. Ivanjko, I. Komˇ sić, and I. Petrović. Simple off-line odometry calibration of differential drive mobile robots. In *Proceedings of 16th Int. Workshop on Robotics in Alpe-Adria-Danube Region (RAAD'07)*, 2007.

[117] R. Jackson and E. Fagan. Collaboration and learning within immersive virtual reality. In *Proceedings of the third international conference on collaborative virtual environments*, pages 83–92, 2000.

[118] J. Jang, S. Kim, and Y. Kwak. Calibration of geometric and non-geometric errors of an industrial robot. *Robotica*, 19(03):311–321, 2001.

[119] K. Johns and T. Taylor. *Professional Microsoft Robotics Developer Studio*. Wiley & Sons, 2008.

[120] W. Johnson, J. Rickel, and J. Lester. Animated pedagogical agents: Face-to-face interaction in interactive learning environments. *International Journal of Artificial Intelligence in Education*, 11(1):47–78, 2000.

[121] C. Joslyn. Models, controls, and levels of semiotic autonomy. In *Intelligent Control (ISIC), 1998. Held jointly with IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA), Intelligent Systems and Semiotics (ISAS), Proceedings*, pages 747–752, 1998.

[122] Y. Jung and K. M. Lee. Effects of physical embodiment on social presence of social robots. In *Proceedings of Presence 2004*, pages 80–87, 2004.

[123] D. B. Kaber and M. R. Endsley. The effects of level of automation and adaptive automation on human performance, situation awareness and workload in a dynamic control task. *Theoretical Issues in Ergonomics Science*, 5(2):113–153, 2004.

[124] P. H. Kahn, N. G. Freier, T. Kanda, H. Ishiguro, J. H. Ruckert, R. L. Severson, and S. K. Kane. Design patterns for sociality in human-robot interaction. In *HRI '08: Proceedings of the 3rd international conference on Human robot interaction*, pages 97–104. ACM, 2008.

[125] T. Kanda, T. Miyashita, T. Osada, Y. Haikawa, and H. Ishiguro. Analysis of humanoid appearances in human-robot interaction. In *Intelligent Robots and Systems(IROS) IEEE/RSJ International Conference on*, pages 899–906. IEEE, 2005.

[126] E. Karlsson. *Software reuse: a holistic approach.* John Wiley & Sons, Inc. USA, 1995.

[127] K. Kawamura, R. Pack, and M. Iskarous. Design philosophy for service robots. In *Systems, Man and Cybernetics. Intelligent Systems for the 21st Century., IEEE International Conference on*, volume 4, pages 3736–3741, 1995.

[128] C. Kidd and C. Breazeal. Effect of a robot on user perceptions. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 4, pages 3559 – 3564. IEEE, 2004.

[129] S. Kiesler. Mental models and cooperation with robotic assistants. In *Proceedings, Conference on Human Factors in Computing Systems (CHI)*, pages 576–577. ACM Press, 2002.

[130] L. Kovar and M. Gleicher. Flexible automatic motion blending with registration curves. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 214–224. Eurographics Association, 2003.

[131] L. Kovar, M. Gleicher, and F. Pighin. Motion graphs. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 473–482. ACM Press, 2002.

[132] E. Krahmer, S. Erk, and A. Verleg. Graph-based generation of referring expressions. *Computational Linguistics*, 29(1):53–72, 2003.

[133] E. Krahmer, M. Swerts, M. Theune, and M. Weegels. Error detection in spoken human-machine interaction. *International Journal of Speech Technology*, 4(1):19–30, 2001.

[134] E. Krahmer and N. Ummelen. Thinking about thinking aloud: A comparison of two verbal protocols for usability testing. *IEEE transactions on professional communication*, 47(2):105–117, 2004.

[135] B. Kröse, J. Porta, A. van Breemen, K. Crucq, M. Nuttin, and E. Demeester. Lino, the user-interface robot. *Lecture notes in computer science*, pages 264–274, 2003.

[136] P. Kruchten. *The rational unified process: An introduction*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[137] C. Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[138] T. E. Kurt. *Hacking Roomba: ExtremeTech*, volume 1. Wiley & Sons, 2006.

[139] P. Lang, M. Bradley, and B. Cuthbert. International affective picture system (IAPS): Instruction manual and affective ratings. *The Center for Research in Psychophysiology, University of Florida*, 1999.

[140] C. Larman. *Applying UML and Patterns: An introduction to object-oriented analysis and design and iterative development (3rd ed.)*. Prentice Hall, USA, 2004.

[141] R. Larsen and E. Diener. Promises and problems with the circumplex model of emotion. *Emotion*, 13:25–59, 1992.

[142] J.-H. Lee, P. Jin-Yung, and T.-J. Nam. Emotional interaction through physical movement. In *Human-Computer Interaction,12th International Conference*, pages 401–410, 2007.

[143] I. Leite, C. Martinho, A. Pereira, and A. Paiva. iCat: An affective game buddy based on anticipatory mechanisms. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*, pages 1229–1232. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, 2008.

[144] A. M. Leslie. Do six-month-old infants perceive causality? *Cognition*, 25(3):265–288, 1987.

[145] J. Lester and B. Stone. Increasing believability in animated pedagogical agents. In *Proceedings of the first international conference on Autonomous agents*, pages 21–27, 1997.

[146] J. C. Lester, S. A. Converse, S. E. Kahler, S. T. Barlow, B. A. Stone, and R. S. Bhogal. The persona effect: affective impact of animated pedagogical agents. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 359–366. ACM, 1997.

[147] M. Lewis, J. M. Haviland-Jones, and L. F. Barrett, editors. *Handbook of Emotions (Third Edition)*. The Guilford Press, 2008.

[148] S. Li, A. Haasch, B. Wrede, J. Fritsch, and G. Sagerer. Human-style interaction with a robot for cooperative learning of scene objects. In *ICMI '05: Proceedings of the 7th international conference on Multimodal interfaces*, pages 151–158. ACM, 2005.

[149] J. Liu and R. Ha. Efficient methods of validating timing constraints. *Advances in Real-Time Systems*, pages 199–223, 1995.

[150] Y. Liu and B. Plale. Survey of publish subscribe event systems. Technical Report TR574, Computer Science Dept, Indian University, 2003.

[151] M. Lohse, F. Hegel, A. Swadzba, K. Rohlfing, S. Wachsmuth, and B. Wrede. What can I do for you? Appearance and Application of Robots. In *AISB Workshop on The Reign of Catz and Dogz*, 2007.

[152] K. F. MacDorman. Androids as an experimental apparatus: Why is there an uncanny valley and can we exploit it? In *Toward Social Mechanisms of Android Science: A CogSci 2005 Workshop, Stresa, Italy,*, pages 106–118, 2005.

[153] K. F. MacDorman. Subjective ratings of robot video clips for human likeness, familiarity, and eeriness: An exploration of the uncanny valley. In *ICCS/CogSci-2006 Long Symposium: Toward Social Mechanisms of Android Science*, pages 48–52, 2005.

[154] S. Maddock, J. Edge, and M. Sanchez. Movement realism in computer facial animation. In *Workshop on Human-animated Characters Interaction(part of HCI 2005: The Bigger Picture, The 19th British HCI Group Annual Conference)*, 2005.

[155] P. Maes. How to do the right thing. *Connection Science Journal, Special Issue on Hybrid Systems*, 1, 1989.

[156] A. Makarenko, A. Brooks, and T. Kaupp. ORCA: Components for robotics. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 163–168, 2006.

[157] A. Martinelli, N. Tomatis, and R. Siegwart. Simultaneous localization and odometry self calibration for mobile robot. *Auton. Robots*, 22(1):75–85, 2007.

[158] M. Mateas. *An Oz-Centric Review of Interactive Drama and Believable Agents*, volume 1600/1999 of *Lecture Notes in Computer Science*, pages 297–398. Springer Berlin / Heidelberg, 1999.

[159] J. McCarley, C. Wickens, J. Goh, and W. Horrey. A computational model of attention/situation awareness. In *Proceedings of the 46th Annual Meeting of the Human Factors and Ergonomics Society. Santa Monica, CA: Human Factors and Ergonomics Society*, 2002.

[160] M. D. McIlroy. Mass-produced software components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, pages 88–98. NATO Science Committee, 1968.

[161] B. Meerbeek, J. Hoonhout, P. Bingley, and J. Terken. Investigating the relationship between the personality of a robotic TV assistant

and the level of user control. *Robot and Human Interactive Communication(ROMAN). The 15th IEEE International Symposium on*, pages 404–410, 2006.

[162] B. Meerbeek, M. Saerbeck, and C. Bartneck. Iterative design process for robots with personality. In *Proceedings Adaptive and Emergent Behaviour and Complex Systems (ASIB 2009) - New Frontiers in Human-Robot Interaction*, pages 94–101, 2009.

[163] A. Mehrabian. Framework for a comprehensive description and measurement of emotional states. *Genetic, Social, and General Psychology Monographs*, 121(3):339, 1995.

[164] A. Mehrabian. Comparison of the PAD and PANAS as models for describing emotions and for differentiating anxiety from depression. *Journal of Psychopathology and Behavioral Assessment*, 19(4):331–357, 1997.

[165] A. Mehrabian and J. Russell. *An approach to environmental psychology*. MIT press Cambridge, MA, 1974.

[166] Q. Meng and R. Bischoff. Odometry based pose determination and errors measurement for a mobile robot with two steerable drive wheels. *Journal of Intelligent and Robotic Systems*, 41(4):263–282, 2005.

[167] T. Minato and H. Ishiguro. Construction and evaluation of a model of natural human motion based on motion diversity. In *HRI '08: Proceedings of the 3rd ACM/IEEE international conference on Human robot interaction*, pages 65–72. ACM, 2008.

[168] N. Mohamed, J. Al-Jaroodi, and I. Jawhar. Middleware for robotics: A survey. In *2008 IEEE Conference on Robotics, Automation and Mechatronics*, pages 736–742, 2008.

[169] L. Mol, E. Krahmer, A. Maes, and M. Swerts. The communicative import of gestures: Evidence from a comparative analysis of human-human and humanmachine interactions. *Gesture*, 9(1):97–126, 2009.

[170] M. Mori. Bukimi no tani [The Uncanny valley]. *Energy*, 7(4):33–35, 1970.

[171] S. R. Musse, M. Kallmann, and D. Thalmann. Level of autonomy for virtual human agents. In *ECAL '99: Proceedings of the 5th European*

Conference on Advances in Artificial Life, pages 345–349. Springer-Verlag, 1999.

[172] B. Mutlu, J. Forlizzi, I. Nourbakhsh, and J. Hodgins. The use of abstraction and motion in the design of social interfaces. In *DIS '06: Proceedings of the 6th ACM conference on Designing Interactive systems*, pages 251–260. ACM Press, 2006.

[173] R. Nakatsu, M. Rauterberg, and P. Vorderer. A new framework for entertainment computing: from passive to active experience. *Lecture Notes in Computer Science*, 3711:1–12, 2005.

[174] W. Nelson. Continuous steering-function control of robot carts. *Industrial Electronics, IEEE Transactions on*, 36(3):330–337, 1989.

[175] S. Nishio, H. Ishiguro, and N. Hagita. Geminoid: Teleoperated android of an existing person. *Humanoid Robots: New Developments, I-Tech Education and Publishing, Vienna, Austria*, 2007.

[176] D. Norman. Affordance, conventions, and design. *Interactions*, 6(3):38–41, 1999.

[177] Object Management Group . UML 2.2 Infrastructure and Superstructure Specification. Technical report, (OMG), 2009.

[178] A. Ortony. *Emotions in humans and artifacts*, chapter On Making Believable Emotional Agents Believable, pages 189–211. Bradford Books, 2003.

[179] A. Ortony, G. L. Clore, and A. Collins. *The Cognitive Structure of Emotions*. Cambridge University Press, Reprint edition (May 19, 1990).

[180] T. Ozaki, T. Suzuki, T. Furuhashi, S. Okuma, and Y. Uchikawa. Trajectory control of robotic manipulators using neural networks. *Industrial Electronics, IEEE Transactions on*, 38(3):195–202, 1991.

[181] R. Parasuraman, T. Sheridan, and C. Wickens. A model for types and levels of human interaction with automation. *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, 30(3):286–297, 2000.

[182] J. Park and I. Sandberg. Universal approximation using radial-basis-function networks. *Neural computation*, 3(2):246–257, 1991.

[183] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[184] J. Patrick and N. James. A task-oriented perspective of situation awareness. *A cognitive approach to situation awareness: theory and application*, pages 61–81, 2004.

[185] K. Perlin. Real time responsive animation with personality. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):5–15, 1995.

[186] K. Perlin and A. Goldberg. IMPROV: a system for scripting interactive actors in virtual worlds. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 205–216. ACM Press, 1996.

[187] R. Pew and A. Mavor. *Modeling human and organizational behavior: Application to military simulations.* National academy press, 1998.

[188] Philips. Open Platform for Personal Robotics(OPPR). Online source: http://www.hitech-projects.com/icat/, 2007.

[189] Philips Research. iCat Research Platform . http://www.research.philips.com/robotics.

[190] R. Picard. *Affective computing.* MIT press, 1997.

[191] R. W. Picard. Affective computing: Challenges. *International Journal of Human-Computer Studies*, 59(1-2):55–64, 2003. Applications of Affective Computing in Human-Computer Interaction.

[192] F. Pollick, H. Paterson, A. Bruderlin, and A. Sanford. Perceiving affect from arm movement. *Cognition*, 82(2):51–61, 2001.

[193] A. Powers, S. Kiesler, S. Fussell, and C. Torrey. Comparing a computer agent with a humanoid robot. In *HRI '07: Proceeding of the ACM/IEEE international conference on Human-robot interaction*, pages 145–152. ACM Press, 2007.

[194] D. H. Rakison and D. Poulin-Dubois. Developmental origin of the animate-inanimate distinction. *Psychological Bulletin*, 127(2):209–228, 2001.

[195] M. Rauterberg and M. Hof. How to get a fitting metaphor for a multimedia interface. In A. Grieco, G. Molteni, B. Occhipinti, and B. Piccoli, editors, *Book of short papers of 4th International Conference on Work with Display Units*, volume 3, pages D15–D17, 1994.

[196] M. Rauterberg and P. Steiger. Pattern recognition as a key technology for the next generation of user interfaces. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics–SMC'96*, volume 4, pages 2805–2810, 1996.

[197] M. Rauterberg and O. Strohm. Work organization and software development. *Annual Review of Automatic Programming*, 16(2):121–128, 1992.

[198] B. Reeves and C. Nass. *The Media Equation : How People Treat Computers, Television, and New Media like Real People and Places*. The Center for the Study of Language and Information Publications, 1996.

[199] H. Rittel and M. Webber. Dilemmas in a general theory of planning. *Policy sciences*, 4(2):155–169, 1973.

[200] C. Rose, M. F. Cohen, and B. Bodenheimer. Verbs and adverbs: multidimensional motion interpolation. *Computer Graphics and Applications, IEEE*, 18(5):32–40, 1998.

[201] D. Rousseau and B. Hayes-Roth. Interacting with personality-rich characters. Technical Report Technical Report No. KSL 97-06, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, Stanford, CA, 1997.

[202] R. Rousseau, S. Tremblay, and R. Breton. Defining and modeling situation awareness: A critical review. *A cognitive approach to situation awareness: Theory and application*, pages 3–21, 2004.

[203] B. Roy and T. N. Graham. Methods for evaluating software architecture: A survey. Technical Report 2008-545, School of Computing - Queen's University at Kingston, 2008.

[204] H. A. Ruff, S. Narayanan, and M. H. Draper. Human interaction with levels of automation and decision-aid fidelity in the supervisory control of multiple simulated unmanned air vehicles. *Presence: Teleoper. Virtual Environ.*, 11(4):335–351, 2002.

[205] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[206] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagation of errors. *Nature*, 323:533–536, 1986.

[207] J. Russell. A circumplex model of affect. *Journal of Personality and Social Psychology*, 39(6):1161–1178, 1980.

[208] M. Saerbeck and B. Bleuzé. Waiter application. In *Robot and Human Interactive Communication, 2003. Proceedings. ROMAN 2003. The 12th IEEE International Workshop on, Video Session: HRI Caught On Film*, page 177, 2007.

[209] M. Saerbeck, B. Bleuzé, and A. van Breemen. A practical study on the design of a user-interface robot application. In *Proceedings of the International Conference of Soical Roboitcs 2009*, 2009.

[210] M. Saerbeck and L. Holenderski. Configuration versus programming in user interfaces for autonomous systems. In *Proceedings of the first International Conference on Adaptive and Self-adaptive Systems and Applications (ADAPTIVE 2009)*, 2009.

[211] M. Saerbeck, T. Schut, M. Janse, and B. Bartneck. Expressive robots in education - influence of social supportive behavior on student's learning performance. submitted, 2009.

[212] M. Saerbeck and A. J. van Breemen. Design guidelines and tools for creating believable motion for personal robots. In *Robot and Human interactive Communication, 2007. RO-MAN 2007. The 16th IEEE International Symposium on*, pages 386–391, 2007.

[213] P. Saini, B. E. R. de Ruyter, P. Markopoulos, and A. J. van Breemen. Benefits of social intelligence in home dialogue systems. In *INTERACT*, pages 510–521, 2005.

[214] M. Scheeff, J. Pinto, K. Rahardja, S. Snibbe, and R. Tow. *Experiences with Sparky, a Social Robot*, volume 3 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 173–180. Springer US, 2002.

[215] B. J. Scholl and P. D. Tremoulet. Perceptual causality and animacy. *Trends in Cognitive Science*, 4(8):229–309, 2000.

[216] T. B. Sheridan and W. L. Verplank. Human and computer control of undersea teleoperators. Technical Report A556750, MIT Man-machine Systems Laboratory, 1978.

[217] T. Shibata, T. Mitsui, K. Wada, and K. Tanie. Subjective evaluation of seal robot: Paro-tabulation and analysis of questionnaire results. *Journal of Robotics and Mechatronics*, 14(1):13–19, 2002.

[218] B. Shneiderman. Universal usability. *Communications of the ACM*, 43(5):84–91, 2000.

[219] B. Siciliano and O. Khatib, editors. *Springer Handbook of Robotics*. Springer, Berlin, 2008.

[220] A. Silberschatz, H. Korth, and S. Sudershan. *Database system concepts*. McGraw-Hill, USA, 1998.

[221] F. Silva, L. Velho, P. Cavalcanti, and J. Gomes. An architecture for motion capture based animation. In *In Proceedings of SIBGRAPI'97, X Brazilian Symposium of Computer Graphics and Image Processing*, pages 49–56, 1997.

[222] R. Simmons and S. Koenig. Probabilistic robot navigation in partially observable environments. In *International joint conference on artificial intelligence*, volume 14, pages 1080–1087. Lawrence Erlbaum Associates LTD, 1995.

[223] H. Simpson. Correctness analysis for class of asynchronous communication mechanisms. *Computers and Digital Techniques, IEEE Proceedings*, 139(1):35–49, 1992.

[224] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, 1998.

[225] E. Skinner. A guide to constructs of control. *Journal of Personality and Social Psychology*, 71(3):549–570, 1996.

[226] C. Smith. *Performance Engineering of Software Systems*, volume 1. Addison-Wesley, 1990.

[227] I. Sommerville. *Software Engineering (8th Editon)*. Addison-Wesley, 2007.

[228] C. Stanislavski. *Building a character*. Theatre Arts Books, 2002.

[229] A. Tanenbaum. *Modern operating systems*. Prentice Hall, USA, 2001.

[230] R. A. Teixeira, A. D. P. Braga, and B. R. De Menezes. Control of a robotic manipulator using artificial neural networks with on-line adaptation. *Neural Process. Lett.*, 12(1):19–31, 2000.

[231] F. Thomas and O. Johnson. *The Illusion of Life - Disney Animation*. Walt Disney productions, 1981.

[232] A. L. Thomaz and C. Breazeal. Teachable robots: Understanding human teaching behavior to build more effective robot learners. *Artif. Intell.*, 172(6-7):716–737, 2008.

[233] S. Thrun. Probabilistic robotics. *Communications of the ACM*, 45(3):52–57, 2002.

[234] R. Tiberius and J. Billson. The social context of teaching and learning. *New Directions for Teaching and Learning*, 45:67–86, 1991.

[235] P. D. Tremoulet and J. Feldman. Perception of animacy from the motion of a single object. *Perception*, 29(8):943–951, 2000.

[236] X. Tu. *Artificial Animals for Computer Animation: Biomechanics, Locomotion, Perception, and Behavior*, volume 1635/1999 of *Lecture Notes in Computer Science*. Springer Berlin, 1999.

[237] A. van Breemen. Animation engine for believable interactive user-interface robots. In *Intelligent Robots and Systems(IROS). IEEE/RSJ International Conference on*, volume 3, pages 2873–2878, 2004.

[238] A. van Breemen. Bringing robots to life: Applying principles of animation to robots. In *Proceedings of the Workshop on Shaping Human-Robot Interaction - Understanding the Social Aspects of Intelligent Robotic Products. In Cooperation with the CHI2004 Conference, Vienna*, 2004.

[239] A. van Breemen. iCat: Experimenting with animabotics. In *Proceedings, AISB 2005 Creative Robotics Symposium*, 2005.

[240] A. van Breemen. Scripting technology and dynamic script generation for personal robot platforms. In *Intelligent Robots and Systems(IROS). IEEE/RSJ International Conference on*, pages 3487–3492, 2005.

[241] A. van Breemen and Y. Xue. Advanced animation engine for user-interface robots. In *Intelligent Robots and Systems, IEEE/RSJ International Conference on*, pages 1824–1830, 2006.

[242] A. van Breemen, X. Yan, and B. Meerbeek. iCat: an animated user-interface robot with personality. In *AAMAS '05: Proceedings of the fourth international joint conference on autonomous agents and multiagent systems*, pages 143–144. ACM, 2005.

[243] V. Vinayagamoorthy, A. Steed, and M. Slater. Building Characters: Lessons Drawn from Virtual Environments. In *Proceedings of Toward Social Mechanisms of Android Science: A CogSci 2005 Workshop*, pages 119–126, 2005.

[244] H. Von Foerster and S. Schmidt. *Wissen und Gewissen - Versuch einer Brücke*. Suhrkamp Frankfurt am Main, 1993.

[245] K. Wada, T. Shibata, T. Saito, and K. Tanie. Analysis of factors that bring mental effects to elderly people in robot assisted activity. In *Intelligent Robots and System, IEEE/RSJ International Conference on*, volume 2, pages 1152–1157 vol.2, 2002.

[246] C. Wang. Location estimation and uncertainty analysis for mobile robots. *Robotics and Automation, IEEE International Conference on*, pages 1231–1235, 1988.

[247] D. Watson, L. Clark, and A. Tellegen. Development and validation of brief measures of positive and negative affect: The PANAS scales. *Journal of Personality and Social Psychology*, 54(6):1063–1070, 1988.

[248] P. Watzlawick, J. Beavin, and D. Jackson. Menschliche Kommunikation: Formen. *Störungen, Paradoxien*, 10, 2000.

[249] W. Weber, J. Rabaey, and E. Aarts. *Ambient intelligence*. Springer Verlag, 2005.

[250] C. Webers and U. R. Zimmer. Motion control of mobile robots–From static targets to fast drives in moving crowds. *Auton. Robots*, 12(2):173–185, 2002.

[251] Z. Wei and G. Fang. Model predictive control for robot manipulators using a neural network model. In *Proceedings of Australian Conference on Robotics and Automation*, pages 62–67, 1999.

[252] H. Williams. *Web database applications with PHP and MySQL*. O'Reilly Media, Inc., 2004.

[253] R. Williams. *The Animator's Survival Kit: A Manual of Methods, Principles and Formulas for Computer, Games, Stop Motion and Internet Animators*. Faber & Faber, 2002.

[254] W. Xu, K. Wurst, T. Watanabe, and S. Yang. Calibrating a modular robotic joint using neural network approach. *Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on*, 5:2720–2725, 1994.

[255] W. Zhao, R. Chellappa, P. Phillips, and A. Rosenfeld. Face recognition: A literature survey. *ACM Computing Surveys (CSUR)*, 35(4):399–458, 2003.

[256] A.-M. Zou, Z.-G. Hou, S.-Y. Fu, and M. Tan. *Advances in Neural Networks - ISNN 2006*, volume 3972/2006 of *Lecture Notes in Computer Science*, chapter Neural Networks for Mobile Robot Navigation: A Survey, pages 1218–1226. Springer Berlin / Heidelberg, 2006.

Appendix A

Robot motion experiment material

A.1 Interview questions

- What was your first impression observing these behaviors?
- Could you see any differences in the behaviors? Which differences?
- How would you interpret this behaviors?
- What was your preferred behavior? Why?
- Could you see any similarities between the Roomba and iCat? Which similarities?
- Was it easier to fill in questionnaire for iCat of for Roomba?
- What behavior would you expect from a robot?

INFORMED CONSENT

By signing this form, I indicate that I participate in this experiment voluntarily. I know that I can stop with the experiment at any time without having to mention the reason for stopping.

I am aware that all data that are collected during the experiment are only used for the purpose of this research and are kept confidential. I agree that my data may be used anonymously for scientific analysis and scientific publication.

I had the chance to ask questions about the experiment and each question has been answered satisfactory.

I have read the form and understand its content.

Date:_____

Name: _____

Address:_____

Email:_____

Gender: ☐ male ☐ female

Nationality: _____

Age: ___

Education: _____

Known disabilities:

_____

Signature:

Participant Experimenter: Martin Saerbeck

Figure A.1: Informed consent was handed before the experiment started

Participant: _____

This scale consists of a number of words that describe different feelings and emotions. Read each item and then circle the appropriate answer next to that word. Indicate to what extent these words describe the behavior of the robot.

		Very slightly or not at all	A little	Moderately	Quite a bit	Extremely
1	Interested	○	○	○	○	○
2	Distressed	○	○	○	○	○
3	Excited	○	○	○	○	○
4	Upset	○	○	○	○	○
5	Strong	○	○	○	○	○
6	Guilty	○	○	○	○	○
7	Scared	○	○	○	○	○
8	Hostile	○	○	○	○	○
9	Enthusiastic	○	○	○	○	○
10	Proud	○	○	○	○	○
11	Irritable	○	○	○	○	○
12	Alert	○	○	○	○	○
13	Ashamed	○	○	○	○	○
14	Inspired	○	○	○	○	○
15	Nervous	○	○	○	○	○
16	Determined	○	○	○	○	○
17	Attentive	○	○	○	○	○
18	Jittery	○	○	○	○	○
19	Active	○	○	○	○	○
20	Afraid	○	○	○	○	○

Please see next side □

Figure A.2: Panas scale was filled by the participants after every stimulus

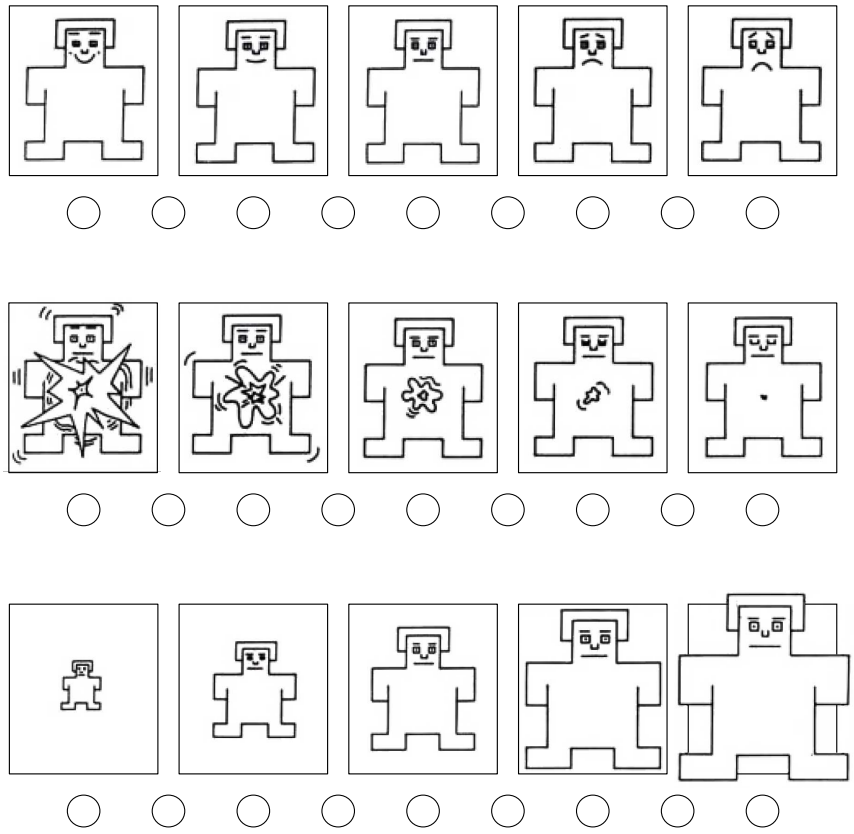


Figure A.3: Self assessment manikin scale was filled in by the participants after every stimulus.

Summary

Software Architectures for Social Robots

Ongoing developments of digital technology steepen the learning curve to operate digital systems. New interface technology is necessary to achieve easy and natural interaction. This PhD project envisions that social interaction technology will change how people perceive and interact with digital technology. Social interaction technology utilizes people's natural abilities to interact with other social communication partners.

The overall goal of this thesis is to equip the application designer with design knowledge, software tools and a unified development environment to support him in the process of creating applications for social robotic interfaces. Throughout this thesis, an iterative design process was followed. Design knowledge has been adopted from the fields of computer science, animation technology, psychology, social science and education to identify design challenges and software requirements.

The design challenges to create applications for social robotic interfaces have been analyzed at three levels: 1) Application design, 2) Interaction design and 3) Behavior design.

At the application design level, it has been demonstrated that incorporating social interaction artifacts in the interface of a system do not necessarily enhance the interaction. To address this problem, a balancing framework has been proposed to guide the design process by requiring a balance between four elements: 1) application, 2) user, 3) interface and 4) technology.

At the interaction design level, this technological design has shown that current interface technology can be classified into two major classes: 1) configuration and 2) programming. Most of the interfaces of contemporary devices fall in the configuration category. However, configuration interfaces do not scale up to provide high level control over device autonomy. Instead, programming interfaces are necessary. The challenge is to make programming easy and understandable for a user. Social interaction technology has the potential to close this gap. For example, a social teaching and learning

metaphor can be used to instruct devices, analogously to how people train their pets to react on certain conditions or to perform certain actions on demand.

At the behavior design level, the application designer faces the challenge to create appropriate behaviors that induce and maintain the perception of dealing with a life-like character. Three basic behavior design dimensions have been identified: 1) naturalness, 2) adequateness and 3) development over time. It has been illustrated that animation technology is well suited for addressing the adequateness dimension. However, interactivity and dynamic change needed to be added to address the other two behavior design dimensions. The functional animation principle was proposed to close this gap. It supports the design process by raising the level of abstraction for behavior design. It combines the control over the expressiveness of keypoint animations with the dynamic power of behavioral animations.

The results of the analysis at these three levels have subsequently been used for deriving the main requirements for the tools and software architecture that have been developed in this thesis. Furthermore, design knowledge from the Philips OPPR software framework and interactions with the iCat community served as valuable source of information in the design for the unified Social Robot Design (SRD) architecture. In summary, four major categories of requirements have been identified: 1) application designer requirements 2) development process requirements 3) domain requirements and 4) framework requirements. For every category, the requirements have been motivated and subsequently been addressed in the architecture specification.

Based on these requirements, a unified software architecture has been developed. Two separate environments have been modeled to best suite the different demands on the software during development and execution time. For both environments, a central component model has been synthesized, which also specified a communication protocol between these components. From these general component structure, several concrete instances have been derived, including a centralized editor concept, preview and debug facilities as well as a situation awareness component that manages sensory input of a robot.

The resulting architecture and tools have been validated in terms of functional and non-functional requirements. Furthermore, the software has been applied in two case studies. The first developed a tutoring application and the second used the developed tools to systematically vary the behavior of two robot embodiments. It was shown that varying the accel-

eration parameter of an animation has a direct influence on the perceived arousal, but that acceleration and curvature have only limited effect on the perceived valence. No significant difference between the two embodiments could be found. Along this line of research, more experiments are necessary to fully classify and predict the perceived emotion. The in this thesis developed tools and architecture facilitate to perform such experiments.

Samenvatting

Softwarearchitectuur for Sociale Robots

Voortdurende ontwikkelingen van nieuwe digitale technologieën, maken het gebruik van deze systemen steeds moeilijker. Nieuwe interactieve technologieën zijn nodig om een gemakkelijke en natuurlijke interactie te bewerkstelligen. Dit promotieonderzoek voorziet dat sociale interactietechnieken de manier waarop mensen digitale technologieën ervaren en gebruiken zullen veranderen. Sociale interactietechnieken maken gebruik van natuurlijke, menselijke vaardigheden om met andere sociale partners te interacteren.

Het doel van dit proefontwerp is om toepassingsontwikkelaars en ontwerpers uit te rusten met ontwerpkennis, softwareapplicaties en een ontwikkelomgeving, om het maken van toepassingen voor sociale robots te ondersteunen. Gedurende het promotieonderzoek is gebruik gemaakt van een iteratief ontwerpproces. Een verzameling van kennis, geleend van computerwetenschappen, animatietechnieken, psychologie, sociale wetenschappen en onderwijs, is toegepast om een programma van eisen voor de te ontwikkelen software op te stellen. De probleemstelling is op drie niveaus geanalyseerd: 1) toepassingsniveau, 2) interactieniveau en 3) gedragsniveau.

Op toepassingsniveau heeft het ontwerp aangetoond dat het gebruik van sociale interactieartefacten niet noodzakelijkerwijs een positief effect heeft op de algehele interactie. Het voorstel om dit probleem op te lossen is een raamwerk, dat het ontwerpproces ondersteunt middels een balans tussen vier essentiële elementen: 1) toepassing, 2) gebruiker, 3) gebruikersinterface en 4) technologie.

Op interactieniveau is aangetoond dat de hedendaagse interactietechnieken kunnen worden opgesplitst in twee categorieën: 1) configuratie en 2) programmering. Het merendeel van de hedendaagse apparaten valt binnen de categorie configuratie. Echter, configuratie is niet voldoende om de gewenste controle over het autonome gedrag van de apparaten mogelijk te maken. Voor zulke controle is een door de gebruiker programmeerbare

interface vereist. De uitdaging hierbij is om de programmering voor de gebruiker begrijpelijk te maken. Sociale interactietechnieken kunnen gebruikt worden om dit mogelijk te maken. Bijvoorbeeld: de manier van leren, zoals gebruikelijk in een leraar-leerling situatie, kan gebruikt worden als een metafoor voor het instrueren van apparaten. Gerelateerd hieraan is de manier waarop mensen hun huisdieren trainen hoe op situaties te reageren en op commando bepaalde acties uit te voeren.

Op gedragsniveau wordt de toepassingsontwikkelaar geconfronteerd met de uitdaging om een geschikt passend gedrag te ontwerpen dat de ervaring oproept van de omgang met een levensecht karakter. Er zijn drie dimensies gedefinieerd voor basisgedragingen: 1) geloofwaardigheid, 2) gepastheid en 3) veranderingen over tijd. Het ontwerp toont dat gepastheid met animatietechnieken kan worden bereikt. Interactiviteit en dynamische veranderingen van het gedrag moesten worden toegevoegd voor de andere twee dimensies. Dit werd bereikt door middel van geparameteriseerde animaties die het ontwerpproces ondersteunen door het verhogen van het abstractieniveau. Hierin wordt de controle over de expressiviteit van keypoint animaties gecombineerd met het voordeel van geprogrammeerde animaties.

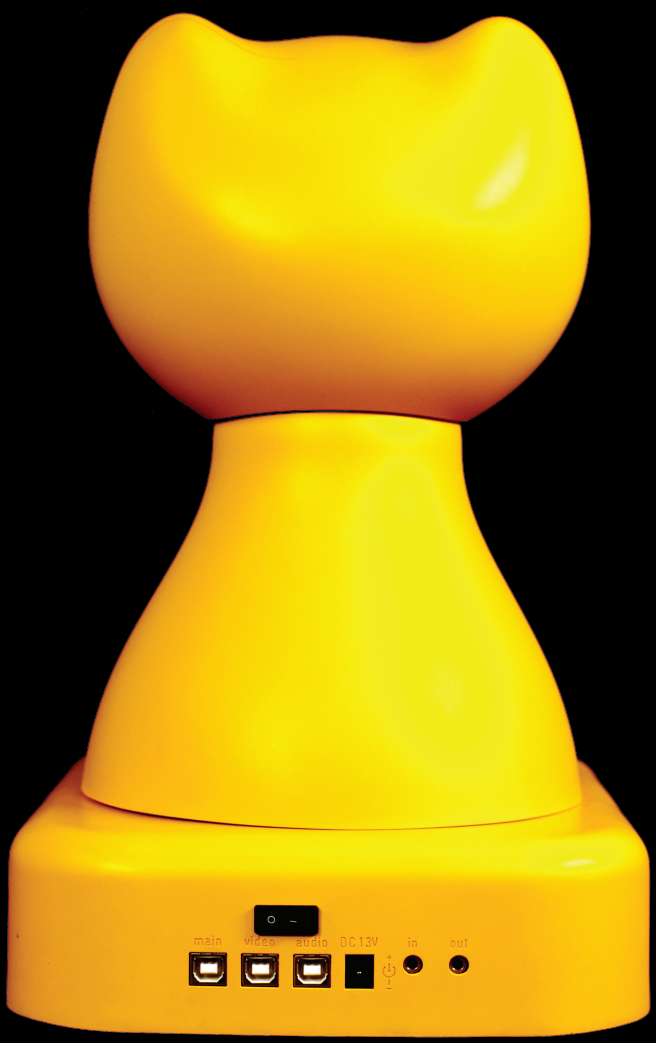
De resultaten van de analyse op drie niveaus zijn achtereenvolgend gebruikt om de basis requirements te definiëren voor de softwareapplicaties en softwarearchitectuur die is ontwikkeld tijdens dit promotieonderzoek. Bovendien hebben kennis van het "Philips OPPR software framework" en interactie met de iCat gebruikersgemeenschap gediend als een waardevolle bron van informatie tijdens het ontwerp van de "Social Robot Design" (SRD) softwarearchitectuur. Samenvattend zijn er vier belangrijke categorieën van ontwerp requirements gedefinieerd: 1) toepassingsontwikkelaar requirements, 2) ontwikkelingsproces requirements, 3) domein requirements en 4) framework requirements. Voor iedere categorie zijn de ontwerp requirements beargumenteerd en daaropvolgend geïmplementeerd in the softwarearchitectuur specificaties.

Deze requirements zijn de basis geweest voor de ontwikkeling van een consistente softwarearchitectuur. Om aan de verschillende eisen aan de software tijdens het ontwikkelen en het uitvoeren te voldoen, zijn er twee onafhankelijke omgevingen gemodelleerd. Voor beide omgevingen is er een centrale component gerealiseerd die tevens een communicatieprotocol tussen beide omgevingen specificeert. Van deze algemene componentstructuur zijn meerdere concrete instanties afgeleid. Onder andere een gecentraliseerde bewerkingscomponent ("editor concept"), "preview-" en "debug"

faciliteiten, alsmede een "situation awareness component" die de invoer van sensordata beheert.
De hieruit resulterende softwarearchitectuur en softwareapplicaties zijn getest op functionele en niet-functionele requirements. Bovendien is de ontwikkelde software toegepast in twee casestudies. In de eerste studie is een studiebegeleidingapplicatie ontworpen en in de tweede studie is de ontwikkelde software gebruikt om het gedrag van twee verschillende robot uitvoeringen systematisch te variëren. Dit toonde aan dat het variëren van de acceleratie ("acceleration") parameter van een animatie van directe invloed was op de "perceived arousal" van de testpersonen. Acceleratie en kromming ("curvature") bleken echter een beperkte invloed te hebben op ervaren "valence". Er konden geen significante verschillen tussen de twee uitvoeringen van de robots worden gevonden. In vervolgonderzoek zullen aanvullende experimenten nodig zijn om de ervaren emoties volledig in kaart te brengen en te voorspellen. De softwareapplicaties en softwarearchitectuur ontwikkeld tijdens dit promotieonderzoek maken het uitvoeren van zulke verdere experimenten mogelijk.

Curriculum vitae

Martin Saerbeck was born in Hamm, Germany, on Juni 10^{th}, 1980. Beginning in April 2001, Martin studied 'Computer Science in the Natural Scienc', an interdisciplinary study on the intersection of computer science with physics, biology, robotics and linguistics, at University of Bielefeld (Germany). He finalized the study in December 2005 and earned the title of a 'Diplom-Informatiker'. In the course of the study he worked in the field of robotics as a student assistant for the European project Cogniron on computer vision and robot middleware. He extended this work during a stay as a guest researcher at University of Amsterdam in January and February 2005. In his diploma thesis he focused on guided machine learning for a robot in an unstructured environment, such as the personal home, and developed a system that allows users to teach a robot objects of the environment.
In January 2006, he joined Philips Research in the 'van der Pol program' and as PhD candidate the Industrial Design department of Eindhoven University of Technology. He has been working on social interaction technology for social robotic interfaces. This technological design is the result from the work in the years 2006 to 2009.



TU/e Technische Universiteit Eindhoven University of Technology

PHILIPS