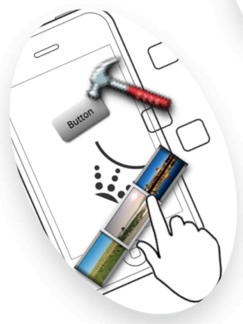
LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

Easing the Creation Process of Mobile Applications for Non-Technical Users

Model-Driven Development of Mobile Applications

Florence Tiu Balagtas-Fernandez M. Sc.



Easing the Creation Process of Mobile Applications for Non-Technical Users

Model-Driven Development of Mobile Applications

Florence Tiu Balagtas-Fernandez M. Sc.

München 2010

Easing the Creation Process of Mobile Applications for Non-Technical Users

Model-Driven Development of Mobile Applications

Florence Tiu Balagtas-Fernandez M. Sc.

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Florence Tiu Balagtas-Fernandez M. Sc.

München, den 03. November 2010

Erstgutachter: Prof. Dr. Heinrich Hussmann
Zweitgutachter: Prof. Dr. Gabriele Taentzer
Tag der mündlichen Prüfung: 07. Februar 2011

Contents

List of Figures

Abstract

In this day and age, the mobile phone is becoming one of the most indispensable personal computing device. People no longer use it just for communication (i.e. calling, sending messages) but also for other aspects of their lives as well. Because of this rise in demand for different and innovative applications, mobile companies (i.e. mobile handset manufacturers and mobile network providers) and organizations have realized the power of collaborative software development and have changed their business strategy. Instead of hiring specific organizations to do programming, they are now opening up their APIs and tools to allow ordinary people create their own mobile applications either for personal use or for profit. However, the problem with this approach is that there are people who might have nice ideas of their own but do not possess the technical expertise in order to create applications implementing these ideas.

The goal of this research is to find ways to simplify the creation of mobile applications for non-technical people by applying model-driven software development particularly domain-specific modeling combined with techniques from the field of human-computer interaction (HCI) particularly iterative, user-centered system design. As proof of concept, we concentrate on the development of applications in the domain of *mHealth* and use the *Android Framework* as the target platform for code generation.

The iterative user-centered design and development of the front-end tool which is called the *Mobia Modeler*, led us to eventually create a tool that features a *configurable-component based design* and *integrated modeless environment* to simplify the different development tasks of end-users. The *Mobia models* feature both constructs specialized for specific domains (e.g. *sensor component*, *special component*), and also those that are applicable to any type of domain (e.g. *structure component*, *basic component*). In order to accommodate different needs of end-users, a clear separation between the front-end tools (i.e. *Mobia Modeler*) and the underlying code generator (i.e. *Mobia Processor*) is recommended as long as there is a consistent model in between, that serves as a bridge between the different tools.

Zusammenfassung

Das unverzichtbarste persönliche Gerät ist heutzutage das Mobiltelefon. Menschen verwenden es nicht nur für die Kommunikation (z.B. zum Telefonieren oder SMS schicken) sondern auch für andere Zwecke. Aufgrund steigender Anforderungen für verschiedene, innovative Anwendungssoftware und des Trends zur kollaborativen Softwareentwicklung haben Mobilfunkunternehmen und Organisationen ihre Geschäftsstrategie verändert. Statt Organisationen, die Anwendungssoftware programmieren, werden APIs und Programmierwerkzeuge öffentlich zur Verfügung gestellt, die Endnutzern ermöglicht ihre eigenen mobilen Anwendungen für den persönlichen Gebrauch erstellen oder zum Verkauf anbieten zu können. Allerdings besteht bei diesem Ansatz das Problem, dass es Menschen gibt, die nette Ideen haben, aber nicht über das nötige technische Fachwissen verfügen, um diese Ideen umsetzen zu können.

Das Forschungsziel dieser Dissertation ist die Vereinfachung der Erstellung mobiler Anwendungen für Menschen ohne Programmierkenntnisse. Um dieses Ziel zu erreichen werden Methoden der modellgetriebenen Softwareentwicklung (insbesondere die domänenspezi sche Modellierung) sowie die iterative nutzerorientierte Gestaltung aus dem Bereich der Mensch-Maschine-Interaktion angewandt. Als "Proof of Concept" konzentrieren wir uns auf die Entwicklung von Anwendungen im Bereich der *mHealth* und nutzen die *Android Framework* als Zielplattform für die Codegenerierung. Die Ausstattung des Frontend Werkzeugs (des sogenannten 'Mobia Modeler'), sind konfigurierbare Komponenten und integrierte "modeless" Designs. Die Mobia Modelle haben domänenspezifische (z.B. *sensor component*, *special component*) sowie domänenübergreifende Komponenten (z.B. *structure component*, *basic component*). Eine klare Trennung zwischen dem Frontend Werkzeuge und der Werkzeug für Codegenerierung ist zu empfehlen, um unterschiedlichen Bedarf der End benutzern zu anpassen. Voraussetzung dafür ist ein einheitliches Modell für die Kommunikation zwischen den verschiedenen Werkzeugen.

Chapter 1

Introduction

This chapter will give the reader an overview of the motivation, problem statement and contributions of this research. A brief overview of the thesis structure and the contents for the succeeding chapters will also be given.

Contents

1.1 Motivation

Nowadays, the most common computing device is the mobile phone. A vast array of features have been incorporated into this device to provide the different demands of users. Aside from hardware features, software applications can easily be downloaded for free or for a price to meet the different computing needs of users through online stores and services such as the App Store from Apple [Appb], the Android Market [Andb] or the Ovi Store from Nokia [Ovi]. While applications related to providing leisure such as games and social networking applications are the most popular in the market today according to a recent survey (Nielsen Survey Q4 2009 [201b]), there is another domain of mobile applications called *mobile health* or *mHealth* which may not be as popular but can have a huge impact to human lives [Con09][IJZ04].

MHealth is characterized by the use of mobile computing, medical sensors and communication technologies for healthcare [IJZ04][OT08]. *MHealth* types of application may be used for general health education and awareness, remote data collection, remote monitoring, training health workers, disease and epidemic outbreak tracking, and diagnostic and treatment support [Con09]. The benefits of *mHealth* systems summarized by Istepanian et al. [IJZ04] can include the promotion of healthy lifestyles through continuous health monitoring (e.g. the MOPET wearable system from Buttussi et al. [BC08], the MPTrain Personal Trainer from Oliver et al. [OFM06b], the Nike+iPod Sports Kit [Nik]), the flexibility and fast access to expert advice at the point of care through telehealth (e.g. the *Sana Platform* for Telehealth [San]), the provision of rapid response to critical medical care in the presence of geographical barrier (e.g. the Alive Heart Monitor from Alive Technologies [Ali]), and for use in the medical research field through real-time collection of health-related data (e.g. the HealthGear system from Oliver et al. [OFM06a]).

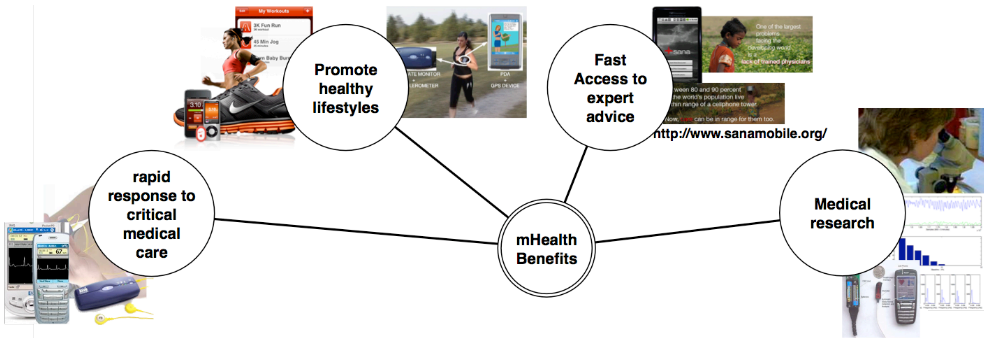

Promote healthy lifestyles
Fast Access to expert advice
http://www.sanamobile.org/
rapid response to critical medical care
mHealth Benefits
Medical research

Figure 1.1: The benefits of *mHealth*.

The challenge for domain-specific mobile applications such as those that can be used for *mHealth* is that it is *quite difficult to find a single application available in the market that would fulfill the different demands and requirements*. A common option is to hire professional software developers who would collect the necessary requirements from domain experts such as doctors, nurses or medical researchers and develop the application for them.

Another option though is through end-user development (EUD). EUD is a paradigm that *allows non-technical users or people with no programming background to develop or modify their own applications* [LPWK06]. In the case of *mHealth*, this means that instead of hiring professional software developers to create these types of applications, domain experts in this field are empowered to create their own applications.

In [BFH09a], some example scenarios were given in order to illustrate the benefits of EUD in the area of *mHealth*. One example is in clinics in which customized *mHealth* applications specific to each patient's needs can easily be created based on the instructions of the doctor. This would be like having some pharmacist concoct a special medicine prescribed by the doctor for a certain patient. Another example is in the field of medical research which may involve studies that require the collection and analysis of physiological information in order prove some hypotheses and formulate conclusions based on the collected data. EUD can be helpful in this area since each experiment is different and may need different types of applications. It would be useful for these researchers to have some tool that they could use to create customized applications depending on the experiments they have devised, and not constantly rely on a programmer to create the application for them. This would be very cost effective and will save a lot on allocation of research funding which can be used for other purposes.

In the scenarios mentioned, EUD has the advantage of *utilizing the end-user's knowledge, ideas and domain expertise as inputs to the design and development of the final application*. *Development time can be reduced* because of the direct translation of ideas into real applications instead of communicating them first to the developers and having something that was not what they wanted to begin with because of some misunderstanding. *Development costs can also be reduced* because there is no more need to hire programmers for development.

1.2 Problem Statement

In the past, software development has been the exclusive domain of technically trained people. However, in the recent years, this barrier is slowly diminishing with the advent of tools that allow non-programmers to create their own software applications. This paradigm is called end-user development (EUD).

According to Lieberman et al. [LPWK06], EUD can be defined as *"a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artifact"*. The challenge for EUD however is on the design of tools and frameworks that would allow end-users to easily develop their own applications that will support them in their goals and needs [LPWK06].

The main goal of this research is to **propose a framework that would allow non-technical users (i.e. non-programmers) to easily create their own mobile ap-**

plications initially for the domain of *mHealth*. This focus on domain allows better capturing and satisfaction of user requirements [LPWK06][KT08]. Later on, the aim for future work is to generalize the solutions discovered to cover other domains as well. In order to come up with the solution, the following questions need to be answered.

What do end-users want to have in tools that allow EUD for this specific application domain (MHealth)? Since the focus of EUD are the end-users, it is important to know what they really want from EUD tools. In order to answer this question, a user-centered design (UCD) [Sta99][Kar96] approach is used in order to capture these requirements. This involves conducting surveys and interviews with people from medical field, people who are involved in medical research, and even those that are not necessarily experts in the health domain but have interest in EUD tools. Also, evaluation in the form of user studies were conducted in order to observe how end-users interact with the tools developed and what their opinions are with regards to the tools.

What design and functionality should tools for EUD of mobile applications provide? There is a vast array of tools that allow EUD of various mobile applications. However, the question is that, do these existing tools provide the necessary design and functionality that the end-users need, and provide an development environment that promotes ease-of-learning and ease-of-use. In order to answer this question, an investigation of existing tools that allow the development of applications for mobile platforms is conducted. Identification of how the different logical constructs for mobile applications are represented in such tools is also needed in order to come up with a simple way to represent them. In order to evaluate if the proposed design and interaction of the tools developed capture what end-users want and need, different tool prototypes were iteratively developed and evaluated in the form of user studies.

What is a good design for an EUD framework? This last question is ultimately the goal of this research which solution is built upon the answers from the previous two questions. The proposed framework follows the model-driven software development (MDSD) [SVC06] paradigm. It is composed of a set of tools, underlying model design and methods as a proposed solution for a framework for EUD of mobile applications.

1.3 Thesis Structure

The three major questions stated in the previous section are answered in each chapter of this thesis. Each chapter starts with a review of related literature that is relevant to the topic of that chapter, a presentation of our own approach, and an evaluation and discussion of the results and lessons learned. Figure 1.2 shows the four core chapters of this thesis.

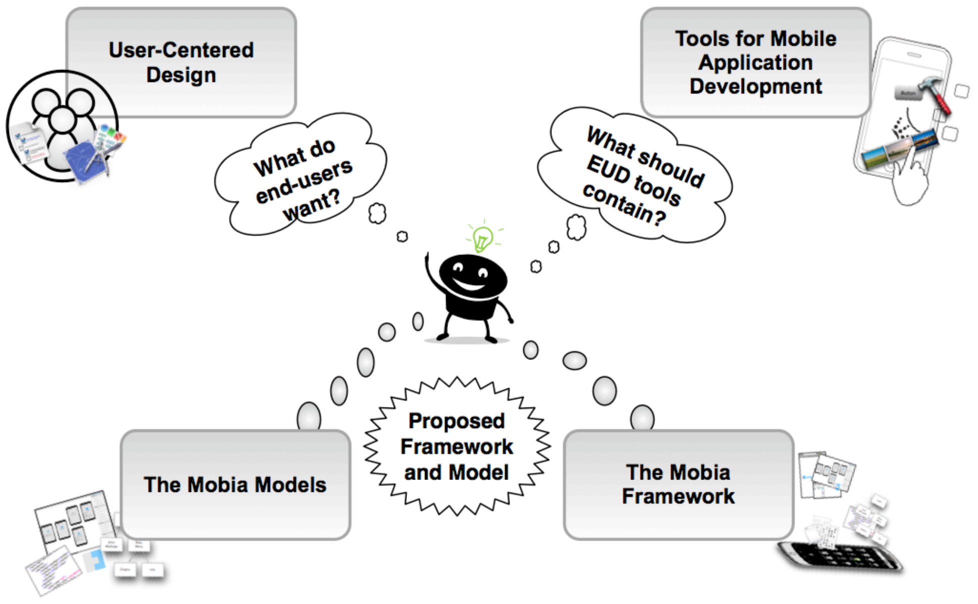

User-Centered Design
What do end-users want?
Tools for Mobile Application Development
What should EUD tools contain?
Proposed Framework and Model
The Mobia Models
The Mobia Framework

Figure 1.2: The core chapters of this thesis.

Chapter 2 aims is to find out *what end users want* by collecting information through the different phases of the user-centered design process. This begins with an overview of the UCD process and other related work. Succeeding sections discuss the different phases of this research and the results discovered in close contact with potential users.

Chapter 3 aims to find out which designs and functionalities EUD tools for mobile application development should provide, by first looking into existing tools that allow mobile application development. These approaches are then analyzed in order to see which would be effective in providing solutions that would allow non-technical users to easily create mobile applications. The different prototypes designed and developed throughout the duration of this research are then presented including evaluation results.

The proposed framework called *Mobia Framework* will be presented in chapter 4. This chapter will give a thorough discussion of the general design, parts and inner workings of the *Mobia Framework*.

One important part of the framework proposed is the underlying model that encapsulates information about the mobile application. The details of the underlying model will be presented in a chapter of its own in chapter 5. The combination of chapters 4 and 5 contains the proposed solution for a *framework that allows EUD of mobile applications*.

1.4 Contributions

The main goal of this research is to propose a framework to make mobile application development accessible to people with no programming skills. The approach combines an iterative user-centered design in the field of Human-Computer Interaction (HCI) and Model-Driven Software Development (MDSD) particularly Domain-Specific Modeling (DSM) approach in Software Engineering.

The main contribution of this thesis is a proposed framework called the *Mobia Framework* which is composed of tools that allow EUD of mobile applications and an underlying model for mobile applications. Also, a minor contribution is made in the understanding of what end-users want and need particularly for the development of *mHealth* applications.

Chapter 2

User-Centered Design

This chapter presents the User-Centered Design (UCD) activities that were carried out throughout the duration of this research. To start off, background information on UCD and some related work are first presented. Discussion will then proceed to the different UCD activities carried out and the lessons learned from them.

Contents

2.1 Related Work

2.1.1 User-Centered Design

User-Centered Design (UCD) or User-Centered System Design (UCSD) is an iterative process which aims to promote usability to different aspects of a system[1]. This is achieved through the active involvement of potential users of the system throughout the development lifecycle [Kar96].

ISO 13407 (Human Centered Design Process for Interactive Systems) [Sta99] is a standard that specifies how to incorporate UCD processes throughout the development of interactive systems. This standard is applicable for both hardware and software design processes. The standard describes UCD as an activity that involves multiple disciplines particularly human factors and ergonomics in order to create an effective and efficient system. The ultimate goal of doing this is to *improve human working conditions, and to empower and motivate the users of the system to learn.* This approach is especially useful in the design and development of EUD systems which goal is to *"empower end-users to develop and adapt systems themselves"* [LPWK06] by featuring tools that are *"easy to understand, learn, use, and teach"* [LPWK06].

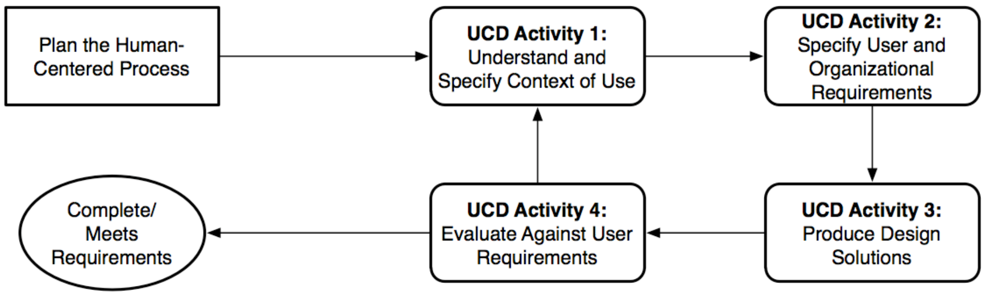

Plan the Human-Centered Process
UCD Activity 1: Understand and Specify Context of Use
UCD Activity 2: Specify User and Organizational Requirements
UCD Activity 3: Produce Design Solutions
UCD Activity 4: Evaluate Against User Requirements
Complete/ Meets Requirements

Figure 2.1: ISO 13407 User-Centered Design (UCD) activities.

There are four UCD activities specified by ISO 13407 [Sta99] (figure 2.1). This includes understanding and specifying the context of use, specifying the user and organization requirements, producing the design solutions and evaluating the design solutions against the requirements. The processes that need to be carried out for each UCD activity during the lifecycle are enumerated in *ISO TR 18529 (Human-Centered Lifecycle Process Descriptions)* [Sta] shown in figure 2.2. These processes are to be done by the organizations applying the UCD process in order to include the users in the whole lifecycle and achieve the defined goals. The processes are linked together and the human-centered lifecycles are iterative. Different versions of the lifecycles can be created depending on the type of system being developed and depending on the target sector the system is intended for [Sta].

[1] This phrase implies that usability does not necessarily have to be applied to the user-interface alone but can be other parts of the system as well.

Although all activities specified in ISO TR 18529 may not be applicable to all systems, the list provides an essential guide for carrying out the UCD process.

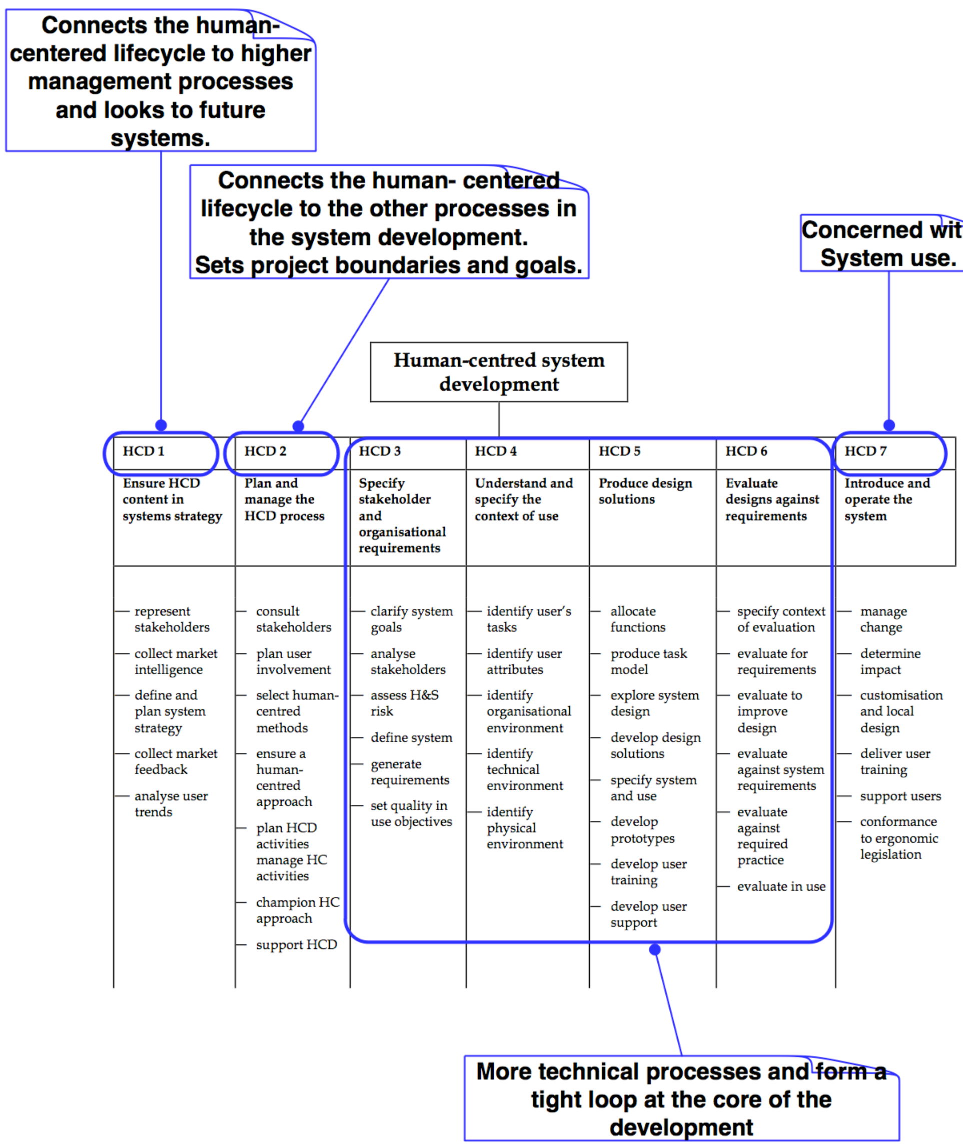

Connects the human-centered lifecycle to higher management processes and looks to future systems.
Connects the human- centered lifecycle to the other processes in the system development. Sets project boundaries and goals.
Concerned with System use.
Human-centred system development
HCD 1
Ensure HCD content in systems strategy
represent stakeholders
collect market intelligence
define and plan system strategy
collect market feedback
analyse user trends
HCD 2
Plan and manage the HCD process
consult stakeholders
plan user involvement
select human-centred methods
ensure a human-centred approach
plan HCD activities
manage HC activities
champion HC approach
support HCD
HCD 3
Specify stakeholder and organisational requirements
clarify system goals
analyse stakeholders
assess H&S risk
define system
generate requirements
set quality in use objectives
HCD 4
Understand and specify the context of use
identify user's tasks
identify user attributes
identify organisational environment
identify technical environment
identify physical environment
HCD 5
Produce design solutions
allocate functions
produce task model
explore system design
develop design solutions
specify system and use
develop prototypes
develop user training
develop user support
HCD 6
Evaluate designs against requirements
specify context of evaluation
evaluate for requirements
evaluate to improve design
evaluate against system requirements
evaluate against required practice
evaluate in use
HCD 7
Introduce and operate the system
manage change
determine impact
customisation and local design
deliver user training
support users
conformance to ergonomic legislation
More technical processes and form a tight loop at the core of the development

Figure 2.2: The processes and base practices from ISO TR 18529 [Sta].

Malmsten et al. [ML08] suggested some methods in order to involve the users throughout the development process. The methods mentioned included interviews, surveys, workshops, focus groups, field studies and usability testing. According to Malmsten et al. [ML08], each specific method has its own strengths and weaknesses depending on which UCD activity it is applied to. For instance, interviews, surveys and field studies are appropriate methods to apply for the first two UCD activities since these methods are good at collecting qualitative data which can provide better information for creating system requirements and context of use (see figure 2.1). On the other hand, focus groups, workshops and prototyping are more suited for producing design solutions and evaluating them [ML08].

In this research, for developing tools that would allow EUD of mobile applications, constant contact with potential end-users were carried out during the design and development processes. The following summarizes the different UCD activities carried out and which will be elaborated in the succeeding sections:

- Interviews and surveys were conducted during the early stages of development in order to gather requirements.
- User studies were conducted during the stages were prototypes were already available for evaluating the design and functionality of the prototypes.
- External validation by experts in the medical domain were gathered in the form of interviews and surveys to validate the usefulness of the tool in this domain, and also to get a general feedback with regards to the design and functionality of the tools developed.

2.1.2 The Worldwide Experimental Platform (WeP)

The *Worldwide experimental Platform (WeP)*[2] is a research project from the LMU Institute of Medical Psychology that aims to *"generate an optimal platform for human experimentation and field studies"* [Roe08]. The studies involved in this project encompass the fields of medicine, epidemiology and genetics.

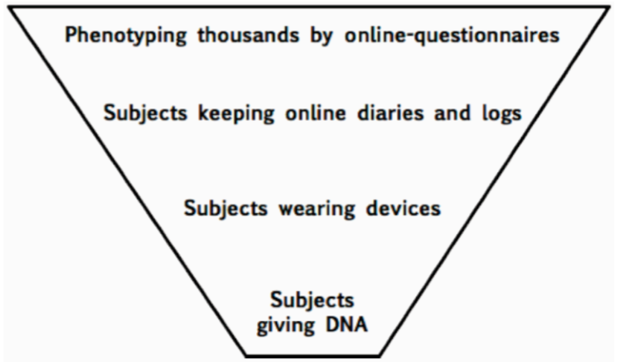

Phenotyping thousands by online-questionnaires
Subjects keeping online diaries and logs
Subjects wearing devices
Subjects giving DNA

Figure 2.3: The WeP strategy from Roenneberg [Roe08][Tea].

The WeP strategy [WeP] consists of different stages as seen in figure 2.3. The first stage involves utilization of the Internet in order to conduct studies for the purpose of documenting typical human behavior. The goal for this stage is to reach as many people as possible from different regions. The second stage involves the recruitment of people

[2]http://www.thewep.org

who were participants in the first stage, to participate in more detailed studies which may involve performing specific tasks such as recording of daily behavior through logs and diaries. From this set of volunteers, a subset will have the possibility of measuring other behavioral and physiological parameters with the use of non-invasive devices. The final level involves collection of DNA for genotyping [Roe08][Tea]. The possible use cases of the WeP platform are shown in figure 2.4.

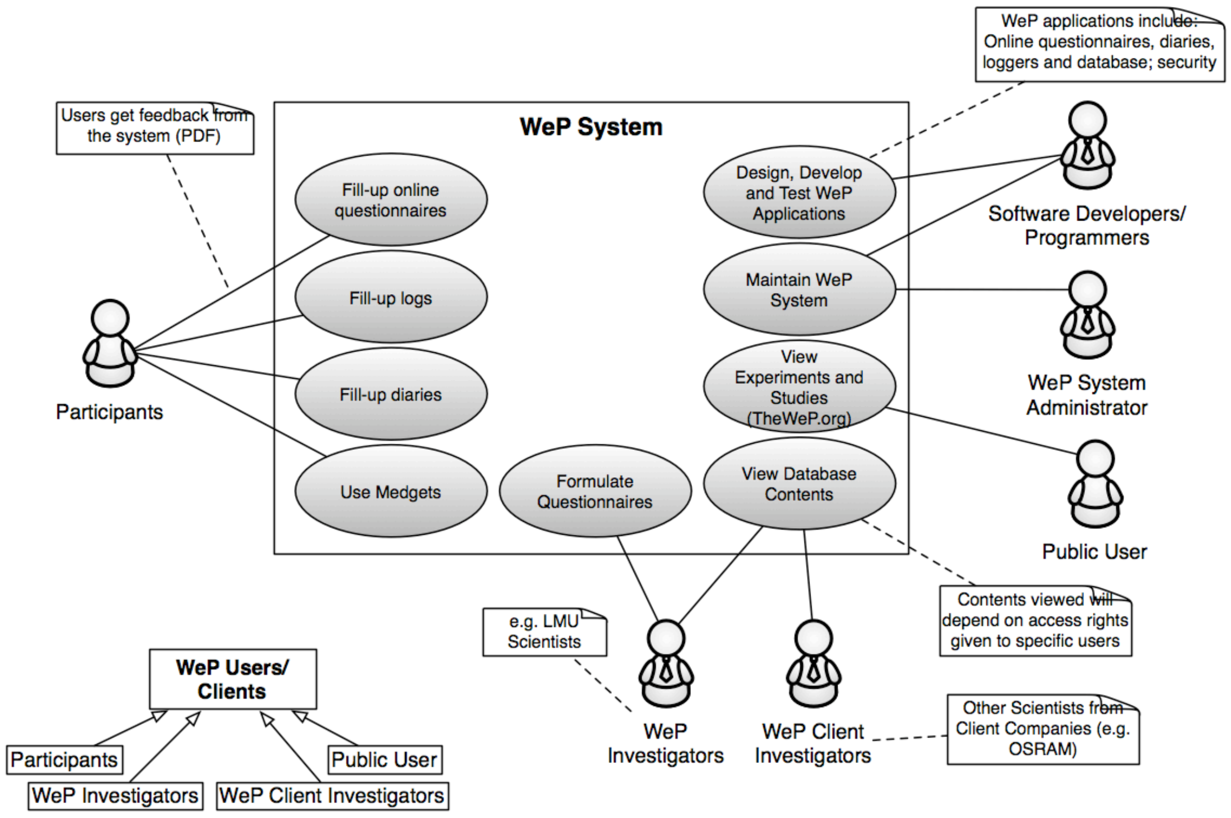

WeP applications include: Online questionnaires, diaries, loggers and database; security
Users get feedback from the system (PDF)
WeP System
Fill-up online questionnaires
Fill-up logs
Fill-up diaries
Use Medgets
Formulate Questionnaires
Design, Develop and Test WeP Applications
Maintain WeP System
View Experiments and Studies (TheWeP.org)
View Database Contents
Participants
Software Developers/ Programmers
WeP System Administrator
Public User
Contents viewed will depend on access rights given to specific users
e.g. LMU Scientists
WeP Investigators
WeP Client Investigators
Other Scientists from Client Companies (e.g. OSRAM)
WeP Users/ Clients
Participants
Public User
WeP Investigators
WeP Client Investigators

Figure 2.4: The possible use cases of the WeP Platform adopted from the WeP Project's Proposal Document Version 1.0 [Tea].

One of the WeP's project vision is to incorporate technological devices such as loggers, diaries and medical gadgets or *medgets* in collecting information from test subjects. *Loggers and Diaries* are devices that collect information by means of explicit logging of information. Loggers and diaries can be in the form of PDAs, cellphones or any other special type of logging device. *MedGets* on the other hand are devices that are used to automatically collect physiological information from the users (e.g. body temperature, amount of light exposure). All of the information collected are transmitted through the network and saved to the WeP database [Tea](figure 2.5).

Since the WeP system is composed of different technological devices, developing applications that would run on these devices are not only tedious but expensive as well. Each experiment that uses a type of device will have a different purpose for a study. This would

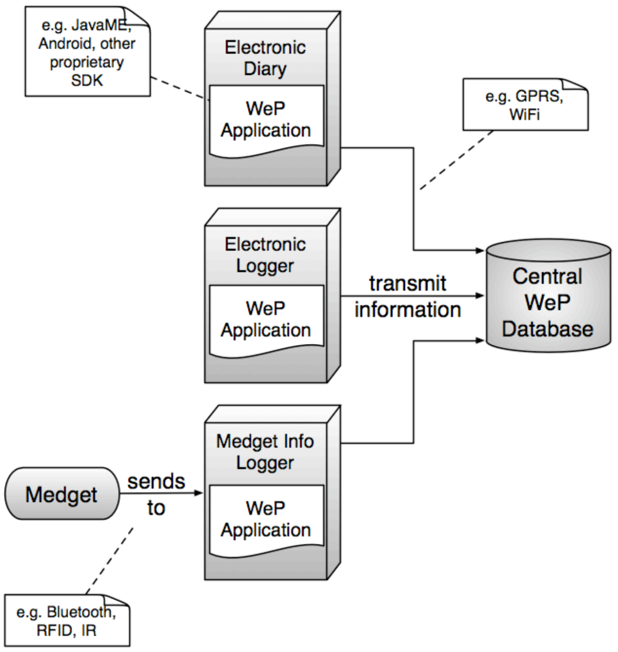


Figure 2.5: Possible device interaction in the WeP System adopted from the WeP Project's Proposal Document Version 1.0 [Tea].

mean developing a separate application for each type of experiment and for a different type of device that would support it. It would be helpful and more cost effective if there were EUD tools available that the WeP Investigators[3] could use so that they can easily create the applications they need for their experiments instead of depending on developers to create these applications for them.

A short involvement with the *WeP project* was the initial step into the decision of making *mHealth* specifically *mobile health monitoring*, as the initial domain to look into for end-user development. The other use cases such as in clinics in which this domain is also applicable, were discussed in the motivation section in chapter 1. The interviews and surveys conducted with the members of the WeP team gave some initial insight to what can be provided for an end-user tool that would allow these researchers to easily create their own *mHealth* applications. Details about the surveys and interviews will be elaborated in the succeeding sections.

[3] The WeP Investigators are the people who carry out studies or experiments using the WeP platform. They are either people who are affiliated with the WeP organization (primary WeP Investigators), or WeP Client Investigators affiliated with organizations who are/become WeP clients [Tea]

2.2 Collection and Validation of Ideas through Surveys

In order get specific ideas on the functionality and design of a tool used for end-user development of *mobile health monitoring* applications, several surveys, each with its own objectives were conducted with potential users.

The name of the tool which is part of the proposed framework is called the *Mobia Modeler*. The *Mobia Modeler* is a modeling tool which aims to allow non-technical users or non-programmers to easily develop mobile applications through graphical modeling. Although the details about the *Mobia Modeler* will be elaborated in the next chapter, the name is introduced because it will be mentioned as we discuss the different UCD processes.

In the latter part of this research, surveys were again conducted in order to validate the design and functionality of the *Mobia Modeler* and its usefulness both in the medical field and research involving *mHealth*.

2.2.1 Tool Functionality Survey

The Objectives. The objective for this survey was to collect information with regards to features of a tool such as the *Mobia Modeler* that potential users want to have.

The Participants. There were five participants who were primary investigators in the WeP project. The different participants' fields of expertise include Computational Physics, Computational Biology, Neuro Cognitive Psychology and Chronobiology.

The Process. The initial part of the survey included questions that aim to collect personal information such as profession, field of expertise, gender and age.

The second part consisted of questions that aim to collect technical background knowledge from the participants, such as the operating systems they use, programming languages for those who have programming backgrounds, and development environments they have used in the past.

The third part consisted of questions with regards to what features they wanted to have in a development environment, which is basically the primary objective of the survey.

Results and Analysis. The results of the survey are summarized in a form of a diagram as shown in figure 2.6. The topmost data in the diagram shows the different operating systems and development tools the participants use. The circles in the center of the diagram represent the participants and the different programming languages they are using or have used in the past. At the bottom of the diagram are the suggested features of development tools the participants would want to have.

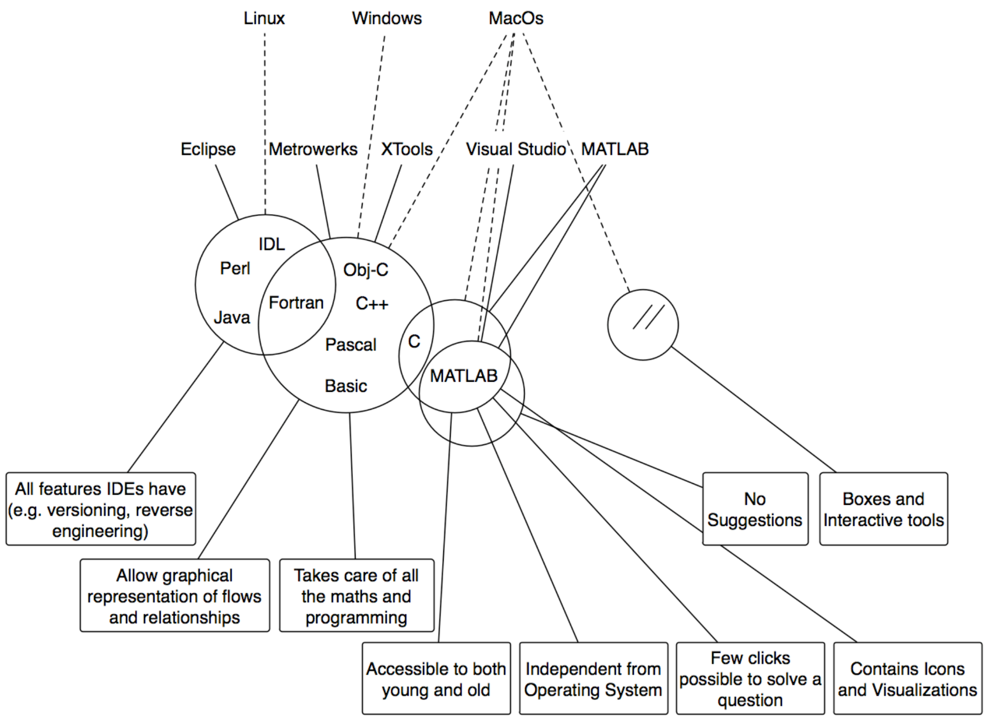

Linux
Windows
MacOs
Eclipse
Metrowerks
XTools
Visual Studio
MATLAB
IDL
Perl
Java
Fortran
Obj-C
C++
Pascal
Basic
C
MATLAB
//
All features IDEs have (e.g. versioning, reverse engineering)
Allow graphical representation of flows and relationships
Takes care of all the maths and programming
No Suggestions
Boxes and Interactive tools
Accessible to both young and old
Independent from Operating System
Few clicks possible to solve a question
Contains Icons and Visualizations

Figure 2.6: A diagram representing the results of the tool functionality survey.

As seen in the results, the first two participants who were exposed to more programming languages than the others were able to suggest specific features that development tools should have such as versioning, graphical representation of flows, programming assistance, etc. On the other hand, the participant that did not have any programming experience as shown by an empty circle in the diagram was only able to give a very vague suggestion concerning boxes and interactive tools. This may be because of the fact that she was not exposed to any development tools at all, so no concrete idea comes to mind. However, one of the participants who was exposed to programming languages such as C and MATLAB, and was also exposed to several tools such as Visual Studio and the MATLAB environment was also not able to give any concrete suggestions despite her exposure to such tools. One participant who only knows MATLAB was still able to suggest some general features that are not specific to development tools, but to any software application in general such as platform independence and usability features such as carrying out tasks with as few clicks as possible and accessibility.

The suggestions from the participants as discussed differed according to their technical background and the tools they were already exposed to. The participants who were exposed to more programming technologies were able to come up with more specific features, while those that were less technologically inclined were only able to come up with more general features.

Recommendations. The challenging part for getting requirements for EUD tools is that, we cannot expect end-users who are not currently exposed to such technology to know and suggest features about tools that they have not yet seen or exposed to. For this survey, although the goal was to gather what features non-technical end-users want, very vague answers were collected from the one participant who in fact was a non-technical user. Those technically inclined users however were able to suggest more advanced features for such tools because of their previous experiences in this type of technology.

A better way of collecting information can be done in two phases of survey questions instead of what was done in this survey. The first phase would be to conduct a survey to collect suggested features from people who are already exposed to technologies such as development tools. After collecting all of these information, a new set of survey questions which contain concrete ideas for features such as versioning, graphical representation of flows, etc. together with some images if possible, would be given to participants who are not exposed to programming at all. In this manner, they will have an idea on what is asked from them. They can either give ratings to which features they prefer and may be able to give concrete suggestions given the examples in the survey.

2.2.2 Health Monitoring Survey

The Objectives. The objective of this survey was to gather information with regards to the trend of health monitoring of outpatients in hospitals. This involves collecting information about the types of technologies currently used, types of information collected that are relevant to medical practitioners such as doctors, and the process of collecting this information. Feedback from experts in the medical field with regards to the potential use of a tool such as the *Mobia Modeler* was also another objective.

The Participants. The challenge for this survey was getting contact with people from the medical field. Only two people, both medical doctors, responded to this survey. One participant was an expert in the field of Endocrinology and Metabolism diseases, while the other one was an expert on Abdominal surgery, Endoscopy and minimal invasive surgery.

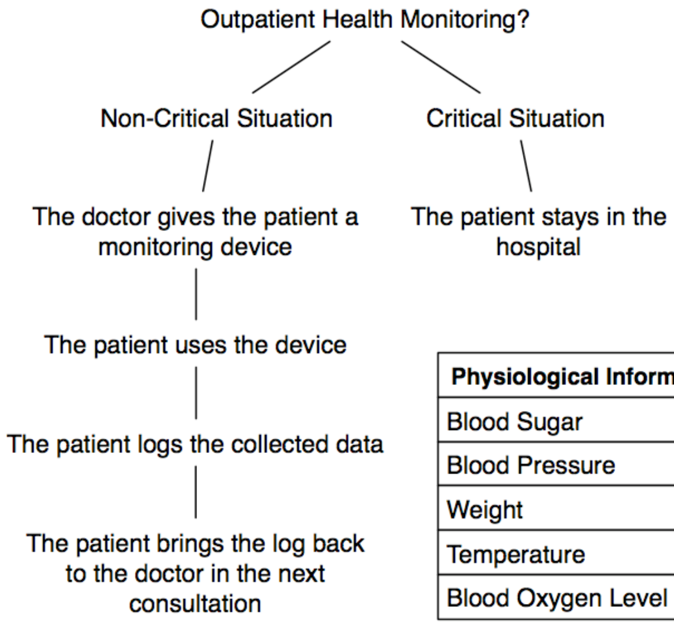
The Process. The initial part of the survey consisted of getting some personal background from the participants such as profession, field of expertise, organization and responsibilities.

The second part consisted of questions with regards to health monitoring devices. This included questions with regards to the purpose of such device and which physiological information were collected.

The third part focused on the typical process that information from these monitoring devices are collected from patients.

Finally, a question with regards to their opinion on how health monitoring technologies would impact future trends in healthcare was asked.

Results and Analysis. Figure 2.7 shows the typical process for collecting physiological information from outpatients. The table shown in the figure are some of the example physiological information with the corresponding devices used in order to collect them. However, this scenario is only typical for non-critical situations. In more urgent and severe situations, patients should be in the hospital, therefore there is no need to use such devices.


Outpatient Health Monitoring?
Non-Critical Situation
Critical Situation
The doctor gives the patient a monitoring device
The patient stays in the hospital
The patient uses the device
The patient logs the collected data
The patient brings the log back to the doctor in the next consultation

Physiological Information	Device
Blood Sugar	Glucometer
Blood Pressure	Tonometer
Weight	Scale
Temperature	Thermometer
Blood Oxygen Level	Pulse Oximeter

Figure 2.7: The typical data collection process for health monitoring of patients and some example physiological information and devices used to collect them.

With regards to the opinions from the participants on how health monitoring technologies would impact future trends in healthcare, one of the opinions expressed by a participant is that there is a potential for using mobile devices and sensors for outpatient monitoring, and would potentially decrease health costs. However, the use of reliable devices such as the quality of sensors for health monitoring is very important.

Recommendations. Because of the nature of work of the people in the medical field, it was quite a challenge to get many of the domain experts to participate in activities such this survey. However, even though only a few participants were able to give their time to answer the few questions in the survey, the information they provided were still relevant in gaining more insight into the field and how outpatient health monitoring is conducted in the real world.

2.2.3 Representation of Inputs and Outputs for Sensor Data Survey

The Objectives. The objective of this survey [BFH09c] was to get user opinions with regards to how inputs and outputs can be represented in a tool such as the *Mobia Modeler*. In particular, the following questions are the things we want to answer in this survey:

- How can information from input devices[4] such as *medgets* be represented in a development environment such as the *Mobia Modeler*?
- How can all available *medgets* be represented in a manner that it is not confusing to the user.
- How can the flow of information from an input device such as a *medget* to the application be visually represented inside the development environment?

In order to find out the answers to the points mentioned, several designs were made and shown to the participants in order to get their opinions.

The Participants. There were 14 participants to the survey, 8 of which have backgrounds in Computer Science while the others are from Engineering, Mathematics and Physics. All of the participants have some programming background and were exposed to different types of development tools in the past.

The Process. The initial part of the survey involved the usual collection of personal background information from the participants such as profession, field of expertise, etc., and collection of technical background such as programming experience, programming language proficiency and development environments used.

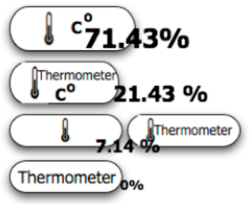
In the second part of the survey, the participants were presented with different designs and questions with regards to how individual *medgets* and their data should be represented, how to display all available *medgets*, and how flow of information from *medgets* to the application screen should be represented.

Results and Analysis.

- **Representing Individual *medgets* and their Data.** One of the concerns in the design of the *medget* is the type of information that should be made available to the user in the development environment. Three types of possible information were identified: *the name* of the *medget*, *the symbol* representing the *medget*, and

[4] Devices that provide information to an application such as sensors that collect information which can be processed as inputs to an application.

the possible data it can provide. The table in figure 2.8 shows the three types of information that can be provided by a *medget* and the possible combinations of these information. The figure also shows the result of the survey in which most of the participants chose the one containing a graphical symbol of the *medget* and the possible information it contains.



Type of Information	Medget Instance
Name of Medget	Thermometer
Representation Symbol	🌡
Available Data	Temperature in Celcius or Fahrenheit

Figure 2.8: Results for the survey on how to represent individual *medgets* and their data.

A conclusion drawn from the said result is that, the user is *not too concerned with the name of the* medget *as long as the symbols representing it is clear enough for the user*. Textual labels were suggested to be shown during a mouseover event for instance. Another thing is the importance of showing the data that a *medget* provides such as the symbol for Celcius for a Thermometer *medget*.

- **Displaying All *Medgets*.** Another objective for this survey was to find out how all available *medgets* and the data they provide can be presented in the development environment that does not confuse the user.

  As an example, let us assume that there are three types of *medgets* available: *medget* A, *medget* B and *medget* C. We will not name the *medgets* here since what is important is how they are positioned and displayed. Each *medget* can have one or more types of data that it can provide which are shown as cross symbols (figure 2.9).

  For the *Individual display*, each *medget* and all possible data it provides are shown as one component in the *medget* palette. For instance, if there are three available data type for each *medget*, then three components for that *medget* are present in the palette.

  The *Grouped display* shows all the possible data grouped according to the *medget* it is taken from. The *medget* palette is divided into three sections for the three types of *medget*, and each section contains all the possible data.

  The *Minimalist display* only shows the available *medgets* in the palette. Later on, when the *medget* is added to a screen instance as shown in the figure, double clicking on the *medget* icon on top of the screen will show possible data representations for each double click.

  For the results of the survey, 46% chose the *Grouped display*, while 31% chose the *Minimalist display* and the rest chose the *Individual display*. From the results collected, we can see that what is important to the user is the *medget* as represented by

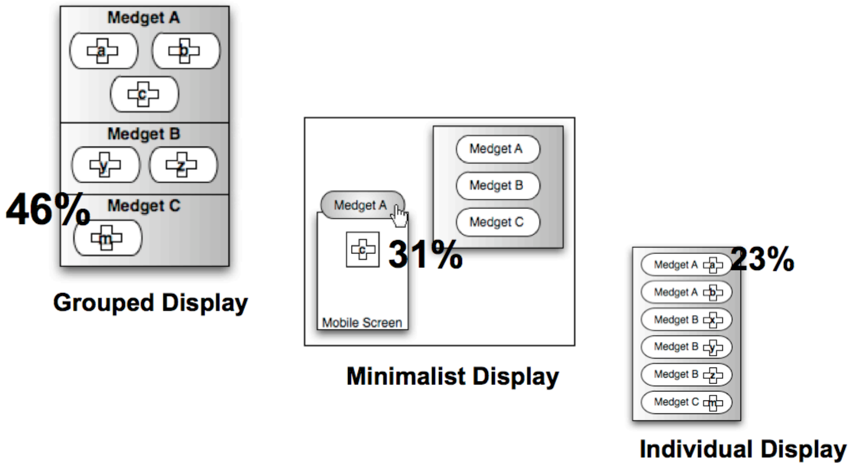


Figure 2.9: Results for the survey on how to display all available *medgets* in the modeling environment.

its icon, and then the type of data it provides. It is also easier to look for what the users need when the *medgets* are grouped according to their type.

- **Visualizing Flow of Information.** The final design question for this survey was how to visualize flow of information from *medgets* to the screen in an application inside the development environment. In order to answer this question, three possible types of interaction were provided (figure 2.10).

  The first one is the *Choose, Drag and Drop* approach which is a common approach in development environments. In this approach, given a set of components in the palette, the user chooses a component and then drags it to a target location.

  The second one is the *Drag, Connect and Click* approach, in which the user drags an arrow from the *medget* to the target screen. This arrow will signify information would flow from the *medget* to the screen. After a connection between a screen and a *medget* is done, the user would then choose the desired data that would appear on the mobile screen.

  Finally, the *Drag, Drop and Click* approach in which the user drags a certain *medget* to the target screen and the *medget* is attached to the top of the screen to symbolize connection. The default data provided by a *medget* will then be shown on the screen. Clicking on the attached *medget* will show a possible data representation for each double click. For instance, if a *medget* has two available data representations, the visuals will toggle between the two each time a double click is done.

  Based on the survey results, majority of the participants preferred the *Choose, Drag and Drop approach* (46%). A big factor that may have resulted from this is because it is the concept that the participants were most familiar with. The second choice (38%) was the *Drag, Drop and Click approach.*The least popular was the *Drag, Connect and*

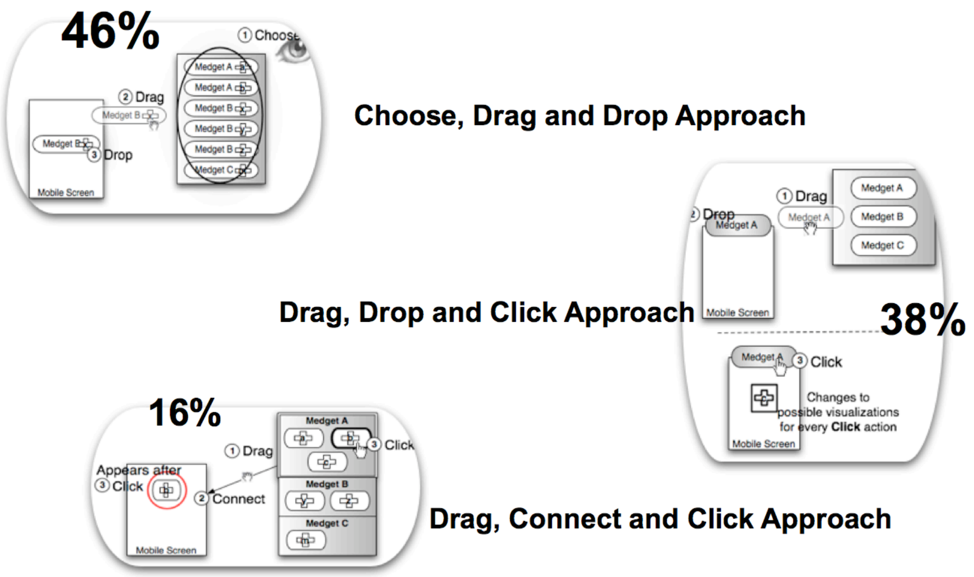

46%
1 Choose
2 Drag
3 Drop
Medget A
Medget B
Medget C
Mobile Screen
Choose, Drag and Drop Approach
38%
1 Drag
2 Drop
3 Click
Changes to possible visualizations for every Click action
Drag, Drop and Click Approach
16%
1 Drag
2 Connect
3 Click
Appears after 3 Click
Drag, Connect and Click Approach

Figure 2.10: Results for the survey on how to visualize flow of information in the modeling environment.

Click approach even though it actually symbolizes some sort of connection with the *medget* in the presence of flow arrows. The reason that this was the least popular among the participants may be because the steps involved in this type of interaction deviate from commonly used approaches such as the simple drag and drop of icons on the screen. It may be difficult for the participants to envision such interaction by just reading how it works and not experiencing how the interaction works themselves.

Recommendations. Some additional recommendations aside from those that were already mentioned for each point in the *results and analysis* follows.

In terms of interaction such as visualizing flow of information, the most commonly encountered approach that the participants were more familiar with was the one mostly chosen. This may be because it is quite challenging to imagine interaction when just shown the design and not be able to interact with it. Surveys are not as effective if the participants are not able to try out the interaction themselves. In terms of the designs though such as representation of *medgets* and their data, the users can easily choose from the designs presented in the survey.

2.2.4 Feedback from Medical Field Experts Survey

The Objectives. During the time this survey was conducted, complete working prototypes of the tools in the *Mobia Framework* were already available, thus the participants were able to see how the tools work. This survey was done together with interviews with some of the participants. The interviews will be discussed further in section 2.3.2.

The following are the objectives of the survey:

- To collect feedback from experts in the medical field with regards to the whole idea of the *Mobia Framework* and its applicability in the medical domain.
- To find out what is still lacking with the *Mobia Framework* particularly the *Mobia Modeler*[5] tool, for the purpose of improving the current design and functionality.

The Participants. There were five participants in the survey, four of which were doctors and nurses, all members of the National Telehealth Center (NThC) of the University of the Philippines[6], while one is an Assistant Professor from the University of the Philippines, College of Public Health whose primary research interest is in the area of Tropical Medicine.

The Process. The first part of the survey involved the collection of personal background information such as profession, field of expertise, and their roles and responsibilities in their respective organizations.

The second part of the survey consisted of questions with regards to the general idea of having a framework such as the *Mobia Framework* for end-user development of applications for *mHealth*. The participants were also asked to try the *Mobia Modeler* available online before answering the questions.

Results and Recommendations. The first three questions in the questionnaire try to capture the initial impression on the concept of the *Mobia Modeler*, its usefulness for experts in the medical field, and what they think are realistic situations in their domain that makes use of such tool. The following points summarize the responses from the participants and validated the importance of having a tool such as the *Mobia Modeler*, and a framework that supports it.

- **As a communication tool with developers.** A tool such as the *Mobia Modeler* and its underlying framework which allows automatic code generation allows health professionals to easily express their ideas to the developers. It will give them a tool that they can use to demonstrate how they think the mobile application should work at the interface level. Most of the time, solutions created by the developers do

[5] The focus is on getting feedback with regards to the design of the *Mobia Modeler* from [Taf09] since it is the one that the target users interact with.

[6] The National Telehealth Center (NThC) of the University of the Philippines Manila was established in June 1998 and *"was given mandate to enhance access to health care through information and communications technology (http://www.telehealth.ph/)"*. Three interrelated programs are managed by the NThC particularly: electronic health records program, eLearning program, and telemedicine program.

not match the actual workflow from these health providers, and thus cause delay in development and even resistance in using the system.

- **As a communication tool with patients.** Having a tool such as the *Mobia Modeler* that can be used to create mobile applications in real-time in front of the patient would be an essential communication tool.
- **As a means of educating health professionals.** Aside from helping health professionals communicate their ideas in a visual format, it can help them appreciate the efforts placed by developers in creating such systems. Since current graduates in the field of medicine are already computer savvy, it is easier for them to use tools such as the *Mobia Modeler*.

Aside from the positive comments, there were also responses that express some concerns on how some people in the medical field would be willing to adopt such technology:

> *" ... despite its ease of use, it would still be a select few who would use this. At the end of the day, most doctors and nurses can't be bothered. This would still fall under the domain of medical and public health informatics, specialists and perhaps researchers. " - Prof. John Solon, UP College of Public Health*

> *" ... it depends on how technology savvy the end-users are. If the doctors are willing to allocate some of their time in learning the technology, then I suppose it would not be a problem. " - Alexandra Belle S. Bernal, Registered Nurse, National Telehealth Center*

However, in general, encouragement on the continued development of the *Mobia Framework* was expressed since it would be very useful if fully completed.

Aside from general feedback, the respondents were also asked if the current version of the *Mobia Modeler* was able to capture which components are needed by medical professionals in order to create health monitoring applications for their patients. The following points summarize the responses from the medical experts during the interview and the survey.

- **Very high-level representation of components.** The current components available in the *Mobia Modeler* are very high level which allows them to easily add complex controls (e.g. voice recorder, note taker) in the model. However, the more basic controls such as text boxes and text areas are unavailable. The expectations from the respondents was that the *Mobia Modeler* would provide them with an interface that would allow them to customize and adjust the look and feel of the forms.
- **Should use domain-specific terminologies.** There are inconsistencies between the terminologies used in the *Mobia Modeler* and the actual terminologies used in the medical field. One example they pointed out was instead of using the word

problems during the configuration wizard in order to describe the health issues, it would be better to use term *domain* and would include options such as *maternal care* or *childcare.*

- **Suggestions for additional target users.** Another additional user for the *Mobia Modeler* suggested was the *midwife*[7].

- **Data Representation in the Modeler.** One question asked was the data model[8] used for persistence for the mobile applications generated by the *Mobia Modeler.* One suggestion to store data specifically in this domain is by using OpenMRS format[9] which is an open-source platform for storing electronic medical records.

- **Design Improvements.** One particular feature that they were looking for was the *undo* functionality. The *X button* on the component was not very visible and obvious for deletion. Another feature they were looking for were form controls (e.g. text boxes, text areas) that would allow them to design how the forms would look like in the mobile application they would be modeling[10].

2.3 Collection and Validation of Ideas through Interviews

Interviews were conducted not only to get ideas for tool functionality and design, but also to validate ideas with regards to the need of having EUD tools for developing applications in the domain of *mHealth.*

2.3.1 An Interview with a Researcher

The Objectives. From the survey conducted with the members of the WeP project (section 2.2.1), only one of the participants during that time uses some application running on a mobile device for her research. An interview was conducted with this researcher for the purpose of knowing more about what she was working on, and what development-related problems she encountered along the way.

[7] A midwife is *"a person trained to assist a woman during childbirth. Additional tasks would be to provide prenatal care, birth education to both parents, and care for mothers and their babies after birth. Depending on the local law, midwives may deliver babies in the mother's home, clinic or in a hospital."*[Mid]

[8] During the design and development of the *Mobia Framework*, data models were not really focused on because of the fact that this information depends on different projects. This responsibility should be taken care of the ones designing and developing code templates for the different platforms.

[9] http://openmrs.org/wiki/OpenMRS

[10] Most of the applications they are concentrating on right now involves diaries and forms that is why they are more concerned on the graphical input components to be present in the modeler.

The Interviewee. The interviewee is a researcher at the Institute of Medical Psychology at LMU. The primary focus of her research is on Circadian Rhythms in real life cognition and shift work settings[11] [201a].

Her Research. The applications she was using for her fieldwork were related to *Psychophysical Tests and Psychomotor Vigilance Tests* which were running on a personal digital assistant (PDA). According to her, this type of experiments can only be conducted in laboratories in the past. However, with the advent of mobile devices, it was possible to bring the experiments to where the test subjects are located (e.g. in the office location of the shift workers). Some examples of the parameters she needed to measure were reaction times and accuracy.

Her Problems. According to her, the tasks performed by her test subjects (i.e. Shift workers) on the PDA were relatively simple tasks. However, the challenge for her was on the modification of the applications that were initially developed by a programmer. Since the programmer was no longer available, she *had to modify parts of the code herself in order to extend the application a bit*, or *correct some functionalities that were not properly done by the programmer* because of some misunderstanding on the specifications.

Fortunately she had a little bit of programming background (e.g. MatLab), so it was still possible for her to understand the code and modify it herself. She did not like the idea of having so many files and forms in the development environment (i.e. Visual Studio) and having to figure out the relationship between them in order to correct or add a certain functionality. She wanted some tool with a visual interface where one can just drag and drop components in order to create an application. In the beginning, she also had to find a the device with the right specifications that would be used for such experiments.

The Need for a tool like the *Mobia Modeler*. She expressed her desire in having a tool to ease her pain in developing these applications so that she can just concentrate on the objective of her research and not worry about the programming part herself.

2.3.2 An Interview with a Medical Expert

The Objectives. The objectives for conducting this interview were enumerated in section 2.2.4.

The Interviewee. In order to collect general feedback from experts in the medical field, interviews with Dr. Alvin B. Marcelo were conducted together with his team from the

[11] http://www.imp.med.uni-muenchen.de/about_us/members/mitarbeiter/vetter/index.html

NThC. Dr. Alvin B. Marcelo [12] is a general and trauma surgeon by training, and did his postdoctoral fellowship in medical informatics at the National Library of Medicine in Bethesda, Maryland. He is currently the manager of the International Open Source Network for ASEAN+3, an Associate Professor and Chief at the Medical Informatics Unit at the UP College of Medicine, and the director of the University of the Philippines Manila, National Telehealth Center (NThC). The NThC is working with doctors in remote areas who have limited access to technology. Since mobile networks and devices are the most accessible in these remote areas, they are looking into technologies (e.g. platforms, tools) and approaches related to mobile technologies that they could use.

General Feedback. The initial part of the interview was a presentation of the *Mobia Framework* which included a brief overview of the goals for creating such framework, the different parts of the framework, and a brief demonstration of how the whole process works from modeling to code generation.

The feedback from Dr. Marcelo was[13]:

> *" I see the value. I think it's very powerful. The approach of having people using it, or the domain experts design the things that they will be using is a very practical and empowering paradigm. The dependence on the programmers is removed.*
>
> *Although the programming side is not yet perfect, the one converting from model to source code, at least from the modeling side, it's already making a big difference. Since the doctors who are designing how it looks like, there is already a sense of ownership. The part where the source code has to be done can be followed later on. Giving the doctor the ability to create the forms or the look and feel, there's already an advantage to that, because they already have a role at the outset.*
>
> *The code at the back will be the problem of the programmers. But it just makes it easier since the programmers will see right away what parts the doctors want to have in terms of interface. " - Dr. Alvin Marcelo, National Telehealth Center*

More detailed feedback were given when he and the rest of his team answered the survey questions which were presented in section 2.2.4.

[12] http://www.alvinmarcelo.com/

[13] The original comments from Dr. Marcelo was a mix of Tagalog and English. Therefore, some parts of the comments presented here is already a translation from the original comment.

2.4 Evaluation through User Studies

Aside from the surveys and interviews conducted in order to collect ideas and feedback from potential end-users, user studies were also conducted in order to evaluate the different *Mobia Modeler* prototypes. The goal of conducting the user studies was to evaluate the usability and designs of the tools.

Since the results of the user studies directly depend on the different versions of the prototypes developed, details about the process, results, observations and lessons learned from these studies will be discussed later in the next chapter, together with the discussion of the *Mobia Modeler* prototypes.

At this point, it is important to mention the presence of this activity as one of the essential parts of the user-centered design process for the development of an EUD tool such as the *Mobia Modeler*.

2.5 Summary and Discussion

The diagram in figure 2.11 shows a summary of the different UCD activities that were carried out during the duration of this research. The activities span a timeline in which prototypes were still unavailable, to the time were the prototypes are already available as symbolized by the two boxes containing the activities. The different purposes for carrying out the activities were also mapped to each activity by connecting them with arrows.

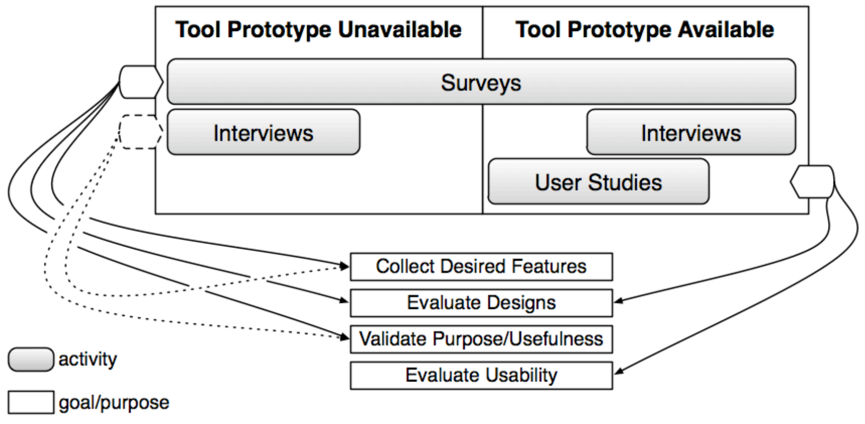


Figure 2.11: A summary of the different UCD activities performed throughout the duration of this research.

As we analyzed the diagram in figure 2.11, during the initial stages of the research wherein prototypes of the *Mobia Modeler* were not yet available, interviews and surveys with potential users can be used to gather requirements for the EUD tools. The requirements can be functional such as enabling users to add flowcharts (section 2.2.1), or non-functional such as requirements concerning usability such as accessibility for both young and old (section 2.2.1). Evaluation of initial designs for such tools can also be carried out through surveys (section 2.2.3) even during the time when prototypes are not yet available. During this initial stage, it is also important to know who would be the potential target end-users for such EUD tools (section 2.3.1).

During the stage wherein prototypes are already available, activities such as user studies are important in order to evaluate usability (section 2.4). Aside from user studies, possible interviews (section 2.3.2) and an additional set of surveys (section 2.2.4) can be carried out in order to get additional feedback from potential users. At this stage, it is relatively easier for survey participants and interviewees to give their feedback since they are able to try out the tools as compared to previous stages where they can only see pictures and read descriptions on how the tools work.

Summary of Lessons Learned. The following is a summary of the lessons learned from the different UCD activities presented in this chapter.

- ***Collecting general information from domain experts and potential users.*** The surveys and interviews conducted were essential in gaining more insight into the domain we are looking into, particularly those that use *mHealth* applications. However, because of the nature of work of some of the participants particularly those in the medical field, it is quite a challenge to get their attention to participate in such activities (i.e. surveys, interviews). However, even though we only got information from a few people, their inputs are still relevant since they gave us a peek into real-world problems and challenges.

- ***Gaining more insight through interviews.*** Different people with different backgrounds have different levels of expectation when it comes to tool functionality as can be seen in the survey responses. It is helpful to really look at a specific type of application and it helps to talk to people working on it in order to discover which problems they encountered during the development of that specific application.

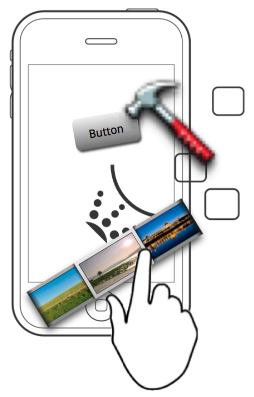
- ***No time to implement? Do a survey first.*** Surveys can be a good tool in order to evaluate design decisions that can be tedious and time consuming to develop before actually implementing the actual prototypes. However, substituting survey for the real thing could pose as a challenge. What can be use instead is to have surveys in which representation of designs (e.g. layout, look and feel, icon designs) are evaluated. For evaluating interaction such as visualizing flow of information, a rough prototype of an application would be more effective.

- ***Nothing like the real thing.*** Conducting surveys in order to collect opinions from participants with regards to interaction methods can be challenging. It is quite difficult for participants and even for the people creating the survey, to imagine what the interaction with the tool might be like without actually having to experience it themselves. As a result, most of the chosen approaches are the ones that they are already familiar with as we have experienced with the survey about modeling sensor information [BFH09c](section 2.2.3).

- ***Evaluation through User Studies.*** Conducting user studies is important to evaluate usability of systems [RC02][DFAB93]. One prerequisite though is the presence of prototypes either low-fidelity ones such as paper prototypes, or high fidelity versions. Details about the user studies will be elaborated in the next chapter as we discuss more about the *Mobia Modeler* prototypes.

Chapter 3

Tools for Mobile Application Development



This chapter will give an overview of currently available systems used for mobile application development. An assessment of the features and approaches used will be presented for the purpose of extracting which of these are suitable for environments that target non-technical users (i.e. non-programmers) as primary users. Also, an overview of how mobile application constructs are represented in some of these systems will be given. Since usability evaluation is the primary means used to evaluate the different *Mobia Modeler* prototypes, a brief overview about this topic will be presented.
The different *Mobia Modeler* prototypes will then be presented in a sequential fashion showing the influence of each prototype to the design of the succeeding versions.
Finally, a summary of the activities that were presented both in this chapter and the preceding one which discussed the user-centered design activities will be given.

Contents

3.1 Related Work

3.1.1 Elements of Development Approaches

After a thorough review of some of the existing systems used to create mobile applications, a set of common elements were extracted. These elements however, may encompass and can be applied not only to mobile application development, but also for other domains and platforms as well.

Methods and Techniques

Methods and techniques refer to the activities involved in order to accomplish the creation of a certain software artifact [Fug00]. This may include the following: programming, scripting, visual programming, modeling and authoring.

Programming is defined to be the act of writing a computer program which is composed of a list of instructions that causes a computer to act a certain way. There are other types of programming aside from the usual typing of source code (e.g. visual programming). However, for explanation purposes, the term *programming* will be used to refer to this traditional way of creating programs.

Scripting is another method used to create programs by typing in source code. It differs from traditional programming in a way that scripting makes use of already existing components and combines them together in order to create an application [Ous98]. Another difference is, scripted programs are usually interpreted while traditionally written programs are compiled. Examples of scripting languages are Perl, Python, Unix shells and JavaScript. Scripting is a bit more high level and is often easier to learn as compared to programming [Ous98]. In the recent years though, the borderline between scripting and programming has more or less disappeared.

Modeling in the context of software development involves the creation of models in order to describe certain aspects of a software application and get insights of the real application through abstraction in order to reduce complexity. Models can be expressed through graphical (e.g. UML[1]), mathematical (e.g. OptimJ[2]) or textual form (e.g. using markup languages such as XML).

Visual Programming is another type of programming which uses graphics to aid in the process of creating, debugging and understanding computer programs [Mye86].

[1]http://www.uml.org/
[2]http://www.ateji.com/optimj.html

Authoring is a method usually employed by users who have little or no experience in programming in order to create software artifacts (e.g. application, document, service). It may combine different methods such as visual programming or scripting. One example in the education domain is the Intelligent Tutoring System (ITS) which is a computer-based instructional system and is usually created by educators with the help of authoring tools [Mur99] (e.g. Mobile Author from Virvou et al. [VA05]).

Programming-by-Example also known as *Programming by Demonstration* uses examples to simplify programming. This is done by letting users demonstrate a specific behavior and having these actions as inputs in order to create a complete program [Mye86]. This approach is mostly applied in the field of robotics for teaching robots new behavior through physical demonstration of tasks.

Development Technology

Development Technology refers to any type of technological support used to create a software artifact and used to *"accomplish software development activities"* [Fug00]. We have surveyed the different types of development technologies currently available which can be integrated into the following general groups: Application Frameworks, Integrated Development Environments (IDE), Modeling Tools, Authoring Tools and Graphical User-Interface (GUI) builders.

An Application Framework is a *"generic structure that can be extended to create a more specific sub-system or application"* [Som04]. It may consist of application programming interfaces (APIs) which can be used to create applications (e.g. Java APIs), or could be an approach or pattern for creating a software artifact (e.g. Model-View-Controller approach to GUI design) [Som04]. One problem with frameworks is that they are inherently complicated and the learning curve in order to learn how to use them is steep [Som04].

An Integrated Development Environment (IDE) is an application that integrates a set of development tools (e.g. code editor, debugger, compiler/interpreter, version control, etc.) that can be used to create computer programs.

A Modeling Tool is an application that may be used to create models of software design, process and/or implementation. These models can be expressed in a variety of ways such as graphical, textual or even mathematical.

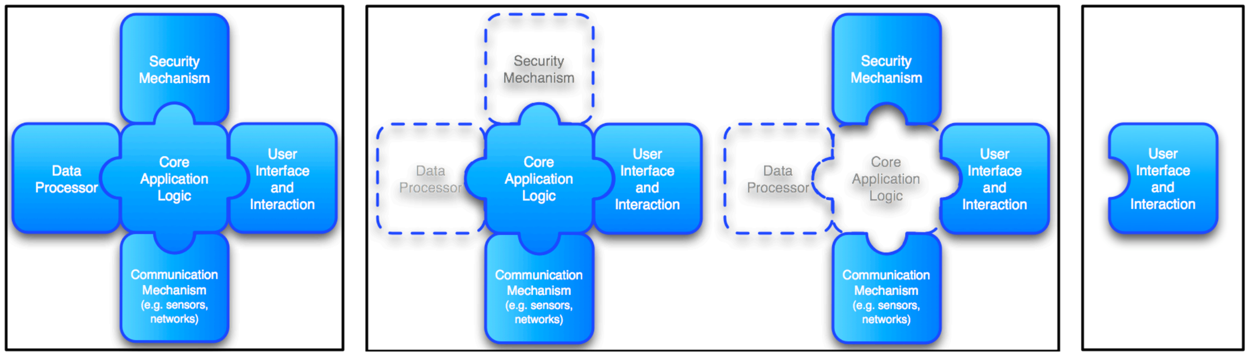
An Authoring Tool is an application that enables users to easily build software artifacts and requires less technical knowledge (i.e. programming skills) to use. Complete applications or their prototypes may be created by combining objects together, defining object relationships or setting particular properties.

A Graphical User Interface (GUI) Builder is an application that enables users to design and create graphical user interfaces for software applications through drag-and-drop means. They are usually WYSIWYG Editors which may be stand alone programs or integrated into a development environment.

Generated Artifact Completeness

Generated artifacts refer to parts of an application being generated when employing a certain type of method or technique. For example, a generated artifact for programming may be a set of source codes, while a generated artifact for modeling may be a set of graphical or textual representations of the model. A generated artifact may be a fully functional application, or may only be a part of an application such as the graphical user interface.

In evaluating the current approaches for mobile application creation, we concentrate not on the generated artifacts themselves, but on the *completeness of the generated application*. We classify them into three types of output base on completeness namely: complete, partial and unit (see figure 3.1 for examples).


Security Mechanism
Data Processor
Core Application Logic
User Interface and Interaction
Communication Mechanism (e.g. sensors, networks)
Security Mechanism
Data Processor
Core Application Logic
User Interface and Interaction
Communication Mechanism (e.g. sensors, networks)
Security Mechanism
Data Processor
Core Application Logic
User Interface and Interaction
Communication Mechanism (e.g. sensors, networks)
User Interface and Interaction

(a) Complete (b) Partial (c) Unit

Figure 3.1: The examples illustrate the differences between the output completeness.

Complete Output refers to a *complete set of compiled source code that can already be deployed* on a device. The output may be prototype versions of the final application, however only minimal work has to be done (e.g. application aesthetics) in order to complete it. One example would be a fully functional *"Chat Application"* which can readily be installed and used on a mobile device.

Partial Output refers to a *partially completed application that still lacks other functionalities in order to complete the application.* It may be able to run by itself (i.e. standalone) or needs to be incorporated to another application in order to be fully functional.

Unit Output refers to an output that is only *one part of an application*. An example of this would be a description file that contains the design and layout of the user interface for a certain application but does not yet contain the necessary application logic.

Level of Technical Expertise

One of the factors that is essential in deciding what *approach* is ideal for the purpose of this research is the target user's (i.e. creator) technical expertise. In this context, we refer to the *technical expertise* as the *ability to create program code or the level of programming knowledge and experience*. It is quite challenging to assess each person's technical expertise and classification cannot be binary (i.e. programmer or non-programmer). However, for our purpose, we try to classify the level of expertise by using the concept of *personas*.

A *Persona* as described by Blomkvist [Blo02] is an *archetypical representation of real or potential users* that represent certain patterns of behaviors, goals and motives which are integrated into one fictional description of an individual. When creating personas, one must take note of the *goals, skills, attitudes and working environment* of each of these personas [Goo01]. The use of personas in order to design systems has been a common practice in the development of software systems. Dotan et al. [DMLG09] for example, used personas for redesigning the user interface of a system that supports informal learning in the workplace. Personas were represented based on their role in the organization (senior or junior employee) and on their work processes (rigid or flexible). Another example is from Allen et al. [ASW05], who also tried to apply the technique of using personas in order to build effective modeling tools to support domain experts in verification, validation and testing. The personas were based on whether they create the models themselves or used models already created by someone else, and also according to which processes they were doing (e.g. modeling checking, verification or validation).

For this research, the personas will be named based on their *technical skill level*. These names are based on the *Benner's Stages of Clinical Competence*[3] [Ben82] which were adapted from the *Dreyfus Model of Skill Acquisition*[4] [DD80]. From Benner's stages, we will only use three of the stages as the basis for our personas namely: the Novice, the Competent and the Expert. For each of the personas, the *goals, skills, attitudes and working environment* will be discussed.

The Novice's goal is to create a software artifact which may be a fully working application (see section 3.1.1 for output completeness), that he can use for his own needs in

[3] Benner classified the different stages as: novice, advance beginner, competent, proficient and expert [Ben82].

[4] The Dreyfus Model of Skill Acquisition postulates that: *"in acquiring a skill by means of instruction and experience, the student normally passes through five developmental stages which we designate novice, competence, proficiency, expertise and mastery"*. In order to facilitate advancement, the skill training procedure must be based on how skills are acquired for every stage [DD80].

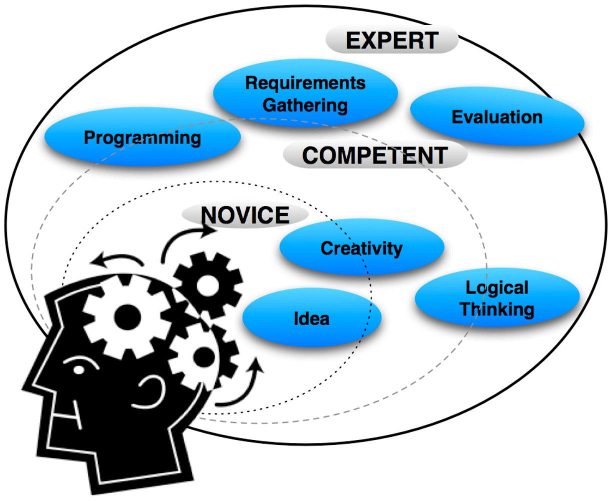


Figure 3.2: The personas and their possible skill levels.

his own specific domain. He has *no experience whatsoever in programming* and has difficulty in expressing what he wants in a logical manner. For example, he might have some idea what he wants to have in his application, but does not know where to start and what to do in order to achieve this. *"Constant monitoring and feedback should also be given"* [DD80] in order to assist the *novice* in improving his skills (e.g. using the tools).

The Competent have the same goals as the novice. He has some knowledge of programming and *has ability to write simple programs.* This competency could be *"acquired when considerable experience have been acquired. These experiences may be in the form of a clear set of examples in order for a person to get meaningful patterns from real situations"* [DD80]. For example, he may have some basic experience in scripting languages, and is able to write simple application logic through pseudocodes.

The Expert also has the goal of creating fully working applications, and may be driven to create such applications for his own purposes or for others. He has the *ability to design, develop (i.e. write code), debug and deploy applications* with the use of any development technology (e.g. IDEs). He may not have the domain expertise for that certain type of application, but is able to extract the necessary information from domain experts in order to create the application. The expertise acquired by this persona comes from a *"a vast repertoire of experiences in the past which allows him to intuitively associate a certain action in a specific situation"* [DD80].

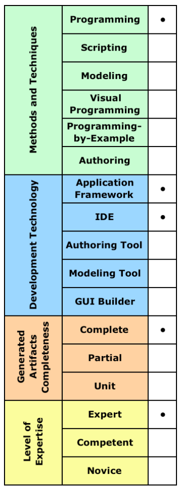
An example of this is when he is creating a web-based business application for a certain company. He may not have enough knowledge about the company itself and the type of business the company has, but he has some idea on where to start, what information he needs to collect and what to do in order to complete this project.

3.1.2 Current Systems for Mobile Application Development

In this section, systems that allow the development of software artifacts for mobile platforms will be presented. To simplify the presentation of the different systems, a feature table that contains the different elements of approach discussed in section 3.1.1 (e.g. methods and techniques, development technologies, generated artifact completeness, user's level of expertise) for each system will be provided. A detailed comparison of the systems will be discussed in section 3.1.3.

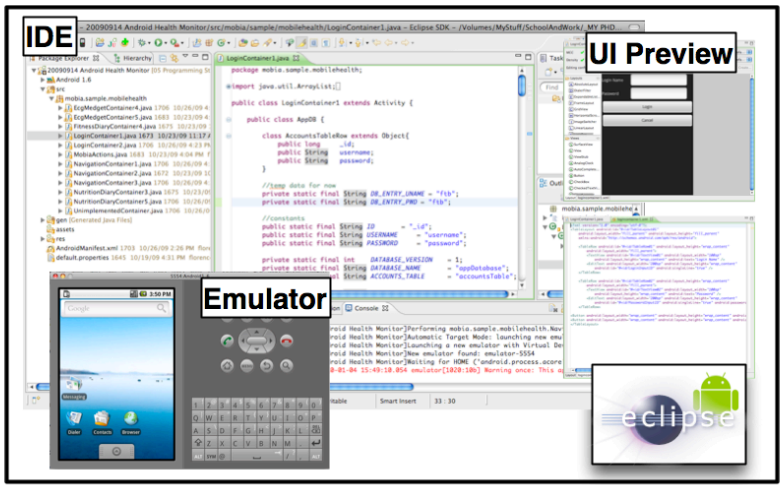
Android Development Tools Plugin for Eclipse

The *Android Development Tools (ADT)* [Andc] plugin (figure 3.3) is an addition to the Eclipse IDE. It is used to easily create, debug and test Android applications. As of the moment, it does not feature a drag-and-drop GUI environment[5] for designing Android user interfaces. The user interface can be created either by editing an XML file for the placement of the GUI components on the screen, or by directly adding lines to the source code for the GUI. A preview of the user interface is provided when the XML editor is used. The ADT also provides an Android emulator and debugger to test the created applications.



Methods and Techniques	Programming	•
	Scripting	
	Modeling	
	Visual Programming	
	Programming-by-Example	
	Authoring	
Development Technology	Application Framework	•
	IDE	•
	Authoring Tool	
	Modeling Tool	
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	•
	Competent	
	Novice	

(a)



```
package mobia.sample.mobilehealth;

import java.util.ArrayList;

public class LoginContainer1 extends Activity {

    public class AppDB {

        class AccountsTableRow extends Object{
            public long     _id;
            public String   username;
            public String   password;
        }

        //temp data for now
        private static final String DB_ENTRY_UNAME = "ftb";
        private static final String DB_ENTRY_PWD = "ftb";

        //constants
        public static final String ID        = "_id";
        public static final String USERNAME      = "username";
        public static final String PASSWORD      = "password";

        private static final int    DATABASE_VERSION    = 1;
        private static final String DATABASE_NAME       = "appDatabase";
        private static final String ACCOUNTS_TABLE      = "accountsTable";
```

```
roid Health Monitor]Performing mobia.sample.mobilehealth.Nav
roid Health Monitor]Automatic Target Mode: launching new emu
roid Health Monitor]Launching a new emulator with Virtual De
roid Health Monitor]New emulator found: emulator-5554
roid Health Monitor]Waiting for HOME ('android.process.acore
0-01-04 15:49:10.054 emulator[1020:10b] Warning once: This a
```

(b)

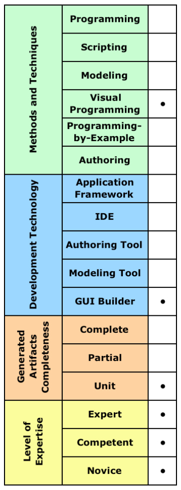
Figure 3.3: The Eclipse IDE with ADT Plugin and the Android Emulator [Andc].

DroidDraw

DroidDraw [Dro] (figure 3.4) is a user interface designer/editor for the Android Platform. It is available both as a web-based application and as a standalone version for some selected

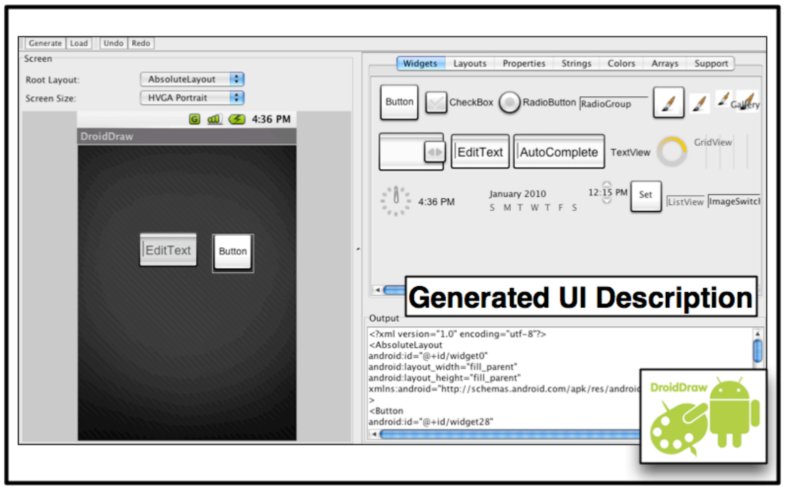
[5] Currently, there is no way to easily create Android GUI within the Eclipse environment. Perhaps in the future, as the platform matures, plugins will be developed for this purpose.

operating systems. It features a graphical interface that allows the design of user interfaces for Android applications by dragging components on a screen. An XML representation of the designed user interface is then generated by *DroidDraw* which can then be integrated to the Android source code. This tool can be used together with the ADT plugin for Eclipse to easily design the user interface for Android applications.



Methods and Techniques	Programming	
	Scripting	
	Modeling	
	Visual Programming	•
	Programming-by-Example	
	Authoring	
Development Technology	Application Framework	
	IDE	
	Authoring Tool	
	Modeling Tool	
	GUI Builder	•
Generated Artifacts Completeness	Complete	
	Partial	
	Unit	•
Level of Expertise	Expert	•
	Competent	•
	Novice	•

(a)



```
<?xml version="1.0" encoding="utf-8"?>
<AbsoluteLayout
android:id="@+id/widget0"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
xmlns:android="http://schemas.android.com/apk/res/androi
>
<Button
android:id="@+id/widget28"
```

(b)

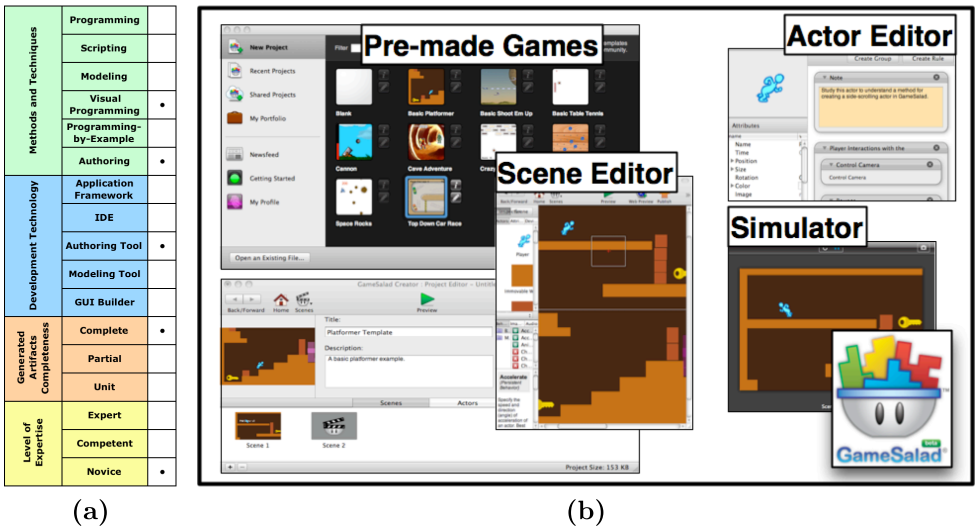
Figure 3.4: DroidDraw User Interface Designer/Editor for the Android Platform [Dro].

GameSalad

GameSalad [Gam] (figure 3.5) is a tool that allows non-programmers to create games for iPhone/iPod platforms and the web. It features a visual environment that allows the creation of games without having to write a single line of code. A game is created by assembling and configuring graphical components and scenes. It features an *Actor Editor* that is used to configure attributes and behavior of a certain actor in a scene (e.g. actor bounces when colliding with a wall). It also has a *Scene Editor* wherein scene components can be easily added and laid out. The creation of games is also easier to learn by providing pre-made templates that users can choose from and customize to their own needs. It also features a simulator that shows the final game product.

MakeIt Toolkit

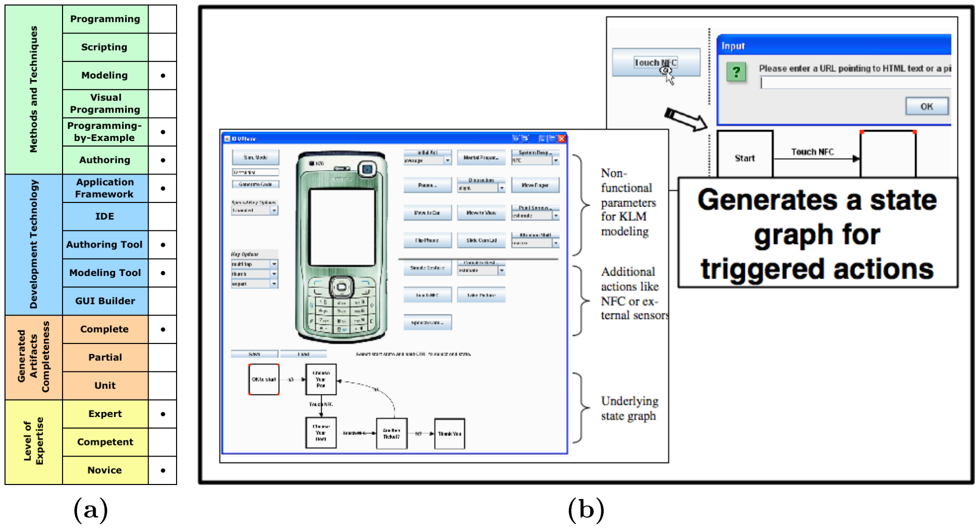
The *MakeIt (Mobile Applications Kit Embedding Interaction Times) Toolkit* [HS08] (figure 3.6) is a prototyping environment that aims to aid designers and developers by allowing the quick creation of high-fidelity application prototypes for mobile devices. It supports the prototyping of applications that uses advanced interaction techniques which may use



Methods and Techniques	Programming	
	Scripting	
	Modeling	
	Visual Programming	•
	Programming-by-Example	
	Authoring	•
Development Technology	Application Framework	
	IDE	
	Authoring Tool	•
	Modeling Tool	
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	
	Competent	
	Novice	•

Figure 3.5: The GameSalad for iPhone and web platforms [Gam].

internal and external sensors. The designer is presented with a graphical interface in which an application can be created by linking actions to visual elements through point and click (i.e. programming by demonstration). The control flow of an application and possible sequences of user actions are represented via a state graph. The tool generates a midlet and Netbeans project files which can be edited further by the developers in order to create the final application.



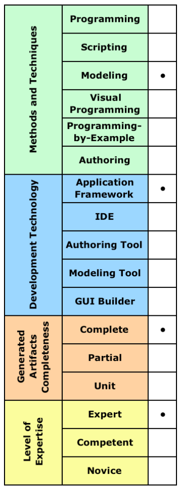
Methods and Techniques	Programming	
	Scripting	
	Modeling	•
	Visual Programming	
	Programming-by-Example	•
	Authoring	•
Development Technology	Application Framework	•
	IDE	
	Authoring Tool	•
	Modeling Tool	•
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	•
	Competent	
	Novice	•

Touch NFC
Input
Please enter a URL pointing to HTML text or a pi
OK
Start
Touch NFC
Generates a state graph for triggered actions
Non-functional parameters for KLM modeling
Additional actions like NFC or external sensors
Underlying state graph
(a)
(b)

Figure 3.6: The MakeIt Toolkit from Holleis et al. [HS08].

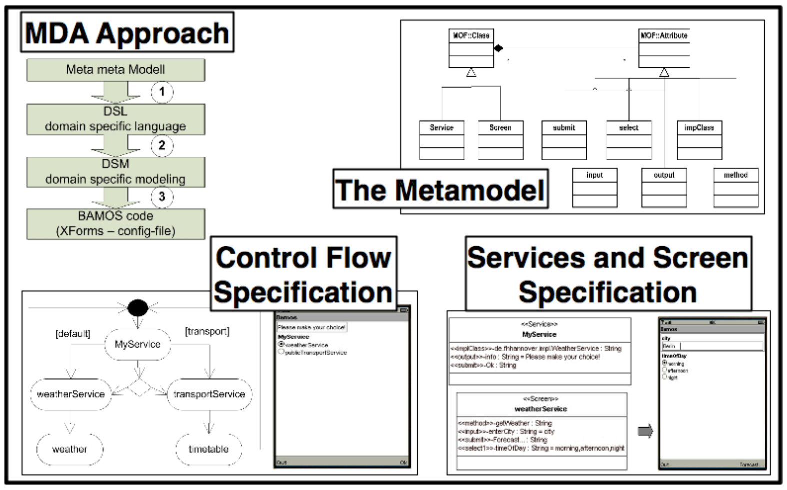
MDA Approach for Mobile Applications Development from Dunkel et al.[DB07]

The work of Dunkel et al. [DB07] (figure 3.7) tries to apply the *Model-Driven Architecture (MDA)* approach to the creation of business applications for mobile platforms. The target application generated uses the *BAMOS (Base Architecture for MObile applications* in Spontaneous networks) architecture which is a platform for developing mobile applications. They defined a domain-specific language which is an extension of the Unified Modeling Language (UML). UML Class Diagrams were used to specify services and screens which are transformed to XForms which are then used by the BAMOS architecture. Control flow is represented with UML activity diagrams. Different tools were used in order to support the creation of the application (e.g. UML modeling tools, code generating tools) [DB07].



Methods and Techniques	Programming	
	Scripting	
	Modeling	•
	Visual Programming	
	Programming-by-Example	
	Authoring	
Development Technology	Application Framework	•
	IDE	
	Authoring Tool	
	Modeling Tool	
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	•
	Competent	
	Novice	

(a)

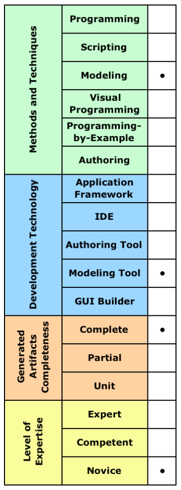


(b)

Figure 3.7: The proposed MDA approach from Dunkel et al. [DB07].

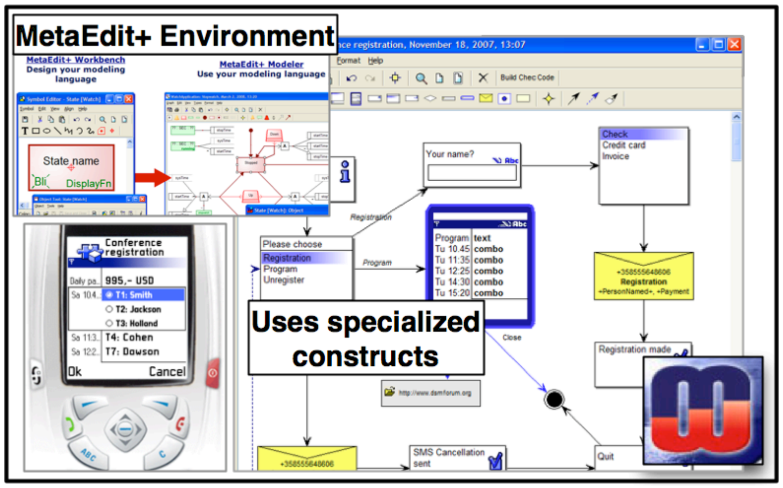
MetaEdit+ Modeler

The *MetaEdit+ Environment* [Meta] (figure 3.8) allows the building of domain-specific modeling tools and generators. It consists of the *MetaEdit+ Workbench* which can be used to build modeling tools without having to write a single line of code. The generated modeler is deployed on to the *MetaEdit+ Modeler* which will contain the constructs defined with the *MetaEdit+ Workbench*. Domain-specific models can then be modeled and full working applications are generated from these models [Tol04]. For our purpose, we will focus on the *MetaEdit+ Modeler* used in the domain of Smartphone applications. The target users for this modeler are people that have no experience in programming Symbian/S60 applications [KT08]. An advantage of using this modeling tool is that users who are already familiar with the target platform can easily identify the different components in the model since it uses familiar constructs specific to that platform.



Methods and Techniques	Programming	
	Scripting	
	Modeling	•
	Visual Programming	
	Programming-by-Example	
	Authoring	
Development Technology	Application Framework	
	IDE	
	Authoring Tool	
	Modeling Tool	•
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	
	Competent	
	Novice	•

(a)



(b)

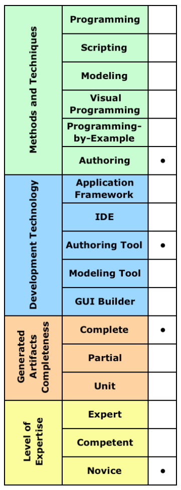
Figure 3.8: The MetaEdit+ Modeler for S60 Applications [Meta][Tol04].

Mobile Author

The *Mobile Author* from Virvou et al. [VA05] (figure 3.9) is an authoring tool that can help teachers create their own Intelligent Tutoring System (ITS). It is a web-based application that can be accessed both through the mobile device or through a computer. The goal of creating ITS is to provide highly individualized guidance to students. Teachers can create different types of test questions (e.g. multiple choice, true or false, fill in the blanks) of which they can add images, texts and links to files for reference. The Mobile Author then generates these data and can be accessed by students via their computers or mobile devices.

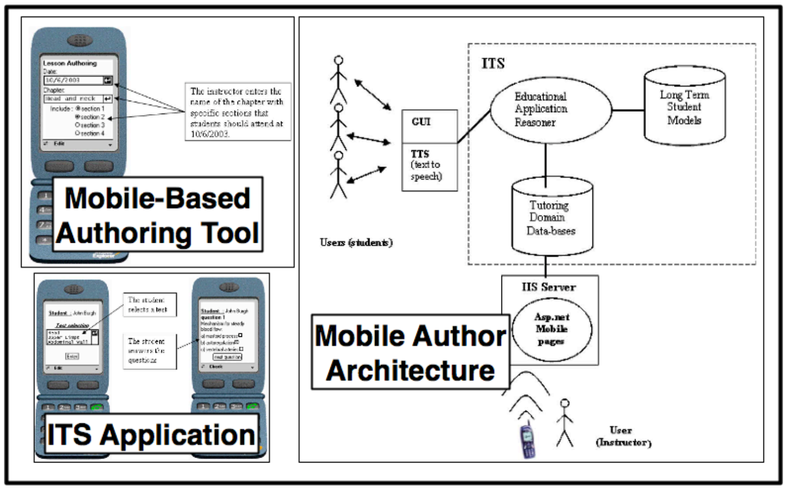
Mobile Bristol

Mediascapes are rich interactive applications containing various forms of multimedia components which are triggered by specific context information (e.g. user's location). The *Mobile Bristol Application Development Framework* from Hull et al. [HCM04] (figure 3.10) allows the creation of ubiquitous applications, specifically *Mediascapes* by non-programmers. It features an authoring environment which is composed of an emulator, publisher, layout editor, point&click editor and media manager. It also has a programmers editor which can be used by more advanced users. The authoring environment generates documents in an XML-based markup language called MBML (Mobile Bristol Markup Language) which describes the application. It is then interpreted by a run-time environment which is composed of an event interpreter, script loader, sensors, user interface and messaging components which runs the application on a mobile device.



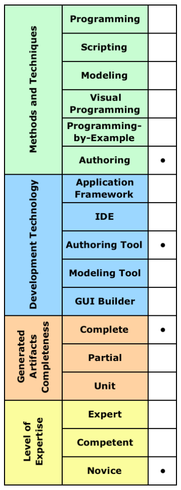
Methods and Techniques	Programming	
	Scripting	
	Modeling	
	Visual Programming	
	Programming-by-Example	
	Authoring	•
Development Technology	Application Framework	
	IDE	
	Authoring Tool	•
	Modeling Tool	
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	
	Competent	
	Novice	•

(a)



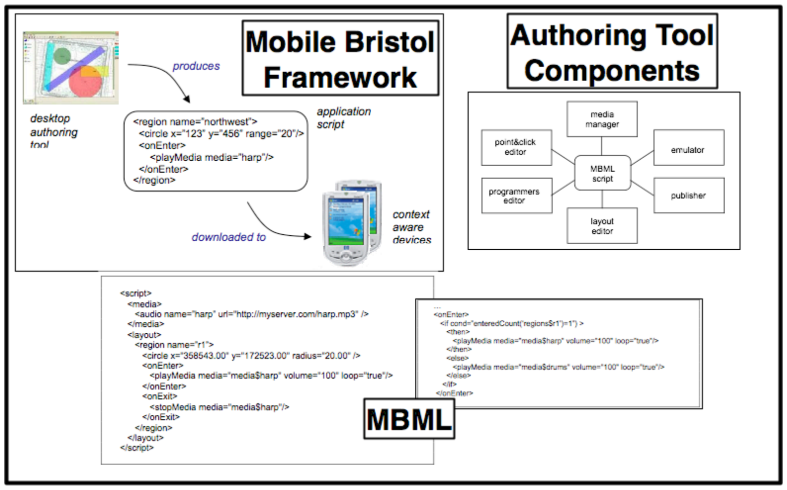
(b)

Figure 3.9: The Mobile Author for creating Intelligent Tutoring Systems for mobile and web [VA05].



Methods and Techniques	Programming	
	Scripting	
	Modeling	
	Visual Programming	
	Programming-by-Example	
	Authoring	•
Development Technology	Application Framework	
	IDE	
	Authoring Tool	•
	Modeling Tool	
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	
	Competent	
	Novice	•

(a)



```
<region name="northwest">
  <circle x="123" y="456" range="20"/>
  <onEnter>
    <playMedia media="harp"/>
  </onEnter>
</region>
```

```
<script>
  <media>
    <audio name="harp" url="http://myserver.com/harp.mp3" />
  </media>
  <layout>
    <region name="r1">
      <circle x="358543.00" y="172523.00" radius="20.00" />
      <onEnter>
        <playMedia media="media$harp" volume="100" loop="true"/>
      </onEnter>
      <onExit>
        <stopMedia media="media$harp"/>
      </onExit>
    </region>
  </layout>
</script>
```

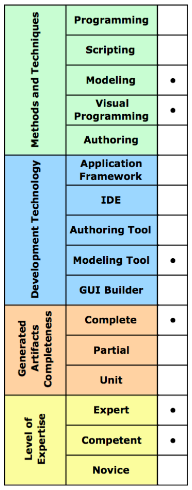
```
...
<onEnter>
  <if cond="enteredCount('regions$r1')=1") >
    <then>
      <playMedia media="media$harp" volume="100" loop="true"/>
    </then>
    <else>
      <playMedia media="media$drums" volume="100" loop="true"/>
    </else>
  </if>
</onEnter>
```

(b)

Figure 3.10: The Mobile Bristol Application Development Framework from Hull et al. [HCM04].

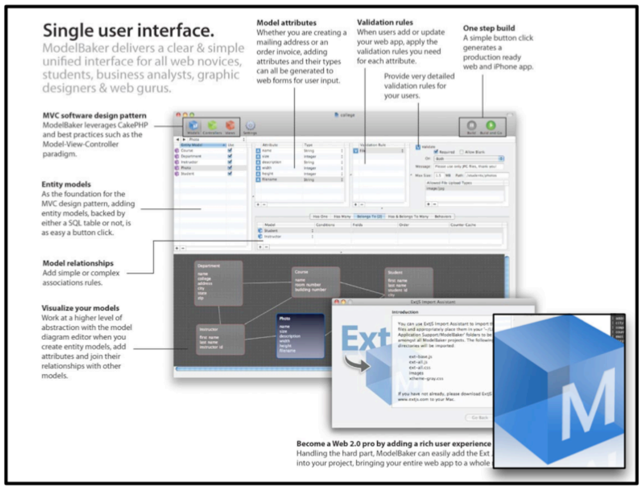
ModelBaker

The *ModelBaker* [Mod] (figure 3.11) is tool that can be used to create customized client web applications for iPhone/iPod and Android platforms. If features template-based and point&click development environment for creating the applications. It uses the MVC (models, views, controllers) design pattern as foundation for building web and mobile applications [Mod]. In using the application though, one must be familiar with the MVC software design pattern, database constructs (e.g. entity models) and web application constructs (e.g. forms).



Methods and Techniques	Programming	
	Scripting	
	Modeling	•
	Visual Programming	•
	Authoring	
Development Technology	Application Framework	
	IDE	
	Authoring Tool	
	Modeling Tool	•
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	•
	Competent	•
	Novice	

(a)



Single user interface.
ModelBaker delivers a clear & simple unified interface for all web novices, students, business analysts, graphic designers & web gurus.

Model attributes
Whether you are creating a mailing address or an order invoice, adding attributes and their types can all be generated to web forms for user input.

Validation rules
When users add or update your web app, apply the validation rules you need for each attribute.

Provide very detailed validation rules for your users.

One step build
A simple button click generates a production ready web and iPhone app.

MVC software design pattern
ModelBaker leverages CakePHP and best practices such as the Model-View-Controller paradigm.

Entity models
As the foundation for the MVC design pattern, adding entity models, backed by either a SQL table or not, is as easy a button click.

Model relationships
Add simple or complex associations rules.

Visualize your models
Work at a higher level of abstraction with the model diagram editor when you create entity models, add attributes and join their relationships with other models.

Become a Web 2.0 pro by adding a rich user experience
Handling the hard part, ModelBaker can easily add the Ext
into your project, bringing your entire web app to a whole

(b)

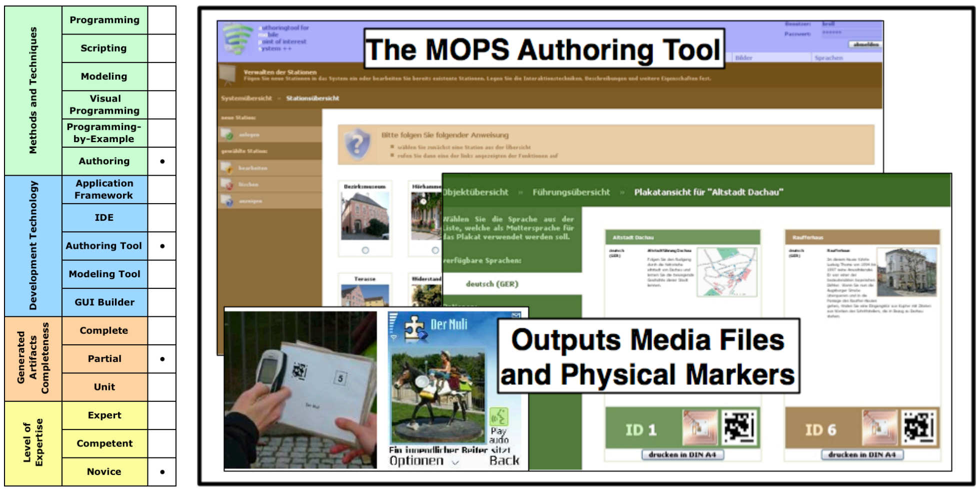
Figure 3.11: The ModelBaker for customized mobile web applications [Mod].

MOPS Authoring Tool

The *MOPS (MObile Petuelpark System)* from Broll et al. [BRW07] (figure 3.12) is *"a mobile application used to provide information to people about exhibits in the Petuelpark in Munich, Germany through physical mobile interactions"*. The *MOPS Authoring Tool* is a web-based application used to create MOPS applications. It helps non-technical users design and model points of interest through the assembly of media files and assignment of physical mobile interactions. The output of the authoring tool is an XML description file, a set of media files and physical markers whose deployment can be left to a developer with more technical skills.

M-Studio

Mobile cinema features discrete cinematic sequences which are delivered based on some context information such as the viewer's location and time. *M-Studio* from Pan et al. [PKCD02]

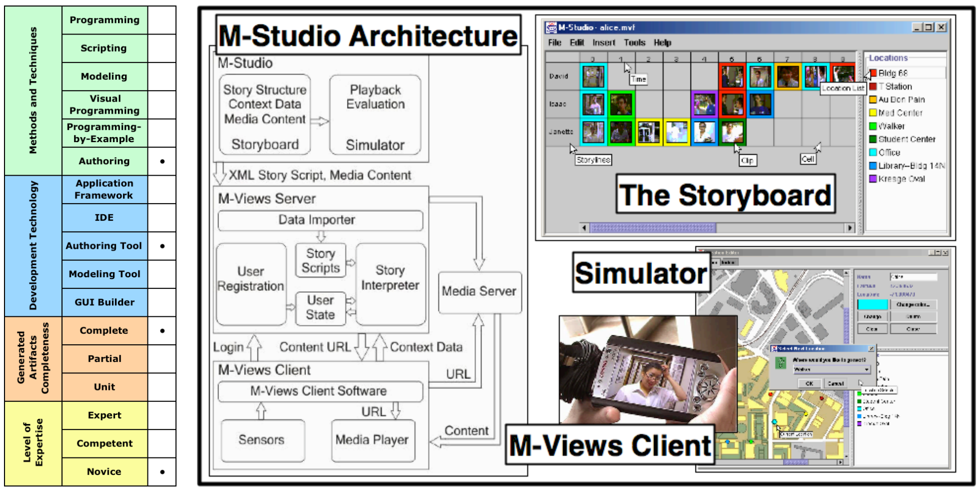


Methods and Techniques	Programming	
	Scripting	
	Modeling	
	Visual Programming	
	Programming-by-Example	
	Authoring	•
Development Technology	Application Framework	
	IDE	
	Authoring Tool	•
	Modeling Tool	
	GUI Builder	
Generated Artifacts Completeness	Complete	
	Partial	•
	Unit	
Level of Expertise	Expert	
	Competent	
	Novice	•

(a) (b)

Figure 3.12: The MOPS Authoring Tool and its outputs [BRW07].

(figure 3.13) is a toolkit which aims to help story creators in the production phase of story development for mobile cinema. Story creators can design, script and simulate multi-threaded context-aware stories with the tool. It consists of components such as a storyboard, location editor, clip editor and simulators.



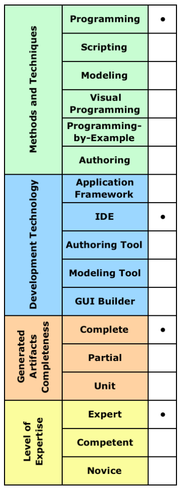
Methods and Techniques	Programming	
	Scripting	
	Modeling	
	Visual Programming	
	Programming-by-Example	
	Authoring	•
Development Technology	Application Framework	
	IDE	
	Authoring Tool	•
	Modeling Tool	
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	
	Competent	
	Novice	•

(a) (b)

Figure 3.13: The M-Studio architecture, story board and simulator [PKCD02].

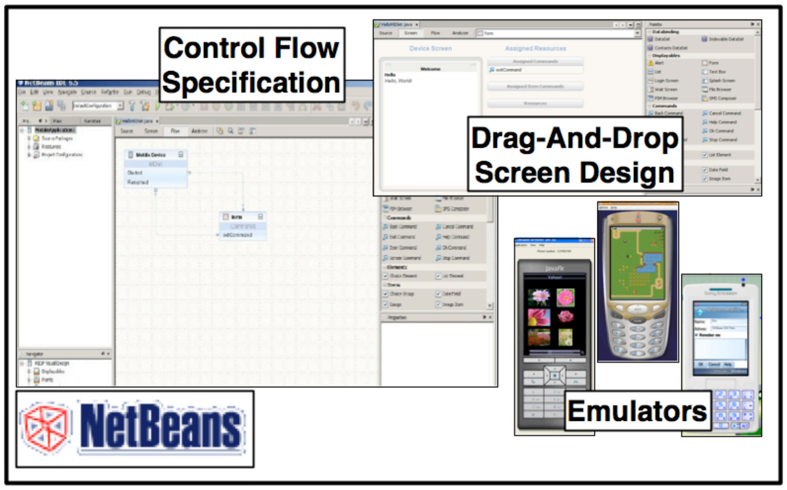
Netbeans Mobility Pack

The *Netbeans mobility pack* [Net] (figure 3.14) is an extension to the *Netbeans IDE* in order to help developers in the development of applications for Java Micro Edition (ME) platforms. Aside from the common features that other IDEs have, there are some features that the *Netbeans mobility pack* possess which gives it an advantage over other IDES. Examples of these features are the following: the ability to add simple application logic by dragging arrows between different screens, the ability to design the user interface through drag-and-drop means, and the provision of various emulators depending on the target device.



Methods and Techniques	Programming	•
	Scripting	
	Modeling	
	Visual Programming	
	Programming-by-Example	
	Authoring	
Development Technology	Application Framework	
	IDE	•
	Authoring Tool	
	Modeling Tool	
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	•
	Competent	
	Novice	

(a)



(b)

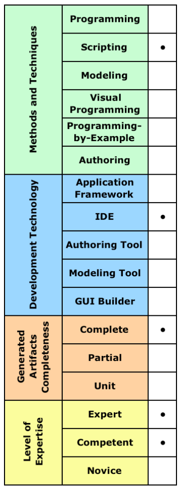
Figure 3.14: The Netbeans Mobility Pack with some of its features [Net].

Project Ares from Palm

Project Ares [Pal] (figure 3.15) is Palm's web-based development environment for creating webOS applications for Palm products. The target users of project Ares are web developers who want to go into mobile development. It features a GUI Editor for designing the screens, a code editor (Javascript) for adding the necessary functions to the application, and a debugger and log viewer. It also allows developers to easily deploy their applications to Palm's App Catalog or on the web.

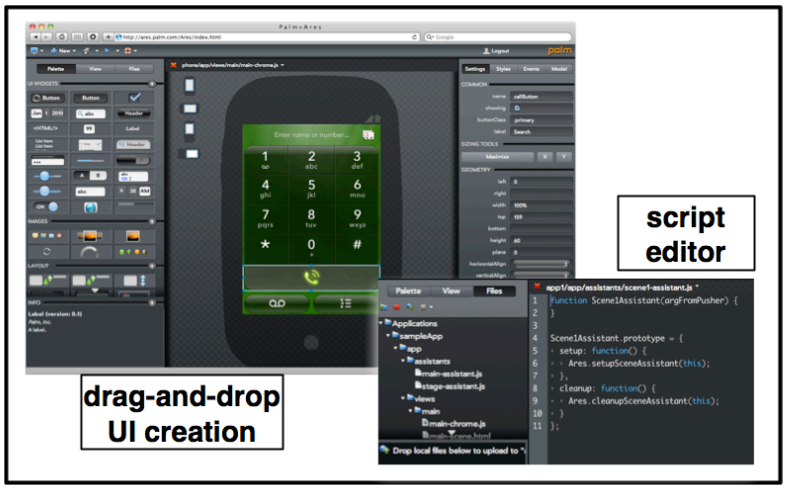
Qt Creator

The Qt Creator [QtM][QtC09] (figure 3.16) is an IDE used for creating applications using the *Qt Application framework* which is a cross-platform application framework that also supports mobile platforms (e.g. S60, maemo Internet tablets, embedded linux and Windows



Methods and Techniques	Programming	
	Scripting	•
	Modeling	
	Visual Programming	
	Programming-by-Example	
	Authoring	
Development Technology	Application Framework	
	IDE	•
	Authoring Tool	
	Modeling Tool	
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	•
	Competent	•
	Novice	

(a)



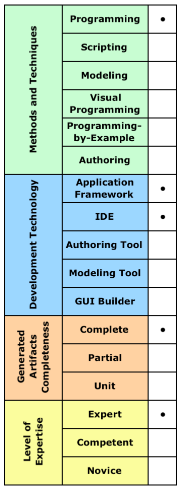
app1/app/assistants/scene1-assistant.js

```
function Scene1Assistant(argFromPusher) {
}

Scene1Assistant.prototype = {
  setup: function() {
    Ares.setupSceneAssistant(this);
  },
  cleanup: function() {
    Ares.cleanupSceneAssistant(this);
  }
};
```

(b)

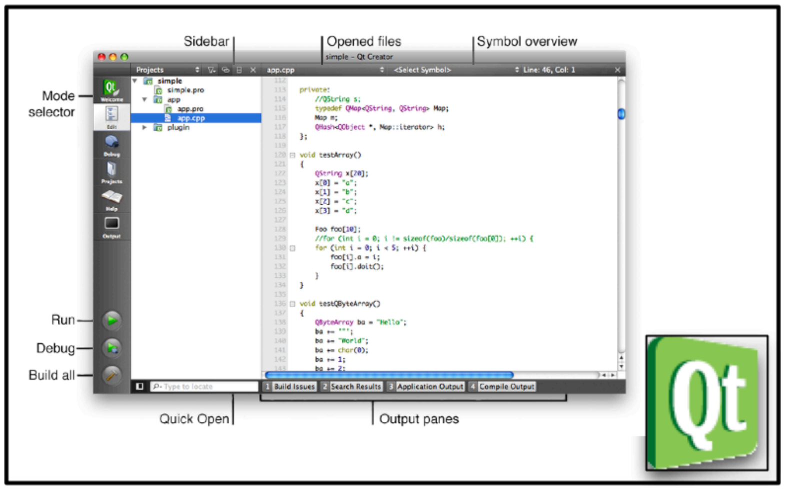
Figure 3.15: Project Ares for webOS applications for Palm handhelds [Pal].

CE). The difference between the *Qt Creator* and *Qt plugins* for existing IDEs (e.g. Eclipse) is that it features Qt-specific constructs in its environment. It also claims that it offers a *"user friendly easy environment to start creating Qt applications"* [QtC09].



Methods and Techniques	Programming	•
	Scripting	
	Modeling	
	Visual Programming	
	Programming-by-Example	
	Authoring	
Development Technology	Application Framework	•
	IDE	•
	Authoring Tool	
	Modeling Tool	
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	•
	Competent	
	Novice	

(a)



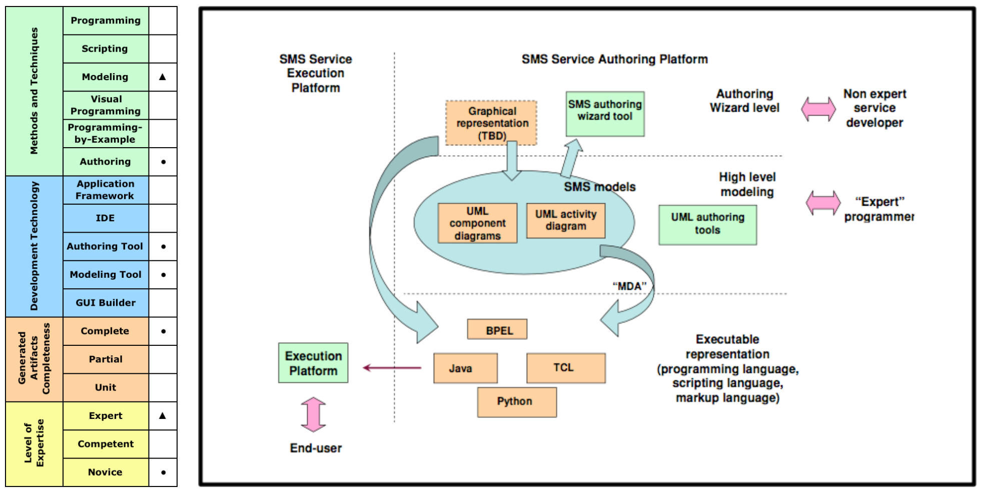
```
private:
    //QString s;
    typedef QMap<QString, QString> Map;
    Map m;
    QHash<QObject *, Map::iterator> h;
};

void testArray()
{
    QString x[20];
    x[0] = "a";
    x[1] = "b";
    x[2] = "c";
    x[3] = "d";

    Foo foo[10];
    //for (int i = 0; i != sizeof(foo)/sizeof(foo[0]); ++i) {
    for (int i = 0; i < 5; ++i) {
        foo[i].a = i;
        foo[i].doit();
    }
}

void testQByteArray()
{
    QByteArray ba = "Hello";
    ba += "";
    ba += "World";
    ba += char(0);
    ba += 1;
    ba += 2;
```

(b)

Figure 3.16: The Qt Creator [QtM][QtC09].

SMS Service Authoring Tool

The *SMS Project* (figure 3.17) aims to create innovative tools that allows individuals and small businesses to become creators and providers of simple mobile services. Two different types of users that the SMS Project wants to address are the *expert user/developer* and *non-expert service developer*. The *expert user/developer* will use modeling tools (e.g. MagicDraw, AndroMDA) in order to create SMS models, while the *non-expert service developer* will use authoring tools to create the services. One example of such authoring tool is the *SMS Service Authoring Tool*.



(a) (b)

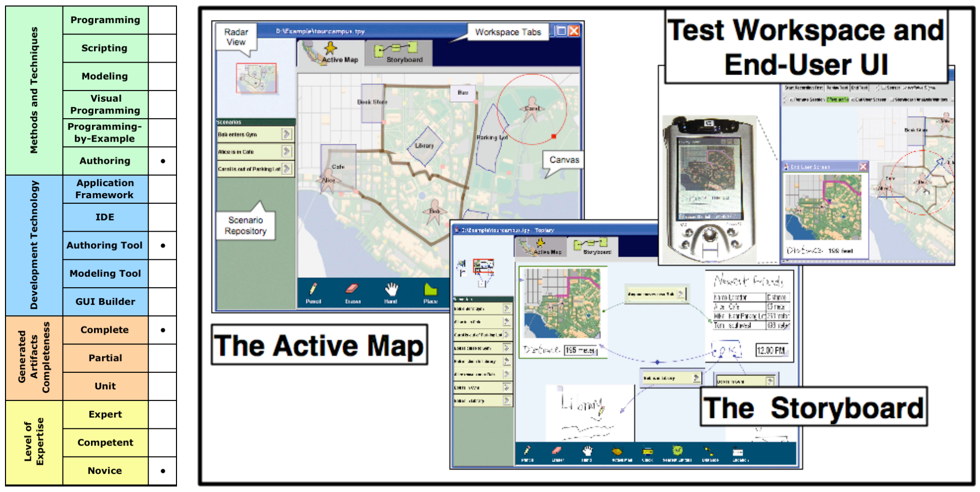
Figure 3.17: The following diagram taken from [BMSBM07][BMC$^+$06] shows the models, translations and tools for Service Authoring.

Topiary

Topiary from Li et al. [LHL04] (figure 3.18) is a tool that aims to help designers in easily creating prototypes of location-enhanced applications for handheld devices. It features an *Active Map* workspace in which spatial relationships between people, places and things can be modeled. The *Storyboard* workspace on the other hand helps create possible scenarios and storyboards for the application. The scenarios are a collection of location contexts that can be used to specify location-enhanced application behavior. It also features a *Test Workspace* that allows the designers to simulate location contexts, and also allows them to see what is viewed on the end-user UI which is run on a device running the Topiary client.

iPhone Development Using XCode and Interface Builder

XCode (figure 3.19) is an integrated development environment that also supports the development of applications for iPhone/iPod/iPad platforms. Aside from the common IDE features, it also provides an emulator for testing applications. In order to aid the design and testing of user interfaces, Apple also provides the *Interface Builder* which contains components for designing interfaces for the iPhone/iPod/iPad. *"Interface Builder works*

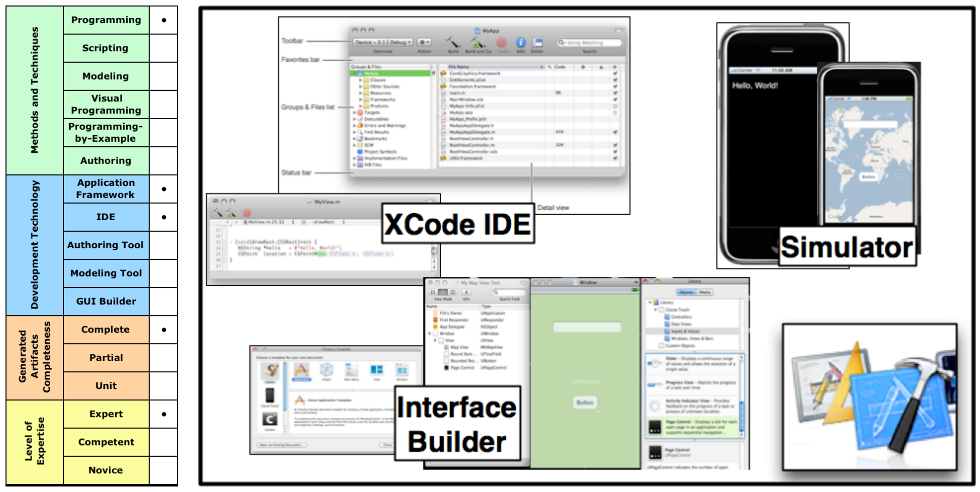


Methods and Techniques	Programming	
	Scripting	
	Modeling	
	Visual Programming	
	Programming-by-Example	
	Authoring	•
Development Technology	Application Framework	
	IDE	
	Authoring Tool	•
	Modeling Tool	
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	
	Competent	
	Novice	•

(a) (b)

Figure 3.18: Topiary's active map and storyboard workspaces [LHL04].

closely with Xcode to provide a development experience that facilitates the concurrent but specialized development of an application's user interface and business logic" [Appc].



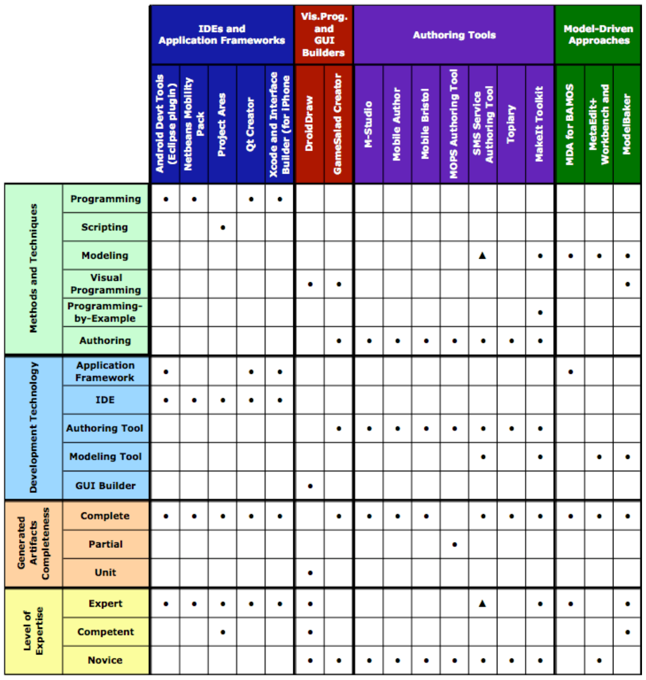
Methods and Techniques	Programming	•
	Scripting	
	Modeling	
	Visual Programming	
	Programming-by-Example	
	Authoring	
Development Technology	Application Framework	•
	IDE	•
	Authoring Tool	
	Modeling Tool	
	GUI Builder	
Generated Artifacts Completeness	Complete	•
	Partial	
	Unit	
Level of Expertise	Expert	•
	Competent	
	Novice	

(a) (b)

Figure 3.19: The XCode IDE, Interface Builder and iPhone emulator from Apple [Appc].

3.1.3 Comparison of Systems for Mobile Application Development

In the previous section, different systems that allowed the creation of software artifacts for mobile platforms were shown. In this section, we will look at the advantages and disadvantages of the different approaches. Approaches with similarities in terms of the development technologies are grouped together (see figure 3.20).



		IDEs and Application Frameworks					Vis.Prog. and GUI Builders		Authoring Tools							Model-Driven Approaches		
		Android Devt Tools (Eclipse plugin)	Netbeans Mobility Pack	Project Ares	Qt Creator	Xcode and Interface Builder (for iPhone	DroidDraw	GameSalad Creator	M-Studio	Mobile Author	Mobile Bristol	MOPS Authoring Tool	SMS Service Authoring Tool	Topiary	MakeIt Toolkit	MDA for BAMOS	MetaEdit+ Workbench and	ModelBaker
Methods and Techniques	Programming	•	•		•	•												
	Scripting			•														
	Modeling												▲		•	•	•	•
	Visual Programming						•	•										•
	Programming-by-Example														•			
	Authoring							•	•	•	•	•	•	•	•			
Development Technology	Application Framework	•			•	•										•		
	IDE	•	•	•	•	•												
	Authoring Tool							•	•	•	•	•	•	•	•			
	Modeling Tool												•		•		•	•
	GUI Builder						•											
Generated Artifacts Completeness	Complete	•	•	•	•	•		•	•	•	•		•	•	•	•	•	•
	Partial											•						
	Unit						•											
Level of Expertise	Expert	•	•	•	•	•	•						▲		•	•		•
	Competent			•			•											•
	Novice						•	•	•	•	•	•	•	•	•		•	

Figure 3.20: Summary of features of all the systems discussed in section 3.1.2.

Integrated Development Environments (IDEs) and Application Frameworks

Advantages. Integrated Development Environments combined with Application Frameworks (e.g. *Android Devt. Tools* [Andc], *Netbeans Mobility Pack* [Net], *XCode and Interface Builder* [Appc], *Project Ares* [Pal], *Qt Creator* [QtM]) have the advantage of having the capability to create *fully functional mobile applications (i.e. completely generated artifacts).* Most IDEs also feature a complete set of tools such as debuggers, design tools, emulators, etc. that can assist developers in creating applications.

Disadvantages. The feature-richness in the development environments however can also pose as a disadvantage. It can take a while before the user can actually take advantage of the many features the environment has to offer and also affects the usability of such tools [BFH09b]. Also, programming skills and the willingness to learn how to use the different capabilities of these IDEs is a big necessity.

Visual Programming Tools and GUI Builders

Advantages. Systems that provide a visual programming environment such as *GameSalad* [Gam]) and Graphical User Interface (GUI) Builders such as *DroidDraw* [Dro] use the *power of images* in order to create fully functional programs. These systems usually target a specific domain (e.g. *GameSalad* [Gam] for game applications) or a specific platform (e.g. *DroidDraw* [Dro] for Android applications), thus utilize graphical representations that are close to the domain or similar to the platform they are targeting. According to Petre [Pet95], such visual programming techniques are only effective when the specific representations support the conventions expected by the users of such tools, which means this representation poses as an advantage to users who are already familiar with the domain or platform.

Disadvantages. A disadvantage of using these approaches is that, the users should be able to understand what the purpose of such graphical representations are (i.e. it should not be ambiguous or interpreted differently by different users) [Pet95].

Another disadvantage specific to GUI Builders is that, it only allows a part of an application to be created, specifically the user interface. Application logic needs to be added by explicit programming. Visual Programming Environments on the other hand may provide fully-functional applications (e.g. *GameSalad* [Gam] for game creation). However, it requires very specific knowledge and uses domain-specific constructs that are not applicable for other domains or application types.

Authoring Tools

Advantages. Most authoring tools also make use of images to represent certain parts of the system. Just like visual programming approaches, authoring tools are also very specific to one type of domain. Just to cite some examples: *M-Studio* [PKCD02] usually targets story creators for creating applications for mobile cinema, *Mobile Author* [VA05] is for teachers who want to create ITS systems, *Mobile Bristol* [HCM04] targets artists to create ubiquitous applications specifically mediascapes, and so on. *Targeting a specific domain* however, can be seen as an advantage especially if the target users of these tools are novices. The design of these tools are usually simpler because it will contain a limited number of components specific to one type of domain than those that support development for general purpose applications. Another advantage of the systems mentioned is that they *take usability into account* when designing and creating the authoring tools. For the examples mentioned, usability evaluations were usually carried out in order to refine their designs and allow non-technical users to easily use them.

Disadvantages. Most authoring tools however have the disadvantage of not being flexible enough to be extended to other domains, such as the example systems cited in the previous section. Some approaches provide some framework to extend the capabilities of the tools, but still on one specific type of domain. Another disadvantage of some of tools is that the user of the authoring tool still needs to coordinate with programmers/technical staff in order to create the fully functional applications (e.g.

The MOPS Authoring Tool [BRW07] still needs developers in order carry out the final deployment).

Model-Driven Approaches

Advantages. The model-driven approaches we have seen in the examples so far varies in terms of the types of users they target and the tools they provide. One common denominator between the different approaches is the *use of models as primary artifacts* in the development of an application. These models have their formal definition in order to run automatic code generation tools to transform the models into code [SVC06]. The *ModelBaker* for example is created in order to assist users who know a lot about web development in the rapid development of web applications through modeling. The *MetaEdit+ Workbench* [Meta] on the other hand targets two types of users, the expert (i.e. developers, modelers) who creates the modeling tools using the workbench, and the novice who uses the modeling tools created by the expert. An advantage of these modeling approaches is the *use of formal models and standardized technologies* (e.g. System from Dunkel et al. [DB07], SMS Project [BMSBM07]). Also, the ability to transform models to different software artifacts for different platforms is another advantage [KWB03].

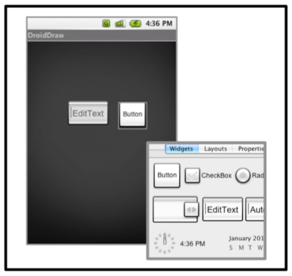
Disadvantages. A disadvantage observed from the different model-driven approaches is that, they do not seem to take the end-user into account when designing the tools. The system from Dunkel et al. [DB07] for example uses a UML modeling tool for modeling the system and other separate tools for code generation. Dunkel et al. [DB07] even emphasized in their conclusion that *"most of the UML modeling tools do not satisfactorily support meta-modeling and the code generating tools are still proprietary and not yet stable"*. The approach is still quite fragmented in a way. Also, knowledge about UML constructs is a must for modeling the applications. The *ModelBaker* [Mod] that also claims ease-of-use still provides a complex interface in a way that one must know the MVC design pattern, database constructs and web-specific constructs in order to be able to create a model. Since the target users of this system are people with experience in web development, the design of the interface should work for them. However, in terms of suitability for other types of users particularly non-technical ones, this is not a very effective design.

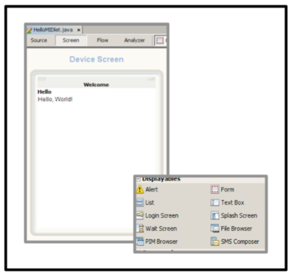
3.1.4 Representation of Mobile Application Constructs

During the design phase of the *Mobia Modeler*, an exploration of existing tools used for mobile application development was carried out. The purpose of this activity was to extract common constructs used in these systems to represent parts of a mobile application.

User Interface

Among the different tools [Net][Dro][Meta] explored, the concept of representing a current instance of an application as a *screen* is being used. The *screen* acts as a blank canvas where different user interface (UI) elements can be placed.


(a) The DroidDraw [Dro] UI designer


(b) The Netbeans Mobility Pack [Net]

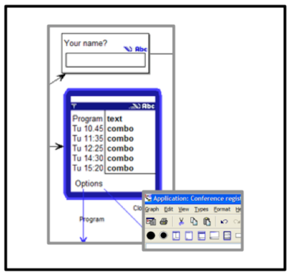

(c) The MetaEdit+ Modeler [Meta]

Figure 3.21: Examples of how user-interface is created using the surveyed tools. Most of them use the concept of screens to represent a certain screen instance of an application. (a) and (b) shows concrete representations of the user interface elements, while (c) has a more abstract representation of the user interface elements.

User interface elements on the other hand are shown in two different ways: as *concrete elements* or *abstract elements* (figure 3.21). *Concrete elements* are shown as they would appear in the application such as the ones in the Netbeans with Mobility Pack Environment [Net] and DroidDraw Environment [Dro]. *Abstract elements* are shown as abstract representations and act as placeholders where the actual UI element would appear on the screen such as the ones in the MetaEdit Modeling Environment [Meta].

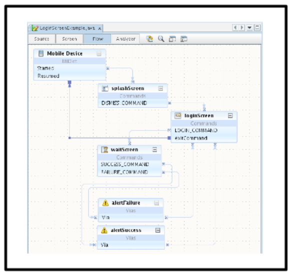
Simple Control Logic: Representing Application Flow

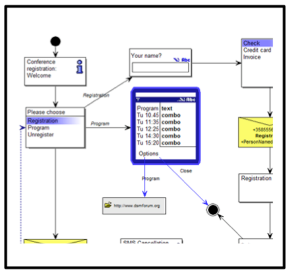
A common way to represent application flow is the use of directed graphs such as the ones used in the Netbeans with Mobility Pack Environment [Net], the MetaEdit Modeling Environment [Meta] and Topiary [LHL04] (figure 3.22). The nodes of the graph are represented by a symbol in the application such as a screen, and are then connected by directed lines to signify transition from one state of the application to the next.

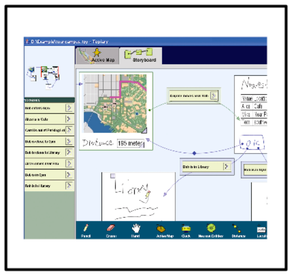
One example from the work of Pan et al. [PKCD02] (figure 3.22d) differs from the graph-like representation of application flow. Instead, they used a tabular form in which the columns represent time, and the rows represent the different story lines [PKCD02].

Complex Control Logic: Representing Inputs and Outputs

A challenging part of the survey was extracting ideas on how inputs and outputs to an application are represented. This still deals with representing control logic in an application,


(a) The Netbeans Mobility Pack [Net]


(b) The MetaEdit+ Modeler [Meta]


(c) The Topiary Storyboard [LHL04]

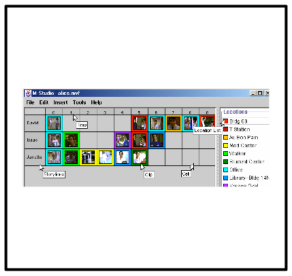
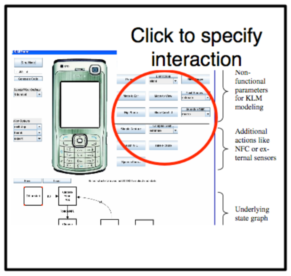

(d) The MStudio Storyboard [PKCD02]

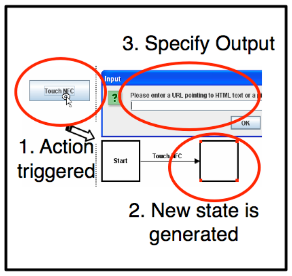
Figure 3.22: Examples of how control logic is represented in the surveyed tools. For tools (a), (b) and (c), simple logic can be added by connecting elements. (c) differs since logic is added in a tabular form, where each column represent a certain time in the story, and parallel rows represent storylines [PKCD02].

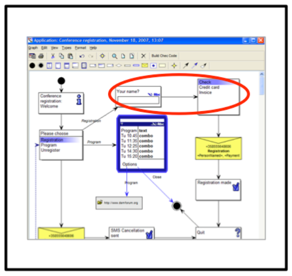
however, additional information such as parameters (i.e. inputs) to a certain condition must be supplied, and certain actions (i.e. outputs) must be taken when the condition is met.

Inputs represent any computational object that provides information to the mobile application. An example would be information taken from a database, or information collected from other devices (e.g. sensors, other mobile devices).

Outputs represent any action that a mobile application carries out as a result of some condition. An example would be sending an SMS message to remind someone of a deadline, or vibrating to alarm or remind the mobile user about something he has to do.


Click to specify interaction
Non-functional parameters for KLM modeling
Additional actions like NFC or external sensors
Underlying state graph
(a) The MakeIt toolkit [HS08] (Input)


3. Specify Output
Touch NFC
Input
Please enter a URL pointing to HTML text or a
OK
1. Action triggered
Start
Touch NFC
2. New state is generated
(b) The MakeIt toolkit [HS08] (Output)


(c) The MetaEdit+ Modeler [Meta] (Input)

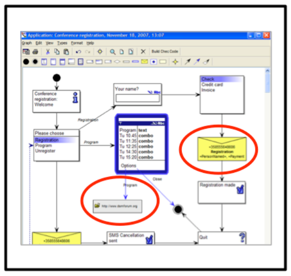

(d) The MetaEdit+ Modeler [Meta] (Output)

Figure 3.23: (a) Inputs are collected by clicking on specific buttons shown in the toolkit's interface (e.g. touch NFC, Take Picture). (b) Outputs of a certain action are inputted via some dialog box. (c) Inputs such as text inputted by a user is represented by an input box. (d) Outputs such as sending a message or going to a web page are specified with special symbols.

In the MakeIt toolkit [HS08] for example, *inputs* can be any information collected using a specific interaction technique such as touching NFC or RFID tags. Inputs are added to the application by clicking on the buttons to simulate the interactions (figure 3.23a). The resulting *outputs* of a certain action can then be added to some text box such as a URL which will be invoked (figure 3.23b).

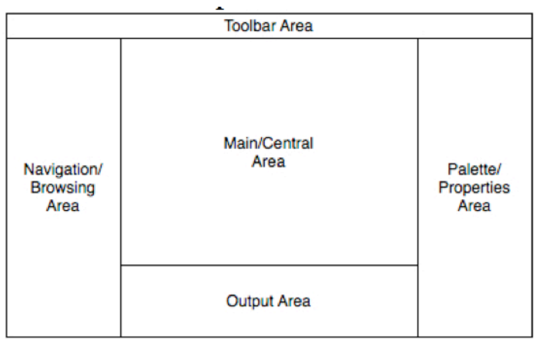
In the MetaEdit+ Modeler [Meta], *inputs* can be any information that can be typed in inside a textbox (figure 3.23c). *Outputs* are represented by special model symbols such as an envelope symbol to represent sending a message (figure 3.23d).

3.1.5 User Interface Design Features

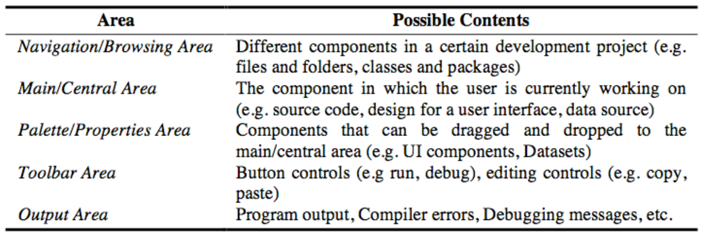
Aside from the common mobile constructs used, an exploration of the design features from different applications ranging from development tools to commonly used software applications was carried out. The purpose of this activity was to look into the specific design features these applications offer that made them attractive and more usable.

Common Layout for Development Tools

In the different development environments surveyed, a common layout is present in this type of application. These tools usually comprise of four basic areas: the navigation/browsing area, the main/central area, the palette/properties area, and the toolbar area [BFH09b]. Figure 3.24 shows those areas and a list of possible contents.



The different areas and their possible contents



Area	Possible Contents
Navigation/Browsing Area	Different components in a certain development project (e.g. files and folders, classes and packages)
Main/Central Area	The component in which the user is currently working on (e.g. source code, design for a user interface, data source)
Palette/Properties Area	Components that can be dragged and dropped to the main/central area (e.g. UI components, Datasets)
Toolbar Area	Button controls (e.g run, debug), editing controls (e.g. copy, paste)
Output Area	Program output, Compiler errors, Debugging messages, etc.

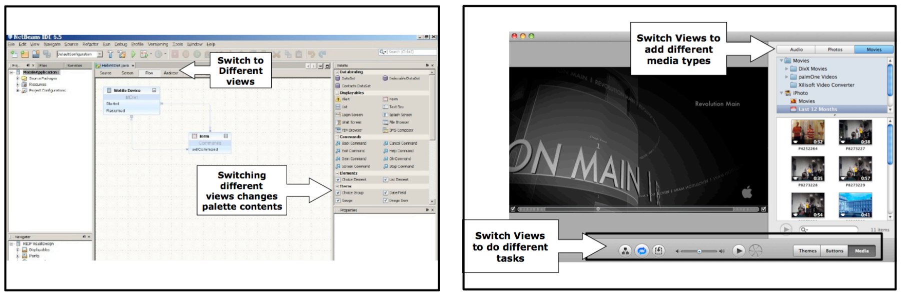
Figure 3.24: The common areas and layout of the different development tools [BFH09b].

Task Separation through Modes

Aside from layout, some applications also offer a way to separate different tasks or information through modes.

In Netbeans [Net] for example, one can switch to different views/modes by clicking on the tabs. The *source view* allows the user to make changes to the source code; the *screen view* allows drag and drop design of the mobile application's user interface; the *flow view* allows adding logic to the program by dragging flow arrows between the different screens; and the *analyzer view* shows unused resources and MIDP compliancy. Switching through the different views changes the contents of the palette area, depending on what components are needed in that certain view [BFH09b].

In iDVD[6] clicking on the different buttons (Themes, Buttons, Media) sets what task the user needs to do. Clicking another set of buttons changes mode depending on the type of content that can be added to the project (e.g. audio, photos, movies).


Switch to Different views
Switching different views changes palette contents
Switch Views to add different media types
Switch Views to do different tasks

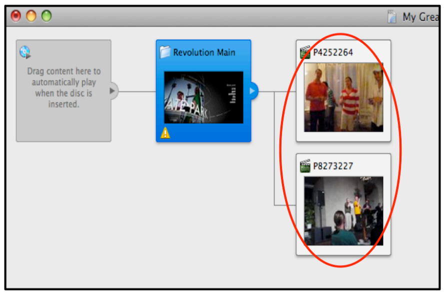
(a) **(b)**

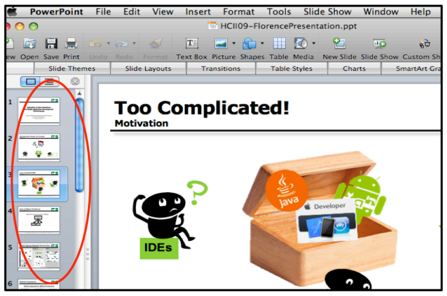
Figure 3.25: (a) Changing modes in Netbeans [Net]. (b) The different modes in iDVD from Apple.

Simplifying Tasks through Previews

Some applications provide previews in order to simplify tasks. One example is the presentation tool Microsoft Powerpoint[7] in which thumbnails of the slides are shown on the side in order to allow the user to easily access the different parts of the presentation by just clicking on them. Another example is the iDVD [8] application which provides previews of the different media added to the DVD project.

[6]http://www.apple.com/ilife/idvd/
[7]http://office.microsoft.com/en-us/powerpoint/default.aspx
[8]http://www.apple.com/ilife/idvd/


Drag content here to automatically play when the disc is inserted.
Revolution Main
P4252264
P8273227
(a)



(b)

Figure 3.26: (a) iDVD offers a preview of the different media added to the project. (b) Microsoft Powerpoint provides preview of the slides on the side panel.

3.1.6 Usability Evaluation Overview

According to Paterno [Pat99], usability is not just concerned with making systems *easy to learn* and *easy to use* but also includes the following:

- The system's relevance in serving the users' needs.
- The efficiency of how users carry out their tasks when using the system.
- The users' feelings or attitudes towards the system.
- The ease of learning the system especially during initial use.
- The system's tolerance to unexpected or wrong usage.

In order to measure the system's usability, it is only natural to initially focus on the user interface in which the user is directly in contact with [DFAB93]. In the case of this research, the *Mobia Modeler* which serves as the front end of the *Mobia Framework* is the one subjected to usability evaluation.

Paterno [Pat99] has given some examples of factors which can be used to measure usability such as: user performance on specified tasks (e.g. measured by getting the task completion rate, no. of errors, etc.), user's subjective preference or degree of satisfaction, learnability (e.g. measured by task completion rate, use of documentation) and flexibility.

There are different types of evaluation methods which can be employed in order to test the usability of a system. Rosson et al. [RC02] divides the types of evaluation into: analytic methods, empirical methods and mediated evaluation.

Analytic methods are characterized by analysis and interpretation of the different features and tradeoffs of the system. During the early stages of analysis, the usability engineer tries to study the features of the system and generates some hypotheses about these features. The advantage of doing such methods is the cost since compared to empirical studies, it is relatively more economical to carry out. A problem with this though is that *"the quality of the inspection mostly depends on the skills and biases* *of the analyst"* [RC02]. Examples of analytic methods are usability inspections and model-based analysis such as GOMS (goals,operators,methods,and selection rules) analysis.

Empirical methods are done by involving the actual users and observing them while they interact with the system [RC02]. Other types of data can be collected from the users (e.g. feedback through questionnaires, log data) during the study. Although empirical methods are considered as the gold standard for usability evaluation, they can be expensive to carry out and some problems may still go unidentified despite the studies [Pat99]. Another challenge is the difficulty of getting the appropriate users for the studies [Pat99] plus the problems of different skills and experiences of the users which leads to variability in the results [RC02]. That is why it is important to have a large enough set of users to be able to get some general pattern [RC02]. Also, in times where it is difficult to find participants for the studies who are the actual target users of a specific system, it is important to find an alternative set of people that have similar characteristics, experiences and skill levels as the target users of the system [RC02]. Examples of empirical methods are field studies and laboratory usability testing.

A mediated evaluation is a mixture of both analytic and empirical methods. Analytic methods are used during the early stages of the design of the system, and empirical methods can then be employed in order to focus on the problem areas that were discovered during the analytic study [RC02]. This type of evaluation was carried out for the usability evaluation of the *Mobia Modeler* prototypes.

3.2 The *Mobia Modeler*: A Tool for EUD of Mobile Applications

In this section, a detailed look into development of the different *Mobia Modeler* prototypes will be presented. This includes a discussion on the designs, features and results of the studies conducted that influenced the design of the succeeding versions. Before discussing all of the things mentioned, a brief overview of the *Mobia Modeler* will initially be presented.

3.2.1 The *Mobia Modeler* Concept

The *Mobia Modeler* is a tool that aims to allow non-technical users (i.e. non-programmers) develop their own mobile applications easily. The *Mobia Modeler* tries to combine the different advantages of different systems discussed in section 3.1.3, particularly:

- **Domain-Specificity.** As we have seen with the different types of authoring tools and DSM systems which were presented in section 3.1.2, creating a system that is *specific to a domain* allows users to easily create their own applications. This may be a product of the user's familiarity with the domain, and/or because of the limited number of domain-specific constructs used. This can be seen from the result of the user studies conducted in order to evaluate authoring systems such as the work from Pan et al. [PKCD02] for mobile cinema, Hull et al. [HCM04] for mediascapes, and Li et al. [LHL04] for location-enhanced applications. Other examples are domain-specific modeling tools such as the MetaEdit+ Modeler [Meta] and ModelBaker [Mod].

- **Model-Driven and Platform-Independent Outputs.** Model-driven approaches have the advantage of *using models and standardized technologies* and the ability to transform these models to different software artifacts for different platforms [KWB03]. The *Mobia Modeler* follows the same concept in which a platform-independent graphical model can be made using the modeling tool, and an underlying processor can be used to transform this model to code for different supported platforms.

- **Follow an Iterative User-Centered Design and Development.** The user-centered iterative design and development of software applications ensures that the design and features match the expectations of potential end-users. Activities such as surveys to collect information on what users want in an application, and user studies to evaluate the usability of these applications are examples of UCD activities that were carried out in the development of the *Mobia Modeler*.

3.2.2 The *Mobia Modeler* Prototypes: An Overview

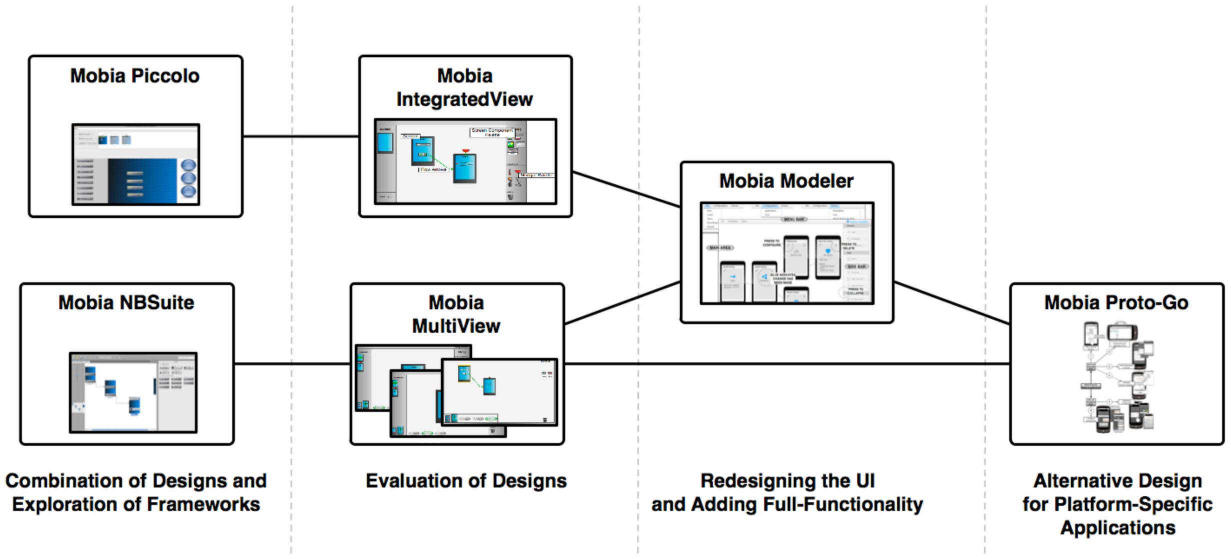

Mobia Piccolo
Mobia IntegratedView
Mobia Modeler
Mobia NBSuite
Mobia MultiView
Mobia Proto-Go
Combination of Designs and Exploration of Frameworks
Evaluation of Designs
Redesigning the UI and Adding Full-Functionality
Alternative Design for Platform-Specific Applications

Figure 3.27: The different *Mobia Modeler* prototypes connected by lines to signify the influence of one prototype to the succeeding versions. The goals of creating the different prototypes are also shown below the figures.

In figure 3.27, the different *Mobia Modeler* prototypes developed throughout the duration of this research are shown. The lines connecting the different prototypes show the influence of a prototype to the succeeding ones. This influence will be elaborated as we discuss the individual prototypes in the succeeding sections.

For discussion purposes, the different prototypes are assigned different names as to prevent confusion when comparing one prototype to the next. The latest version of the prototype is the one named *Mobia Modeler*.

The goal of developing the first batch of prototypes which are called the *Mobia Piccolo* and the *Mobia NBSuite*, was to combine different designs and functionality based on the results of the survey with potential users (Chapter 2), and based on the different surveyed systems discussed in the previous sections (sections 3.1.2 and 3.1.5). Another aim is to explore possible frameworks to use as base framework for the prototypes.

The design features of the *Mobia Piccolo* and the *Mobia NBSuite* were then reorganized and made into two distinct versions of the tool called the *Mobia Integrated-View* and *Mobia Multi-View*. The goal of creating these two versions was to evaluate which of the designs offer a more usable interface. The resulting findings from this evaluation led to the redesign of the *Mobia Modeler*. In this version of the *Mobia Modeler*, the corresponding *Mobia Processor* which allowed full code generation from the models was also developed. Details about the *Mobia Processor* will be discussed in the next chapter.

Finally, a supplementary tool called the *Mobia Proto-Go* was designed and develop for the purpose of addressing the limitations of the *Mobia Modeler*. The *Mobia Proto-Go* differs from the *Mobia Modeler* in a way that it uses constructs specific to a target platform and runs on the target mobile platform rather than on a Personal Computer.

3.2.3 The *Mobia Modeler* Trial Prototypes: Combination of Designs and Exploration of Frameworks

Research Goal

The *Mobia Piccolo* and *Mobia NBSuite* were developed for the purpose of combining the different features that were collected from potential users based on the surveys conducted, and combine different features and designs based on the different systems explored. This was also to explore possible base frameworks for the *Mobia Modeler*.

Implementation

The *Mobia Piccolo* was implemented using the Java Framework as the base framework with additional APIs from the Piccolo Graphics Framework particularly Piccolo2D.Java[9] which abstract low level graphics code.

[9]http://www.piccolo2d.org/learn/index.html

The *Mobia NBSuite* was implemented using the Java Framework. However, it uses the Netbeans[10] Module Suite which allows the development of client applications on top of the Netbeans platform[11].

Design Details

Modeling the User Interface. For both *Mobia Piccolo* and *Mobia NBSuite*, a mobile application's user interface (UI) is designed by dropping abstract representations of the user interface elements onto a screen (figure 3.28).

The reason for not using concrete representations (i.e. what they actually look like on the device) of the UI elements is because we want the tool to be used to create models for different target mobile platforms. Featuring UI elements in the tool that are specific to a platform might give the impression that it is only applicable to that platform.

Setting the properties of the UI element (e.g. labels for buttons) was not possible for both versions since the focus for this phase was on the overall design of the tool.

Modeling the Application Flow. In order to discuss the difference between adding application control flow between the two versions, the layout of the navigation area and the steps on how screens are added to the model will be presented first.

- **Navigation Area Layout.** For the *Mobia Piccolo*, the navigation area is located on top of the modeler. It shows a preview of the screens that are present in the model. One can view a certain screen by clicking on the screen in the navigation area, and that screen would appear in the UI design area located at the center of the modeler.

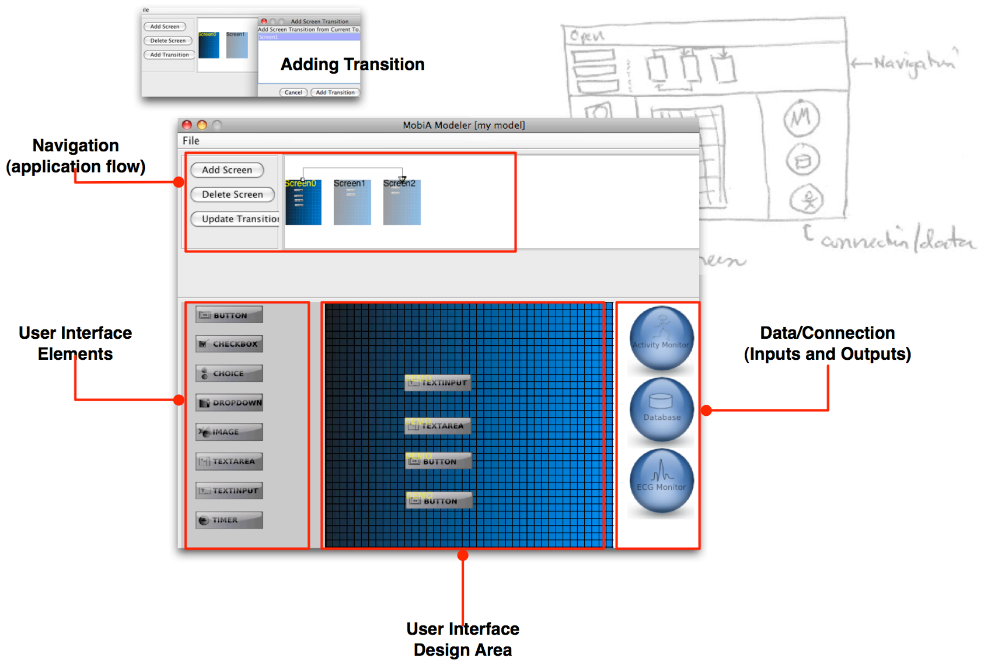
  For the *Mobia NBSuite*, the navigation area and the UI design area are incorporated into one location. In cases where there are many screens in the model, one can just zoom in and out, or drag the small box around in the zooming area to view a particular part of the model.

- **Adding Screens.** For the *Mobia Piccolo*, adding screens to the model is done by pressing on the *Add screen* button on the small panel at the left side of the navigation area.

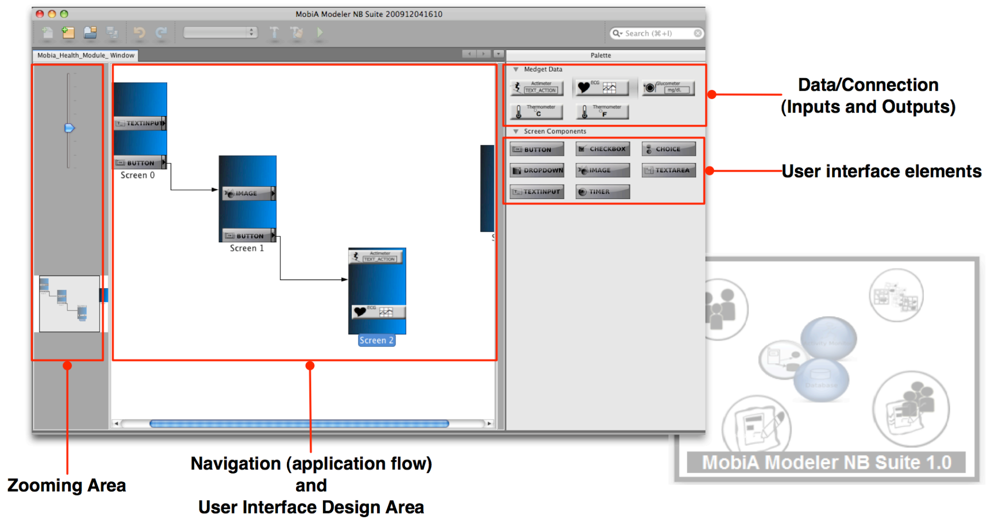
  For the *Mobia NBSuite*, adding screens is done by simply clicking on a blank part of the navigation/design area.

- **Adding Application Flow.** For the *Mobia Piccolo*, to add flow of control from one screen to the next, the button *Add/Update transition* is pressed. A dialog will then appear in which, the user can choose the target screens for the transition.

[10]http://netbeans.org
[11]http://wiki.netbeans.org/DevFaqAppClientOnNbPlatformTut



(a) *Mobia Piccolo*


MobiA Modeler NB Suite 200912041610
Mobia_Health_Module_Window
Palette
Medget Data
Screen Components
BUTTON
CHECKBOX
CHOICE
DROPDOWN
IMAGE
TEXTAREA
TEXTINPUT
TIMER
Screen 0
Screen 1
Screen 2
Data/Connection (Inputs and Outputs)
User interface elements
MobiA Modeler NB Suite 1.0
Zooming Area
Navigation (application flow) and User Interface Design Area

(b) *Mobia NBSuite*

Figure 3.28: The *Mobia Piccolo* and *Mobia NBSuite* prototypes.

For the *Mobia NBSuite*, adding flow of control is done by first adding a user interface element in the screen which can trigger a transition (e.g. button) and then drag the arrow from this element to the target screen.

For both versions, there is no part wherein one can specify certain conditions that would trigger the transition from one screen to the next. This is actually one of the challenging parts in the design of the modeler, which we were able to solve in the version of the modeler discussed in section 3.2.5.

Modeling Inputs and Outputs. For the first two versions of the *Mobia Modeler*, an implementation of how users can specify inputs and outputs to the mobile application was not fully implemented. Instead, visual representations of inputs and outputs were just shown in the model, and the user can just drag and drop these to the screen.

Results and Discussion

From the prototypes created, a conclusion drawn from developing the prototypes using the Java Framework is that, there is no simple way of creating rich graphical user interfaces using this framework. One way to simplify this task is to use additional APIs such as the ones from the Piccolo Graphics Framework to abstract low level graphics code. Another way is to extend or build applications on top of existing development environments through the use of modules for Netbeans or plugins for Eclipse.

The succeeding versions of the *Mobia Modeler* however, are based on the Adobe Flash Platform[12]. The decision for choosing this platform is that it is relatively easier to develop Rich Internet Applications that feature animations, interactivity, rich UI, etc. as compared to using Java Framework APIs.

3.2.4 The *Mobia Modeler* Integrated-View and Multi-View: Evaluation of Integrated Modeless and Multiple-Mode Designs

Research Goal

The *Mobia Integrated-View* and the *Mobia Multi-View* were developed for the purpose of evaluating which of the designs promote ease-of-use and ease-of-learning [BFH09b]. A user study that aims to compare the *Mobia Integrated-View* which features an integrated modeless design and the *Mobia Multi-View* which presents a multi-modal design was carried out. The designs of the two prototypes originate from the designs of the first generation of prototypes *Mobia Piccolo* and *Mobia NBSuite* as shown in figure 3.27.

[12]http://www.adobe.com/flashplatform/

Implementation

The *Mobia Integrated-View* and the *Mobia Multi-View* were developed using Adobe Flash CS3[13] and ActionScript 3.0[14]. The prototypes were designed and developed together with Ugur Örgün [Örg09] as part of his Diploma Thesis.

Design Details

Before we go into details about the differences between the two prototypes, we will briefly discuss the different components and interactions that are common to both (see figure 3.29).

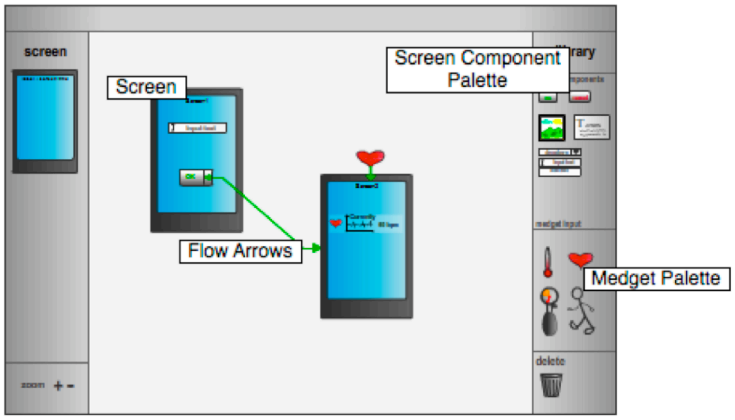


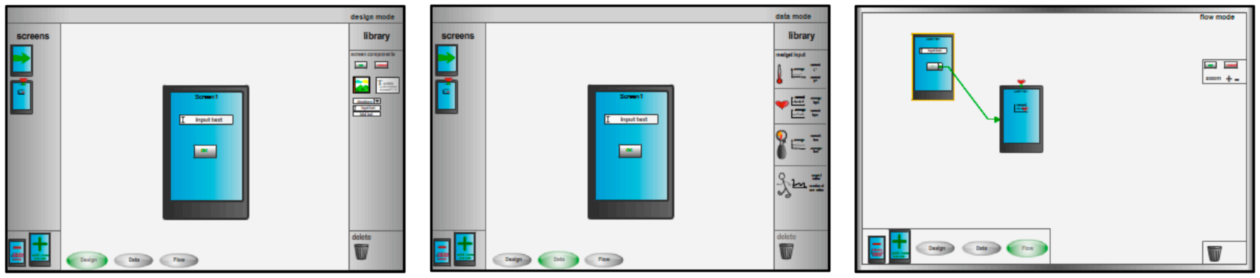
Figure 3.29: The *Mobia Integrated-View* prototype.

Modeling the User Interface, Inputs and Outputs The *screen* represents an instance of what is displayed on the mobile device at a certain point in time. It can contain user interface elements (e.g. buttons, textboxes) which are found in the *screen component palette*. The *screen* can also contain abstract representation of information collected from medical sensors that are found in the *Medget palette*.

Modeling the Application Flow. To represent the navigation from one screen to the next, *flow arrows* can be dragged from the buttons to the respective screen. A small arrow is integrated inside the button UI element to indicate that it is possible to add such a control flow.

The following shows the differences between the two prototypes.

[13]http://www.adobe.com/products/flash/
[14]http://www.adobe.com/livedocs/flash/9.0/ActionScriptLangRefV3/



(a) Design Mode (b) Data Mode (c) Flow Mode

Figure 3.30: The *Mobia Multi-View* prototype.

Modes. Figure 3.29 shows the *Mobia Integrated-View* that features an integrated and modeless design. Tasks such as designing the screen, adding data and adding control flow to the model are all done in one integrated view. The user can zoom in, to focus on designing a specific screen. The user can then also zoom out, in order to see an overview of the whole model.

The *Mobia Multi-View* features multiple modes. The interface presents a design which allows a stepwise design approach to modeling the mobile application. Figure 3.30 shows the three different modes: design, data and flow. Unlike the integrated design wherein all possible contents in the palette are shown, the palette contents in the multiple-mode version change depending on the task the user is doing.

- The *design mode* allows the user to design the contents of the screen individually.
- The *data mode* allows the user to add visualizations of possible data sources to the screen.
- And finally, with the *flow mode*, the user can add control flows between the screens.

The different modes are designed in a way that guides the user to the sequential phases of modeling an application particularly designing and adding data to the screens, and then later on think of the logic behind the application by adding flows between the screens.

Evaluation

The Objective and Hypothesis. The objective of the user study was to compare which of the two prototypes promotes ease-of-use and ease-of-learning. The hypothesis is that, participants will perform the tasks faster for the prototype version which is easier to learn and use. In order to further validate the results, the participants were asked to fill out a questionnaire in the end in order to get their subjective feedback (e.g. which is easier to use) with regards to the prototypes.

The Participants. There were 10 participants, 60% of which have professional backgrounds in Computer Science and 40% in the field of Archeology, Architecture and Social Welfare. The ratio of male to female participants is 2:3. The age range is from 22 to 31 years old.

In terms of programming background, 60% of the participants have programming background in general purpose languages and only one has a basic background in mobile programming.

The User Study Design. The user study design was a *repeated measures within subjects factorial design* [FH03]. This basically means that all the participants in the user study carried out the same tasks for each version of the tool. In order to reduce the *carry-over or learning effects* from using one version of the *Mobia Modeler* onto the next, each participant was randomly assigned on to which version to use first.

The *independent variable* is the version of the tool used, whether it is the *Mobia Integrated-View* or the *Mobia Multi-View*. The quantitative data recorded is the time it took for the users to accomplish the tasks (screen design task, control flow task).

Laboratory Setup. The user study was conducted in a closed room equipped with a laptop and an external monitor connected to it (see figure 3.31). A video camera was positioned in front of the external monitor and the participant in order to capture what the participants were doing as well as their reactions to the tasks respectively.

The online form that contains the task instructions and the questions that the participant had to answer were displayed on the laptop, while the prototypes were displayed on the external monitor directly positioned in front of the participant.

Each participant was asked to read and follow the instructions shown on the laptop screen. They were allowed to ask questions to the evaluator if the instructions were unclear. However, the evaluator was not allowed to teach the participants on how to use the prototypes. The goal was to find out how easily the participants can accomplish the tasks without any help, and with only the interface and the tooltips to guide them. However, in cases were it took the participants a very long time to figure out how things with the prototypes work, they were given bits of hints by the evaluator.

The Tasks. Each participant was asked to do three tasks for both the *Mobia Integrated-View* and the *Mobia Multi-View* prototypes. The first task was to *explore the tool* and give some comments and feedback based on their first impression of the tool. The second task was a *screen design task* in which the participants were asked to design several screens with different components and data. The third task was to *add control flow* between the different screens designed in the previous task. The time spent by the participants during screen design and flow design were logged. To wrap up, the participants were asked which version they found easier and more fun to use by filling out a questionnaire.

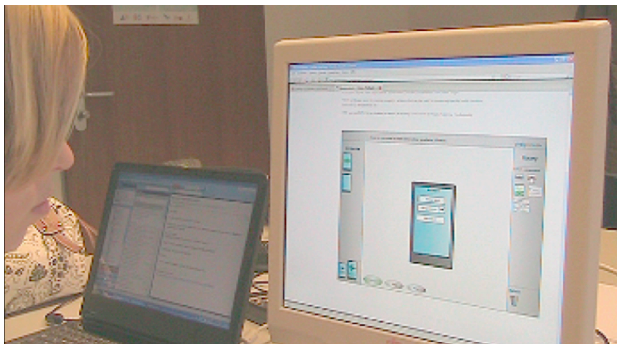


Figure 3.31: The setup for the user study.

Results and Discussion

The *Mobia Integrated-View*: The Choice of Non-Technical Users. Based on the results of the user study, it took less time for the users to accomplish the task using the *Mobia Integrated-View*. The average time needed for the screen design task using the *Mobia Integrated-View* is significantly faster (M = 3.61 minutes, SE = 0.19), than the *Mobia Multi-View* (M = 6.06 minutes, SE = 1.27, t(8) = -1.980, $p < 0.05$). For the flow design task, again the *Mobia Integrated-View* is significantly faster (M = 0.80 minutes, SE = 0.14), than the *Mobia Multi-View* version (M = 2.11 minutes, SE = 0.74, t(8) = -1.875, $p < 0.05$). This correlates with the user's subjective feedback in which 60% of the participants said it was easier to use the *Mobia Integrated-View* as compared to the *Mobia Multi-View*. In terms of enjoyment in using the modeler, 50% preferred the *Mobia Integrated-View* while 40% chose the *Mobia Multi-View*, and 10% chose neither.

Especially for a tool that targets non-technical users, it is important that it provides ways to help the user accomplish the task in the easiest possible way, but also offer the user a more fun experience so that they will be motivated to use it.

Observing the User Experience. While observing how the participants interacted with the different prototypes, some specific user traits were collected. These traits can serve as a guide when developing EUD tools such as the *Mobia Modeler*.

Instead of enumerating the individual traits, we have grouped them together and associate it with a specific type of user.

- **The Clueless.** This type of users basically have no idea on how to start with their model. They do not know which goes first: the screen or the component, or what action to do in order to add flows for instance. An example mistake that the clueless user did during our user study was to drag components (e.g. a button) to the empty area in the center without creating a screen beforehand. 40% of the participants actually made such a mistake, until they realized that the screen should go first and then the components.

These types of users are also prone to freeze, just stare at the screen and do nothing when they do not know what to do anymore. They are basically afraid to explore. They need a bit coaching in order to move on.

A possible solution to assist the *clueless user* is provide some wizard in the beginning to guide them in the process of creating their application. Help functions such as tooltips that pop up after a specific period has passed without any sign or user activity can also be useful to assist users when they are stumped. Such tooltips can also be used to signal the user if they are doing some incorrect actions (e.g. dragging a component to an empty area).

- **The Clicker (Trigger) Happy**

  These type of users mainly click on everything from the clickable to non-clickable items and expect something to happen. They are similar to the clueless user, but more proactive in a sense that they do not stop doing anything, but just not the correct action in order to accomplish the task.

  A possible solution to assist the *clicker happy user* is to provide tooltips that gives some hints to the user in order to stir them into the right direction.

- **The Free Spirits**

  Also known as the non-conformist, these types of users basically do not follow instructions. They do not read the hints and tips given to them even if it is right in front of their eyes. They also do not try to solve the problem in a step-by-step and logical manner, and just try to do anything as they please.

  Providing hints and tips in the form of dialogs and tooltips to the *free spirits* would be futile because of their habit to ignore such functions. One possible solution is just to allow them to explore and do as they please but provide ways for them to recover in cases where they make critical mistakes (e.g. deleting the whole model), or find ways to help them avoid making mistakes (e.g. nothing happens when a wrong action is performed).

- **The Overthinker**

  As oppose to the free spirits, these types of users follow instructions too much and read the hints and tips too often, that it slows them down in performing the tasks. They are basically successful in accomplishing the tasks despite their slow speed, and are liable to be successful in subsequent tasks.

  A possible solution to assist the *overthinker* is to provide time-triggered help functions such as tooltips that would give hints for the next possible steps that need to be taken when no activity has been detected for a certain amount of time. For example, if the a certain screen is already populated with user interface elements, a pop-up would appear asking if the user wants to add control logic and hint which UI elements they can modify to add control logic to.

- **The Neat Freaks**

  These types of users want everything in their model to be organized. Half of the time spent doing the assigned task is actually used to arrange the different components in their model. They also like to concentrate on one thing at a time and are confused when too much information is presented to them.

  A possible solution to assist the *neat freaks* is to provide automatic layout of the model so that they do not have to spend time organizing it themselves. Providing the tool with different modes each featuring a minimalist design is also a good thing in order for them to concentrate on one task at a time. Feature reduction for the tool is also a possible solution. An example of feature reduction will be reducing the number of components in the palette if those components will not be used in the model.

- **The Idea Generator**

  This type of users are usually a good source of ideas for new types of interaction techniques that designers and programmers have not thought about. They may do actions that sound like a clueless action at first, but after a good consideration, can in fact be sensible. For instance, out of the 10 participants, 40% actually returned a screen component to the palette thinking they should put it back where they got it, instead of putting it directly to the recycle bin. This may at first sound like a nonsensical action, but it just makes sense that people should put things back when they do not need it.

  Listing down the peculiar things that the *idea generators* were doing when interacting with the system, and later on analyzing if it would make sense to implement such features for future versions is a good way to improve usability of the tool.

3.2.5 Redesigning the *Mobia Modeler*

Research Goal

The findings from the previous *Mobia Modeler* prototypes were taken into account in redesigning the *Mobia Modeler* [BFTH10]. The following design decisions were made based on the findings:

- The *Mobia Modeler* will feature an integrated modeless design since the *Mobia Integrated-View* proved to be easier to use based on the user study made.
- Functions that assist the user such as wizards and tooltips are made available. Also, invalid user actions are ignored.
- To reduce time users spend on actions unrelated to model development such as arranging model components (e.g. screens), manual layout is not permitted.

- To reduce time users spend on finding components for their model, feature reduction based on the information collected during the initial wizard is made.

The primary goal was to develop a complete version of the *Mobia Modeler* which will try to address the issues that were not taken into account in the past versions. The following issues are:

- **Modeling the User Interface.** Usually, non-technical users are unaware which user interface components to use for a certain application. Sometimes, users know what features they want to be present in their applications but have no idea on how to combine the different UI elements available to complete this feature. It would not make sense to give the non-technical user a rich set of user interface components such as buttons, text boxes, etc., and complex layouting capabilities when they do not know which one to use and how to use it anyway.

  Another challenge is how to achieve platform independence [KWB03] in modeling such interfaces. There is a vast difference between different mobile platforms in terms of capabilities, input methods and how user interfaces are represented. A certain feature that is present in one platform may not exist, or is represented differently in another platform. With this in mind, the challenge is how to model user interfaces in a way that it can achieve platform independence, but at the same time, keep the contextual meaning of components in the application being modeled intact.

- **Modeling the Application Flow.** Another challenging problem is finding an easy way for non-technical people to model the logic of an application. This may range from simple application logic such as *"transitioning from one screen to the next if a button is pressed"*, to more complicated logic such as, *"phone dials an emergency number if a patient's heart rate as indicated by the heart rate monitor goes more than 150 bpm"*. In the previous versions of the *Mobia Modeler*, we have seen that application flow was represented by dragging arrows between screens. However, representing the conditions that trigger these actions was still not met in the previous prototypes.

- **Modeling Inputs and Outputs.** In section 3.1.4, the definition of inputs and outputs in the context of mobile applications were presented. A challenge for us would be finding a way to easily represent such input and output information to the non-technical user such that they would be able to easily integrate and make use of it in their applications.

Implementation

The *Mobia Modeler* was developed using the *Adobe Flex 3.3*[15] which is based on *ActionScript 3.0*[16] and uses *MXML*[17] to define user interfaces. In order to simplify development, the *Mate Flex Framework*[18] which is a tag-based event-driven framework was used as a layer on top of the Flex Framework. This helps in the structuring of complex applications.

The design and development of the *Mobia Modeler* was developed with Max Tafelmayer [Taf09] as part of his diploma thesis.

Design Details

The *Mobia Modeler*'s New Design Approach: Configurable Component-Based Design. The main design idea for the *Mobia Modeler* is *"configuration over combination"* [Taf09]. This means that instead of building mobile applications by *combining individual user interface elements* in the model, *components are combined and configured* instead.

The approach of *configurable component-based* design has been applied to many areas in software and embedded systems. However, according to Fernando et al. [FSH+01], there are still issues that need to be addressed in the design of such systems. First of all, there is no clear definition in literature about what a *component* really is. The common characteristics often mentioned in literature is that a component is a self-contained unit that can be independently deployed and should have clear specifications of its requirements and what it can provide [FSH+01]. The key issues emphasized by Fernando et al. [FSH+01] that are of importance to this research are related to software components. These issues pertain to the attribute-dependent categorization of components, the development and storage of component configurations, and how to provide guidance to the developer/user to choose the right components.

The formal definition of configurable components in the context of Mobia is: *"logical container for multiple user interface elements that has a clearly defined meaning and acts as a whole, and which functionalities can be modified through simple configuration"* [BFTH10] [Taf09].

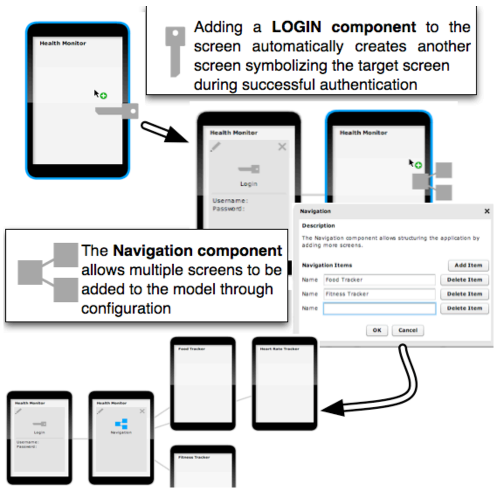
The issue of categorization mentioned by Fernando et al. [FSH+01] is addressed by grouping configurable components in the *Mobia Modeler* into: Basic, Structure, Sensor and Special.

Before we discuss how configurable components can address the issues in designing for the non-technical user, the different types of configurable components in the *Mobia Modeler* and its functionalities will first be presented.

[15] http://www.adobe.com/products/flex
[16] http://www.adobe.com/devnet/actionscript/articles/actionscript3_overview.html
[17] http://opensource.adobe.com/wiki/display/flexsdk/MXML+2009
[18] http://mate.asfusion.com/

- **Basic components** represent functionalities that are common to mobile devices (e.g. notepad, camera, recorder, etc.).

- **Structure components** represent components that have the ability to add new screen instances to an application. An example is a login screen which can branch to another screen after a successful authentication. Another example is a navigation screen that can serve as a starting point for the application to branch to several other screens and can act just like a main menu screen. Examples are shown in figure 3.32.



Adding a **LOGIN component** to the screen automatically creates another screen symbolizing the target screen during successful authentication

The **Navigation component** allows multiple screens to be added to the model through configuration

Navigation

Description

The Navigation component allows structuring the application by adding more screens.

Navigation Items

Name	Food Tracker	Delete Item
Name	Fitness Tracker	Delete Item
Name		Delete Item

Figure 3.32: The *Mobia Framework*'s structure components in action

- **Sensor components** represent real-world sensors that are used in addition to the mobile device. These components serve as input devices that can collect various information. An example in the domain of health monitoring is an ECG sensor that collects ECG information from an individual. Sensor components aside from representing the real device allow displaying data of real sensors by using custom visualizations. Take note that a sensor component in the modeler may represent a group of real sensors. Configuration of outputs on the mobile device can also be done via the sensor components. Unlike the first two types of components, sensor components represent domain specific information.

- **Special components** are components that represent mini applications in a certain domain. For our example domain which is mobile health monitoring, we have examples like fitness diary, nutrition diary and call for help functions for the application. Some special components such as the personal data component may be available to other domains as well.

The following describes how configurable component-based design and the different types of components can solve the design issues previously discussed:

- **Modeling the User Interface through Configuration.** The common approach used in creating user interfaces is by assembling individual elements on a palette (e.g. a screen). However, as already mentioned, most non-technical users have no idea what elements they need for a specific application (e.g. choosing between using a text field or drop down box for age input). Therefore, we proposed using configurable components to alleviate this problem.

  Specifically for mobile devices wherein the screen size is fairly limited, there is only a number of elements that you can put on the screen. Instead of letting the user create user interfaces composed of individual elements, a proposed way would be *to provide the user with configurable components that have already some predefined meaning and which can easily be configured to meet the user's needs*. In this way, the user can *concentrate on the solution to the problem domain and not be bothered about technical details* such as deciding which user interface components are needed, layout problems when multiple components are placed in one screen, validation, etc.

  This solution works since the applications created are *domain-specific, and it is possible to define specific configurable components that targets to solving problems in that domain*. This solution also addresses *platform independence* which is one of the goals of model-driven development. Since *different mobile devices have different features and may represent different user interface elements* (e.g. a textfield in one platform may be represented as another user interface component in another platform), it would make sense *to provide the user with a component that represents a solution to the problem instead. The responsibility of transforming these representations to a target platform lies in the underlying model processor and code generator.*

- **Modeling the Application Flow through Configuration.** In previous versions of the *Mobia Modeler*, simple application logic such as transition from one screen to the next is represented as lines connecting multiple screens and with buttons representing triggers to screen transitions. These transitions are manually created by the user by dragging a button instance on the screen, and dragging the arrow from the button to a target screen. However, a problem with this approach is when we try to model transitions that involves validation (e.g. validating login). Validation of the models created by the non-technical users will also prove to be problematic when it comes to processing these models for later code generation.

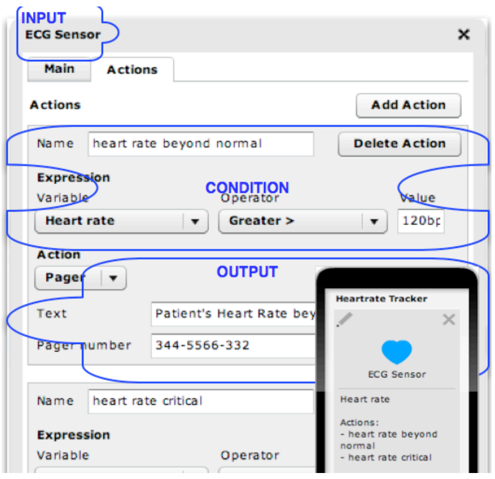
  The *Mobia Modeler structure component* is the proposed solution to this problem. *Structure components* already have a predefined meaning that allows an application to branch from one screen to the next depending on certain conditions that are defined through configuration. The lines shown between screens still signifies transitions, however, its difference with the previous approach is that, these transitions are automatically created by the modeler and not the user. Validation will be easier since the user is limited to using predefined conditions specific to the type of structure component there is.

- **Modeling Inputs and Outputs through Configuration.** One challenging problem that was not covered in the previous versions of the *Mobia Modeler* is the modeling of inputs, outputs and conditions that bind them together. There were three questions that we had to look into before designing the solution to this problem:

  - What are the common sources of inputs?
  - What are the common outputs?
  - What are the conditions based upon and how can these conditions be easily modeled?

  To answer the first question, the most common source of input information are the external devices that supply information to the application. The conditions (as an answer to the last question), are basically based on these input information. The output information is based upon the capability of the target device, which in this case is the mobile device. Binding these pieces of information together, we need to find a way to model complex logic that involves information taken from external devices (inputs), and the actions (output) that will be executed that satisfies certain conditions.

  As a solution, instead of treating these three types of information as independent constructs of the model, we encapsulate the three into one configurable component which are called *sensor components*. The *sensor component* represent as the input to the application itself, and conditions and outputs are bounded to it (figure 3.33).



Name: heart rate beyond normal
Expression
Variable: Heart rate
Operator: Greater >
Value: 120bp
Action: Pager
Text: Patient's Heart Rate bey
Pager number: 344-5566-332

Name: heart rate critical
Expression
Variable
Operator

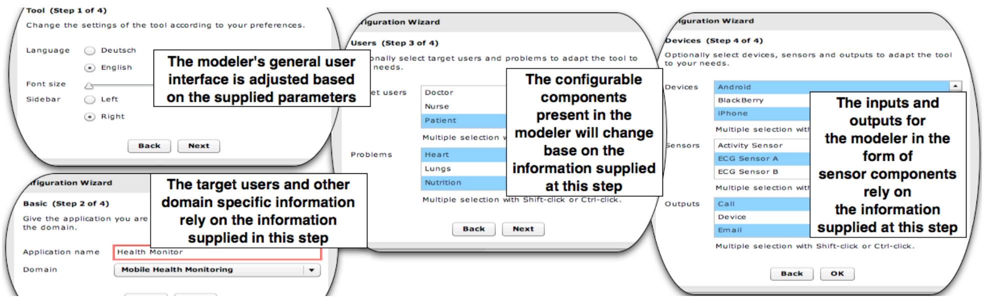
Heart rate
Actions:
- heart rate beyond normal
- heart rate critical

Figure 3.33: An example ECG component that is configured based on the example scenario

A Guide to the *Mobia Modeler*. The basic features of the *Mobia Modeler* such as the wizard and the user interface will be presented next.

The Wizard. The modeler starts with a wizard which helps the user configure the modeler's general user interface and supported functionalities. This process consists of four simple steps as shown in figure 3.34.

- **UI Configuration.** The first step consists of configuring the modeler's basic UI such as language, font size and sidebar orientation.
- **Application Information.** The second step asks for the application name and the domain of the application being modeled. As of the moment, the *Mobia Modeler* only supports the mobile health monitoring domain.
- **Domain-Specific Information.** The details in the third step of the wizard rely on the domain that was chosen in the second step, which in this case is the mobile health monitoring domain. In this domain, there are two essential pieces of information that are needed: the target users of the application and the types of health problems. The configurable components that are made available to the user later on will rely on the information supplied in this step.
- **Output Configuration.** The final step of the wizard involves getting information about the target devices that will be used in conjunction with the application being modeled. It should be noted that changes to the data supplied in the wizard is possible by choosing the *Configuration* option in the application's menu bar.


Tool (Step 1 of 4)
Change the settings of the tool according to your preferences.
Language: Deutsch [✓ English]
Font size
Sidebar: Left [✓ Right]

iguration Wizard
Basic (Step 2 of 4)
Give the application you are the domain.
Application name: Health Monitor
Domain: Mobile Health Monitoring

iguration Wizard
Users (Step 3 of 4)
onally select target users and problems to adapt the tool to needs.
et users: Doctor, Nurse, [✓ Patient]
Multiple selection w
Problems: [✓ Heart], Lungs, [✓ Nutrition]
Multiple selection with Shift-click or Ctrl-click.

iguration Wizard
Devices (Step 4 of 4)
Optionally select devices, sensors and outputs to adapt the tool to your needs.
Devices: [✓ Android], BlackBerry, [✓ iPhone]
Multiple selection wit
Sensors: Activity Sensor, [✓ ECG Sensor A], ECG Sensor B
Multiple selection wit
Outputs: [✓ Call], Device, [✓ Email]
Multiple selection with Shift-click or Ctrl-click.

Back
Next
OK
The modeler's general user interface is adjusted based on the supplied parameters
The target users and other domain specific information rely on the information supplied in this step
The configurable components present in the modeler will change base on the information supplied at this step
The inputs and outputs for the modeler in the form of sensor components rely on the information supplied at this step

Figure 3.34: The wizard configures the interface for the modeler based on the user's preferences and the target domain of the application to be modeled

After the whole process is done, the *Mobia Modeler*'s interface is adapted based on the supplied information in the wizard. Figure 3.35 shows the user interface of the modeler.

The User Interface. The modeler's interface is composed of three main parts: the Main Area, the Menu Bar and the Side Bar.

- **The *Main Area*** is the only view used by the *Mobia Modeler* for modeling the mobile application.

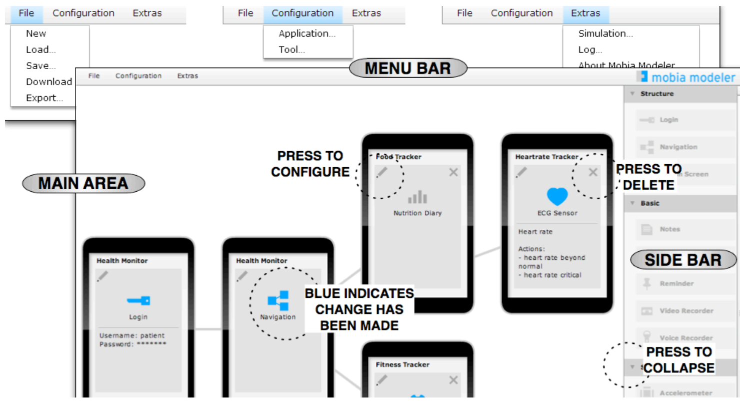


Figure 3.35: The *Mobia Modeler* with the Health Monitor sample application model

- **The *Side Bar*** contains elements that represent the different configurable components that can be added to the screens in the Main Area. An element is made up of an icon representing its functionality and the name of the configurable component. Each element in the sidebar belongs to one of the following four groups: Structure, Basic, Sensors, and Special. Each of the groups can be collapsed by clicking on its header, as symbolized by the small arrow to the left of each group name. Depending on the current state of the main area, not all configurable components are available at a given time. In that case some or all of the elements in the sidebar are disabled. This design was done to guide users in the modeling process and prevent mistakes. The sidebar can be adapted indirectly through settings made in the configuration wizard during the creation of a new application. Configurable components that are not needed for an application are simply removed from the sidebar.
- **The *Menu Bar*** is located at the top of the *Mobia Modeler*'s user interface and contains additional functions. These functions are divided into the following three menu items: File, Configuration and Extras. All the menu items shown in figure 3.35 are already supported in the current version of the *Mobia Modeler* except for the *Simulation* function. The created model can be saved and loaded either through the server or as a local copy.

Components can then be added to the main area depending on the application the user wants to create. Each component can be configured by clicking on the pen symbol (see figure 3.35) on top of each component. A component can also be deleted from the model by clicking on the cross symbol. Aside from tool tips, the *Mobia Modeler* also offers visual hints to the user such as the change in color of a configurable component if the default values have been changed, or by disabling components in the Side Bar to prevent users from doing invalid moves.

Going back to the issues raised by Fernando et al. [FSH$^+$01] as previously mentioned, developer guidance is achieved by the *Mobia Modeler* in the form of wizards, help functions (e.g. tool tips), visual hints (e.g. change in color to signify changes) and error prevention techniques (e.g. removing and disabling unnecessary components in the user interface).

Finally, the graphical model can then be serialized into some XML format that is an important data used for processing the model into code. Details about the transformations (Chapter 4) and models (Chapter 5) will be discussed in the succeeding chapters.

Evaluation

The Objectives. In order to evaluate the *Mobia Modeler*, a qualitative user study was conducted which generally aims to identify issues that arise from the current design, and find ways to improve it. The goals were:

- To evaluate the different features of the *Mobia Modeler*(user interface elements, workflow design, interaction).
- To find out if people (especially the non-technical people) understood the concepts of the *Mobia Modeler* and its design.
- To collect subjective feedback (opinions, comments, suggestions) from the participants in order to improve the *Mobia Modeler*.

The Participants. There were 16 participants in the user study (7 males, 9 females) with the average age of 30. All of the participants had experience in computer usage. 10 of them have no experience in programming, while the other 6 use programming in their respective professions. Although none of the participants have experience in using mobile health monitoring applications, they claimed that they understood the concept.

The Process. The user study started with the participants filling out a questionnaire to collect personal information, technical experience and overall background knowledge that were relevant to the study.

The next phase was the *Exploration phase* in which the participants were asked to use and explore the different features of the *Mobia Modeler* for five minutes. The features that the participants used during the exploration phase were recorded.

After the exploration phase, a *short interview* was conducted to find out their understanding of the *Mobia Modeler* and its features.

The participants were then asked to do two more modeling tasks *(Task phase)* specifically, the *Activity and ECG Monitor* application and the *Epilepsy Safety System* application. For these tasks the participants had a copy of the step-by-step instructions in order to create such applications. Because of the detailed instructions, all participants were able to finish the tasks and differed only in the duration in which they were able to accomplish each task. Finally, the users were asked to fill out a questionnaire to get the participants' feedback. Each user study session took approximately 45 minutes to finish.

Results and Discussion

In order to *evaluate the different features of the* Mobia Modeler, we observed how the participants interacted with the modeler during the exploration phase and combined it with the participants' answers during the interview.

The feedback in terms of how an application's control flow is modeled using *structure components* is that, it was not very intuitive and easy to use (e.g. some participants got irritated with using the *Navigation component* to add screens).

In terms of the subtle visual hints (e.g. change in color of the component when configured), only 63% of the participants understood the meaning of the hints.

The modeler's features such as disabling components in the sidebar or disallowing deletion of some screens when subsequent screens are connected to it, were added in order to prevent users to make major mistakes in the modeling process. However, during the user study, it was apparent that these error-prevention features were not clear to most participants.

In order to *find out if people (especially the non-technical people) understood the concept of using the Mobia Modeler and its design*, we present in figure 3.36 feedback from the participants.We also want to compare the views of the two types of participants (programmers and non-programmers or the non-technical people) just to see how the two groups perceive the overall concept and design of the *Mobia Modeler*. As we can see in the graphs, in terms of understanding the general concepts, usability and design approach of Mobia, it scored higher in the programmers' group as compared to the non-programmers. The variance between the answers of the people in the non-programmers group is also higher which correlates to the different experiences that the participants in this group have.

Most of the *feedback from the participants* with regards to their views in improving the *Mobia Modeler* pertain to providing more help functions in the *Mobia Modeler* such as video tutorials and demos. Most of them also wanted to have the simulate function in order to see what they have created. Other points for improvement were:

- **Provide Richer User Experience and Help.** Users in general are more encouraged to work on a certain task if they see immediate feedback. One thing that is currently missing in the modeler is the simulator which tries to simulate the current model the user is working on. Another thing that can be added is to provide templates or pre-made models that the user can explore in order to see the capabilities that can be done with the tool.

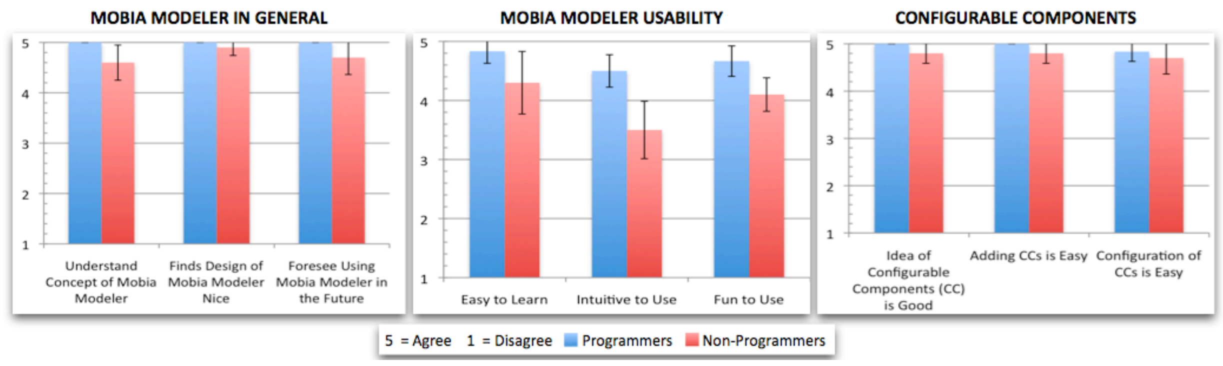


Figure 3.36: Comparing feedback from non-programmers (non-technical people) and programmers with regards to the *Mobia Modeler*and its concepts

- **Configuration Validation.** Configuration done by the user to the component in the model should be validated. An example of validation of the conditions in a sensor component, such as sending an SMS if the heart rate drops below 200 would not make any sense because the SMS would always be sent. The *Mobia Modeler* should detect such situations and warn the user about potential problems and make suggestions for alternative solutions.

- **Support for Plug-and-Adapt.** The current approach to getting information about the devices the user wants to use with the application is by choosing it in the modeler's wizard. To ease the user of this task, there should be a way for the *Mobia Modeler* to automatically detect hardware components that the user wants to use together with the application being modeled (similar to [vHVF09]). This can be done by having the device plugged to the computer where the *Mobia Modeler* is running, or by detecting the device through wireless methods.

Aside from the user study conducted, surveys and interviews were also conducted with experts in the medical field in order to get feedback about the *Mobia Modeler*. The details about the results were presented in sections 2.2.4 and 2.3.2 in the previous chapter.

3.2.6 The *Mobia Proto-Go*: An Alternative Tool for Platform-Specific Development

Research Goal

The goal of the *Mobia Modeler* was to create a tool for non-technical users that would allow them to easily create platform-independent models of a mobile application. However, a limitation of the approach employed by the *Mobia Modeler* is that it only gives a high level overview of the components. There is no way for the user to see how the application actually looks like or how it works unless code has been generated and compiled from the model which then has to be deployed by the *Mobia Processor* either on an emulator or the target device. This was because the *Mobia Modeler* was designed for creating a single processable model that can be used to generate applications for different platforms. This trade-off in the design was necessary in order to achieve this goal.

The goal for creating the *Mobia Proto-Go* is to propose a supplementary platform-specific tool that will support features that are currently lacking from the *Mobia Modeler*. This tool differs from the *Mobia Modeler* in a way that it features platform-specific constructs and runs on the target mobile device. The tool is part of and is fully interoperable with the *Mobia Framework* family of tools as will be presented later.

Implementation

The *Mobia Proto-Go* was developed using the Android Framework[19].

Design and Functionality

The *Mobia Proto-Go* features a configurable component-based design which stems from its sister tool, the *Mobia Modeler*. This means that, applications are built upon existing components which are then configured and adapted based on the needs of that application.

The tool incorporates the advantage of configurable component-based design such as guiding the users in development by giving them some base templates, which they can then combine and modify based on specific requirements. It also has the additional advantage of WYSIWYG tools, which shows the user immediately what the end result would be [Tid06].

Custom User Interface. One of the problems of the *Mobia Modeler* as raised by one of the participants in the survey (section 2.2.4) is that the representation of components in the *Mobia Modeler* is too high level. They want to be able to fully customize the user interfaces of their applications by having a tool that provides basic controls (e.g. textbox, buttons). This feature is added to the *Mobia Proto-Go* by featuring platform-specific UI elements in the tool.

Modes and Operations. The *Mobia Proto-Go* is composed of two modes with several allowable operations on each mode (figure 3.37). The different modes have to be introduced because of the many functionalities that have to be overloaded [Tid06] on a screen space that is highly limited relative to the screen space of a personal computer.

- In the **View and Simulate mode**, the user is allowed to view either ① individual screens when the device is in a vertical position; or ② view all screens, including a representation of the components inside the screens in the

[19]http://developer.android.com/sdk/index.html

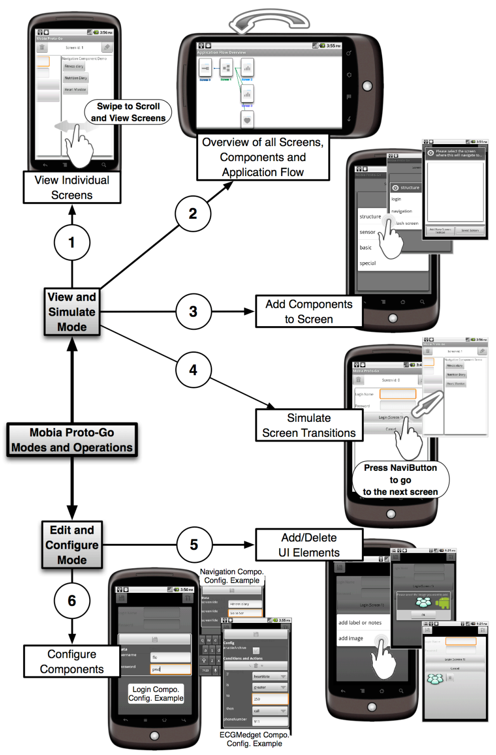

Swipe to Scroll and View Screens
View Individual Screens
Overview of all Screens, Components and Application Flow
1
2
3
4
5
6
View and Simulate Mode
Add Components to Screen
Simulate Screen Transitions
Press NaviButton to go to the next screen
Mobia Proto-Go Modes and Operations
Edit and Configure Mode
Add/Delete UI Elements
Navigation Compo. Config. Example
Configure Components
Login Compo. Config. Example
ECGMedget Compo. Config. Example

Figure 3.37: The *Mobia Proto-Go* modes and operations.

application and the transitions between them when the device is in a horizontal position. In this mode, the user ③ can also add configurable components in each individual screen.

Just like in the *Mobia Modeler*, *structure components* (i.e. login, splash, navigation) are used to add new screens to the application, where the components also incorporate pieces of application logic. Posing this restriction in adding screens exclusively through *structure component* components ensures that there are no unreachable screens in the application (i.e. all the screen instances are connected in some way).

In order to ④ simulate the transition between the screens in the application, *NaviButtons* are made available. These buttons act like hyperlinks which trigger transition from the current screen to a target screen. By default, *structure components* are equipped with *NaviButtons* to simulate such transition.

- In the **Edit and Configure mode**, the user is allowed to ⑤ customize the user interface by adding new UI elements or deleting non-default UI elements (i.e. those which are not part of the original UI elements provided by the component). Depending on the type of the component available on each screen, only selected types of UI elements can be added to the screen. Because of the ability to modify the UI of a screen, the model that will be exported by the *Mobia Proto-Go* has been extended to accommodate changes in the layout. Only screens whose layout have been modified will contain this information. Figure 3.38 shows the parts of the model in the *Mobia Framework* and the added information called *layout* that contains a serialized form[20] of the new UI layout. The *Mobia Processor* responsible for transforming the model to code will use this information for rendering the new layout information instead of using the default layout templates.

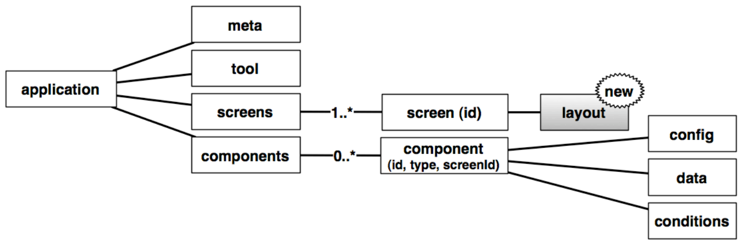


Figure 3.38: The *Mobia Proto-Go* application model.

Aside from modifying the UI, ⑥ configuration of each component and addition of more complex application logic (i.e. carrying out certain actions or outputs based on inputs and conditions) are performed in the *Edit and Configure mode.*

Interoperability with *Mobia Framework* tools. As already mentioned earlier, the *Mobia Proto-Go* is interoperable and works with the *Mobia Framework* family of tools (figure 3.39). The model generated from the *Mobia Proto-Go* can be imported to the *Mobia Modeler* and viewed there. However, since the *Mobia Modeler* does not support UI rendering, the modified UI in the *Mobia Proto-Go* will not be shown. The *Mobia Processor* can also be used to generate the final working code by processing the model generated by the *Mobia Proto-Go*.

Screen Space Management by Utilizing Advanced Mobile Device Capabilities. A challenge in designing function-rich mobile applications such as the *Mobia Proto-Go* includes the decision on where to put the application controls to invoke such functionality with as few clicks/presses possible and without taking up a lot of screen space on the main screen. The proposed solution is to utilize the advanced capabilities with which mobile devices are now equipped such as the following:

- **Adding Components and UI Elements via Context Menus.** Instead of providing a separate button to add a component during *the view and simulate mode* or to add a new UI element on the screen during *the edit and configure mode*, context menus which are accessed via a long press on the touch screen device are used to bring up a list of components/UI elements to be added to the screen.

[20] The serialized layout is expressed in Android's XML vocabulary.

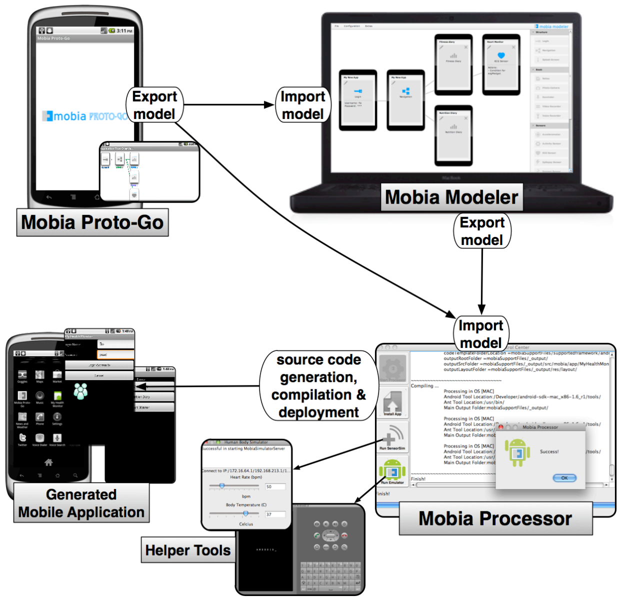


Figure 3.39: Interoperability between the different *Mobia Framework* family of tools.

- **Changing Views via Physical Device Manipulation.** In the *the view and simulate mode*, both detailed view of individual screens and an overview of all the screens and corresponding components in the application can be done by simply rotating the device in a vertical position or in a horizontal position. No additional controls or menu that should be present on screen are necessary in order to invoke the different views.
- **Fast Viewing via Gestures.** To view the individual screens in the application, simply swiping across the mobile device's screen in order to show the available application screens and corresponding components.

A Comparative Evaluation

The *Mobia Proto-Go* was primary developed in order to prove the *Mobia Framework*'s extensibility which will be discussed further in section 4.3.3. Since user studies have yet to be conducted in order to test the usability of the *Mobia Proto-Go* tool, a comparative evaluation between the *Mobia Proto-Go*(+ *Mobia Processor*) with systems from both commercial and academic initiatives that allow the development of *fully functional mobile applications* ranging from runnable prototypes to marketable software products (figure 3.40) will be presented instead.

Integrated Development Environments (IDEs) such as Netbeans with Mobility Pack [Net], Android Development Tools (Eclipse Plugin) [Anda] and XCode+Interface Builder [Appc] have the advantage of creating fully functional mobile applications and feature a rich set of tools (e.g. debuggers, emulators, GUI builders, etc.) that can assist users during de-

		IDEs			Visual/ Graphical Prog.		Authoring Tools			Model-Based Systems		
		Android Devt. Tools (Eclipse Plugin)	Netbeans Mobility Pack	Xcode + Interface Builder (for iOS	GameSalad Creator	AppInventor	MakeIt Toolkit	Topiary	Mobile Bristol	ModelBaker	Mobia Modeler + Mobia Processor	Mobia Proto-Go + Mobia Processor
Target Users	Programmers	●	●	●			●				●	●
	Non-Programmers				●	●	●	●	●	●	●	●
Running Platform	Computer-Based	●	●	●	●	●	●	●	●	●	●	●
	Mobile-Device-Based											●
Produced Software Artifacts	Target Platform Source Code (Importable to IDEs)	●	●	●			●				●	●
	Target Platform Mobile Application	●	●	●	●	●					●	●
	Web-Based Application (Runnable on any web browser)									●		
	Tool-Specific Application (Requires a special interpreter running on the mobile device)							●	●			

Figure 3.40: Comparison of systems/tools for mobile application development.

velopment. However, programming skills are required in order to build applications using these systems.

Systems that feature drag-and-drop or point-and-click methods for building applications such as graphical/visual programming environments (e.g. Game Salad [Gam], App Inventor [Appa]), authoring tools (e.g. Mobile Bristol [HCM04], Topiary [LHL04], MakeIt Toolkit [HS08]) or model-based systems (e.g. ModelBaker [Mod], *Mobia Modeler* [BFTH10]) target less technically inclined users (i.e. low or no programming skills). Some disadvantages of these systems, however, include the unavailability of raw source code (i.e. only the final mobile application is made available) [Gam][Appa], which limits it extensibility (i.e. add more functionality, improved interface) by people with programming skills. [HS08] on the other hand, provides a way to generate the source code and let it be extended using IDEs (e.g. Netbeans). However, there is no way to abstract this step if the user is a non-programmer. [HCM04][LHL04] require special interpreters on the mobile device in order to run the generated prototypes, and [Mod] has the disadvantage of just producing web-based applications which cannot fully access more advanced mobile device features, and [BFTH10] do not provide a way for the user to see how the interface would look like, unless the model is already processed, and the code is generated, compiled and deployed on the target device or emulator.

Together with the *Mobia Processor*, the *Mobia Proto-Go* can directly generate the complete code and the fully functional mobile application, whose deployment is abstracted from the user through the use of automated scripts. Programmers can also use these tools (*Mobia Proto-Go*+*Mobia Processor*) to help them rapidly create the base classes, control flow and user interface (provided the components they need are already supported) for their application. They can then just import the generated source code to the supported IDEs and focus on the application enhancements (e.g. richer user interface, added functionality, etc.). The application generated also makes use of the target framework APIs and do not require special interpreters in order to run. Lastly, the *Mobia Proto-Go* differs from the mentioned systems in that it is the only tool that allows prototyping anytime and anywhere directly on the target mobile device.

3.3 Usability of the *Mobia Modeler* Prototypes

Since the *Mobia Modeler* is designed to be used by non-technical users, usability is one priority in the development of the prototypes. As mentioned in section 3.1.6, Paterno [Pat99] defined a system to be usable not only if it is easy to learn and use, but also if it has the following:

- The system's relevance in serving the users' needs.
- The efficiency of how users carry out their tasks when using the system.
- The users' feelings or attitudes towards the system.
- The ease of learning the system especially during initial use.
- The system's tolerance to unexpected or wrong usage.

In the following sections, each point will be discussed on how these attributes are addressed in the development of the *Mobia Modeler* prototypes and wraps up all the lessons learned throughout the duration of this research.

3.3.1 System Relevance

"The system's relevance in serving the users' needs."

The goal of creating the *Mobia Modeler* is to simplify the creation of mobile applications for non-technical users. Therefore, the *Mobia Modeler*'s relevance can only be measured when we get feedback from potential target users of the system. For the initial target domain of *mHealth* for example, collecting opinions from medical professionals and those people doing medical-related research is important in order to see if tools like the *Mobia Modeler* can be useful for people in this domain.

A mix of positive and some reluctant feedback were collected and presented in sections 2.2.4, 2.3.1 and 2.3.2. The positive feedback expressed were: how the *Mobia Modeler* can be used as a communication tool between the developers and also the patients, an empowering tool which allows the medical professionals to have a sense of ownership with the applications they have made, and also a tool that can be used to educate health professionals. The hesitation with regards to the use of the *Mobia Modeler* by medical professionals was with regards to how some are not as technology savvy as the others, and also the willingness of medical professionals to allocate time from their busy schedule in creating such *mHealth* applications.

3.3.2 Task Efficiency

"The efficiency of how users carry out their tasks when using the system."

In order to evaluate if the different prototypes allowed the users to carry out the different tasks efficiently, user studies were conducted. During the user studies, task times were measured, the participant's interaction with the tools we observed, and the participants' subjective feedback were collected. The results for each user study influenced the design of the succeeding versions. The details of the user studies and lessons learned were discussed in sections 3.2.4 and 3.2.5.

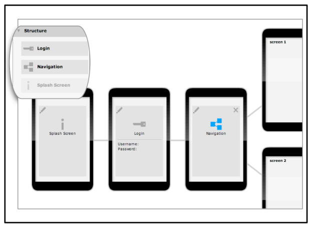
A summary of the lessons learned that allowed users to efficiently carry out tasks are the following:

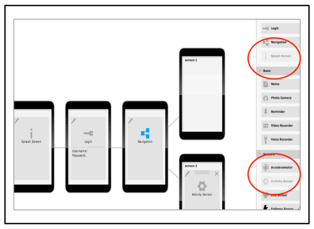
Task Simplification via Configurable Components. The general design of the *Mobia Modeler* which features configurable components can simplify the tasks of the users by allowing them to concentrate on the solution to the problem domain and not be bothered about technical details such as deciding which user interface components are needed, layout problems when multiple components are placed in one screen, validation, etc.

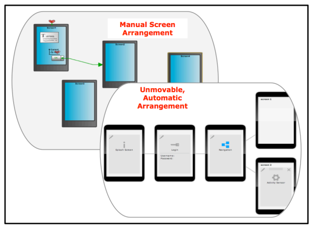
Specifically for mobile devices wherein the screen size is fairly limited, there is only a limited number of elements that you can put on the screen. Instead of letting the user create user interfaces composed of individual elements, providing the user with configurable components that have already some predefined meaning and which they can easily be configured to meet their needs.

Task Simplification via Constraints. According to Norman [Nor02](from Norman's Seven Principles for Transforming Difficult Tasks into Simple Ones) one must exploit the power of constraints in order to simplify tasks. The following points present how constraints are used in the *Mobia Modeler* in order to assist the user.

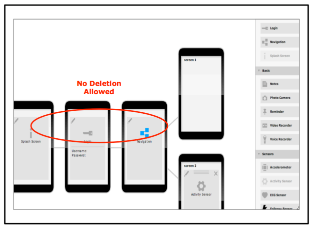
- **Adding Screen and Control Flow via Structure Components.** One example of introducing constraints in the *Mobia Modeler* is by allowing users to add screens only via *structure components* (figure 3.41a). In this way, the user is trained to think about the flow of the application being modeled by discovering which structure component is needed and should be used. However, in the user study [Taf09], a common feedback from participants is that, the use of *structure components* is not intuitive. To ease the user into using such types of components, a short tutorial can be shown or animated help functions


(a) Adding Screen via Structure Components


(b) Disabling Components



(c) Disallowing Explicit Screen Arrangement



(d) Deletion Restriction

Figure 3.41: Task simplification via constraints.

that show how the *structure components* should be used can be added to the modeler for first time users. These help functions can be disabled later on when the user already has some idea on how to use them.

- **Disabling Components.** Another example of constraints in the *Mobia Modeler* is disabling some of the components in the component palette if there are no available screens to put the components in, or if a component is already available in the model and should not have multiple instances in the model (figure 3.41b). An example would be disabling the splash screen component in the palette if it is not the first component in the model. Adding of sensor components is also allowed only once since it does not make sense if the application adds two instances of the same sensor for one application to be used at the same time.

- **Disallowing Explicit Screen Arrangement.** With the *Mobia Modeler*, the screens in the model are automatically added and arranged. The user is not allowed to move or arrange the screens themselves (figure 3.41c). In this way, the user cannot lose time doing unnecessary tasks such as screen layout and concentrate on the task at hand. This was one of the problems in the *Mobia Integrated-View* and *Mobia Multi-View* prototypes wherein the users took too much time with the tasks because they had to explicitly arrange the screens themselves.

  For users who really want to take control in arranging the screens, an alternative solution would be to provide the user with an option to toggle between *auto-arrange mode* and *manual arrange mode*.

- **Deletion Restriction.** In some cases, the components in the model are not allowed to be deleted if there are already components in the consecutive screens that are connected to it (figure 3.41d). This is to prevent users from losing parts of their model during accidental deletion. The trade-off for this design is that, if the user really wants to delete parts of a specific branch in the model, it would be tedious to explicitly delete the components one by one. In the user study, some users got confused why some of the components cannot be deleted [Taf09].

Again, as an alternative, the user should always be allowed to delete any part of the model. In cases where the deletion action can pose serious side affects to the model, a confirmation dialog pops up and informs the user of the consequences of the action. Providing an *Undo* function is another alternative.

Choosing between Modeless and Modeful Interfaces. The use of modeful interfaces has been discouraged by experts in interface design since it may introduce confusion to the user and add unnecessary overhead in switching through the different modes [Tid06]. This was also confirmed with the result of the user study conducted between the *Mobia Integrated-View* and *Mobia Multi-View* prototypes (section 3.2.4), which result was adapted to the succeeding version of the *Mobia Modeler* (section 3.2.5).

However, modeful interfaces can be used in cases where there is a limited screen space in the running platform. One example is the use of modes in the *Mobia Proto-Go* in order to separate the different operations that can be done inside the application (section 3.2.6). However, as mentioned earlier, user studies still have to be conducted in order to evaluate if the current design of the modeful interface for the *Mobia Proto-Go* provides ease-of-use.

3.3.3 Users' Feedback

"The users' feelings or attitudes toward the system."

For each user study, the last phase involves getting information from the user in the form of a questionnaire. This is in order to collect the users' subjective preference or degree of satisfaction [Pat99]. Specific information collected during this phase are user preference (e.g. which UI they prefer) and comments. According to Rosson et al. [RC02], the challenge in this phase of the usability study is the organization and interpretation of the information collected from the participants. In order to simplify this, comments for instance, can be categorized into positive or negative, or can be categorize according to function (e.g. navigation, error recovery, etc.). Aside from that, this can be combined with the raw data collected during the user study (e.g. video of user behavior) in order to fully understand the users' interaction with the system [RC02].

For the *Mobia Integrated-View* and the *Mobia Multi-View* prototypes, since comparison between the two was the goal of doing the user study, the users were asked which of them was easy and more enjoyable to use. 60% of the participants said that the *integrated modeless design* was easier to use. This result correlated to how fast the users were able to carry out the tasks. The users were able to carry out the tasks faster using the *integrated modeless design* of the *Mobia Modeler*. However, with respect to user enjoyment, both versions scored the same.

For the *Mobia Modeler*, the user feedback was not only aimed to get information with regards to the users' feelings towards the tool, but also with regards to their understanding of the concepts and basic idea behind the *Mobia Modeler*. Comparison was made between the feedback from the users with programming backgrounds, and those without any programming background (i.e. non-technical people). As expected, understanding of the general concepts, usability and design approach of the *Mobia Modeler* scored higher in the programmers' group as compared to the non-technical group. Overall, 81% of the participants said they understood the concept of the *Mobia Modeler*, and 88% agreed that the concept of *configurable components* was a good one. 67% said it was easy to use for creating mobile applications [Taf09]. With regards to the overall design of the *Mobia Modeler*, 94% gave the highest rating.

With regards to the added features that the users would like to have in the *Mobia Modeler* from Max Tafelmayer [Taf09], most of them are related to providing additional help to the users in order for them to easily learn how to use the *Mobia Modeler*. Examples of such help functions are: tutorials and guided tours in the beginning, videos that describe each feature of the modeler, and integrated help function that explains what each type of component is for.

3.3.4 Ease of Learning

"The ease of learning the system especially during initial use."

Ease-of-learning for the different versions of the *Mobia Modeler* was measured according to the period of time the users were able to carry out their tasks without any assistance, and combined with the users' subjective feedback with regards to ease-of-use of the tool.

As already mentioned in previous sections, according to ease-of-use, the *Mobia Integrated-View* allowed the user to easily create their models and was therefore the adapted design for the succeeding version of the *Mobia Modeler*.

Additional help functions can also be provided to the user in order for them to have a grasp at how the system works. Tooltips is one example of such function in order for the user to know what a certain component is for. However, during the user studies conducted by Ugur Örgün [Örg09] with the *Mobia Integrated-View* and *Mobia Multi-View*, one observation was that, most of the users did not pay attention to the tooltips [Örg09] which was located at the top (i.e. eye-level) of the modeler. This was a design flaw since most common tooltips usually appear beside the component where the mouse cursor is. Animated hints can be provided to catch the users' attention [Taf09]. However, for more experienced users, this feature can be disabled [Örg09]. Animated clips that show how to use and configure a specific component (e.g. adding screens to a navigation component) would also be helpful.

Another type of hinting mechanism integrated in the design of the *Mobia Modeler* showed a change in a component's color when it is configured. This signifies that the component has been changed from its default configuration. However, one participant expressed that the blue color of the configured components was not that noticeable. In order to make the user be aware of such subtle hints (e.g. change in color), a popup message that explains what is happening should appear during the first time the user encounters such feature.

3.3.5 System Tolerance

"The system's tolerance to unexpected or wrong usage."

The system's tolerance pertains to how the system reacts when unexpected actions are carried out by the user [Pat99]. A usable system should give enough leeway for users to make mistakes [Pat99]. One of the unimplemented features of the different *Mobia Modeler* versions was the *Undo function*. The user is allowed however to delete model constructs when needed. The *Mobia Modeler* allows deletion of components without any warning if there were no changes to the default configuration of a component. However, once the user has modified something in the component, and then tries to delete the component, a warning message is issued. This is to ensure that the user is aware of the consequences (e.g. losing configuration data) of his actions. For unexpected actions (e.g. dragging a component to a screen that is already occupied) that users make, the *Mobia Modeler* simply ignores the action.

3.4 Summary and Discussion

The summary of activities that influence the design and development of the different *Mobia Modeler* prototypes is shown in figure 3.42.

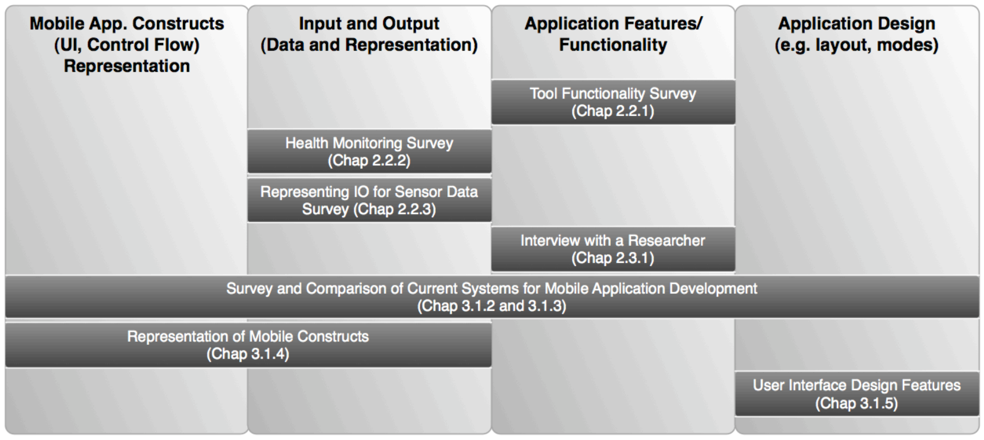

Mobile App. Constructs (UI, Control Flow) Representation
Input and Output (Data and Representation)
Application Features/ Functionality
Application Design (e.g. layout, modes)
Tool Functionality Survey (Chap 2.2.1)
Health Monitoring Survey (Chap 2.2.2)
Representing IO for Sensor Data Survey (Chap 2.2.3)
Interview with a Researcher (Chap 2.3.1)
Survey and Comparison of Current Systems for Mobile Application Development (Chap 3.1.2 and 3.1.3)
Representation of Mobile Constructs (Chap 3.1.4)
User Interface Design Features (Chap 3.1.5)

Figure 3.42: Summary of activities that influence the design and development of the *Mobia Modeler* prototypes.

In order to discover the different ways *mobile application constructs* are represented, a survey and comparison of different systems that allow mobile application development was carried out (sections 3.1.2, 3.1.3 and 3.1.4). Aside from the surveys mentioned, additional surveys were made in order to find out the different *input and output data* needed for mobile health monitoring applications (section 2.2.2) and how they are represented in the development environment (section 2.2.3).

With regards to the *application features* for the tool, a survey with potential users (section 2.2.1) and an interview (section 2.3.1) were carried out in addition to the systems survey (sections 3.1.2 and 3.1.3) in order to collect possible features. Aside from looking into systems used for development, a survey of different designs of other software applications was carried out in order to get design ideas (section 3.1.5).

The different usability properties of the *Mobia Modeler* prototypes such as system relevance, task efficiency, user feedback, ease of learning and system tolerance were discussed in detail in section 3.3. The usability of the prototypes developed is very important especially for a tool that targets non-technical people as primary users.

Chapter 4

The Mobia Framework

In this chapter, details about the *Mobia Framework* will be presented. This includes a thorough discussion of the different parts of the framework and an evaluation of the framework against quality attributes such as extensibility, usability, etc. Before going into the details of the *Mobia Framework*, this chapter will first give an overview about some related work in the area of model-driven development.

Contents

4.1 Related Work

Models have played an essential role in the creation of different products whether they are physical such as electronic and mechanical products, or intangible ones such as software. In the former, models have served as direct input to the system (e.g. production line) that allows the automatic construction of the actual products. In software engineering, the same concept is being applied and such paradigm is called *Model-Driven Software Development (MDSD)*[1]. Instead of treating models only as documentation artifacts just as they were in the past, models are now taking center stage in the actual creation of software applications [SVC06]. In this chapter, we use the book from Thomas Stahl and Markus Völter entitled *Model-Driven Software Development* [SVC06] as the main reference to report more about MDSD.

The main goals of MDSD include the following [SVC06]: *development efficiency*, *software quality* and *reusability* through the provision of a thoroughly worked-out and formalized software architecture (i.e. formally-defined models, architecture and transformations). Just as any other development paradigm, MDSD has its different variants which resulted from different goals and approaches from different people in the software engineering community. To name a few of these variants as discussed by Stahl & Völter [SVC06] , this includes: *Model-Driven Architecture (MDA)* [OMG][KWB03] from the Object Management Group (OMG), *Generative Programming* [Cza05], *Software Factories* [GSCK03], *Model-Integrated Computing* [SK97], *Domain-Specific Modeling* [KT08] and *Architecture-Centric MDSD* [SVC06].

4.1.1 MDSD Concepts and Terminology

The following are the main concepts and the terminology of Model-Driven Software Development (MDSD) as defined by Stahl & Völter [SVC06] with a few minor additions from other references.

Models. A *model* represents an abstraction of a system and its environment [SVC06] in order to aid people (e.g. engineers) in analyzing certain parts of the system. Models can be any type depending on the *"kind of information they contain, level of formality they are used, representation, level of abstraction, target users of the models (e.g. engineers, designers) and purpose (e.g. describe ways to perform a task)"* [Pat99].

Metamodel. A *metamodel "describes concepts that can be used for modeling the model. Fomalization of models takes place in the form of a metamodel"* [SVC06]. Examples of ways to express metamodels are through UML Profiles. However, metamodels do not necessarily have to be UML-based [SVC06].

[1]Model-Driven Development (MDD) is also a common name for MDSD. However, Stahl & Völter [SVC06] prefer the latter term for its completeness.

Meta Meta Model. A *metametamodel "describes the concepts available for metamodeling"* [SVC06].

Relationship: Models, Metamodels and Metametamodels. As already defined, a *model* represents an abstraction of a certain system and its environment [SVC06]. In order to describe the concepts used for *modeling*, a *metamodel* is used. On the other hand, in order to specify the concepts used in *metamodeling*, a *metametamodel* is used.

To simply put it, *a model is an instance of a metamodel, which in turn is also an instance of a metametamodel.* See figure 4.1 describes the relationships between the models.

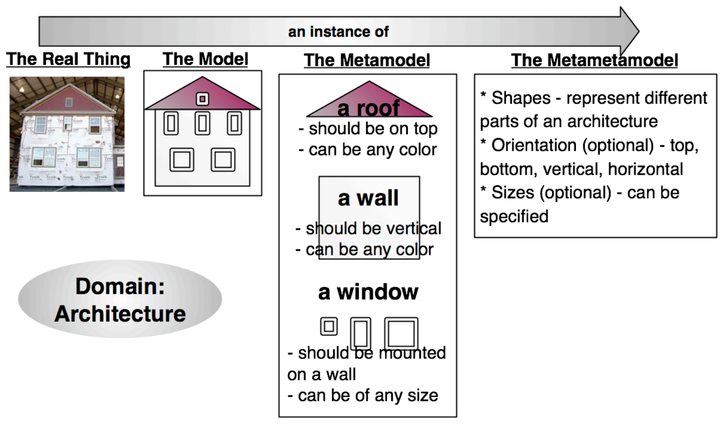


Figure 4.1: An example illustrating the different model relationships.

Abstract and Concrete Syntax. The *concrete syntax* of a language specifies what the parser for that language accepts. It may be in textual or graphical form and is basically the interface to the modeler [SVC06]. For example, in UML, *the boxes and arrows* compose the concrete syntax.

The *abstract syntax* of a language specifies the structure for that language [SVC06]. For example, in UML, *constructs such as classes, attribute operation, association, etc., and the relationship between these constructs* are examples of the abstract syntax.

"The concrete syntax is the realization of the abstract syntax" [SVC06].

Static Semantics. The *static semantics* of a language describe the structure and determines the criteria for well-formedness of a language [SVC06]. An example static semantics in programming languages would be the declaration of variables before using them in a program. Static semantics are important in the context of MDSD because of *"their role in detecting modeling errors in terms of the formalized domain"* [SVC06].

UML Profiles. A *UML profile* is a specification used to extend UML models in order to customize the language to specific areas (i.e. domains and platforms)[2]. It is an extension of the UML metamodel [SVC06]. According to the UML Profile specification[3], UML profiles can be used to:

- Identify a subset of the UML metamodel.
- Specify well-formeness rules (i.e. set of constraints written in UML's Object Constraint Language to check if a certain model is valid or not) beyond those that were already specified by the UML metamodel.
- Specify standard elements which is used to describe a standard instance of a UML stereotype, tagged value or constraint.
- Specify semantics expressed in natural language.
- Specify common model elements.

Domain. A *domain* is *"an area of interest to a particular development effort"* [KT08]. Domains can be *horizontal* (e.g. user interfaces, communication or transactions), or *vertical* (e.g. banking, insurance or robot control) [KT08].

Domain Specific Languages. A *domain specific language (DSL)* is used to formally express and model the key aspects of a certain domain (i.e. adopts concepts from the problem space) [KT08][SVC06].

The DSL is defined as a metamodel [KT08] which includes the static semantics and corresponding concrete syntax [SVC06]. *"The semantics of a DSL must either be well-documented or intuitively clear to the modeler"* [SVC06].

Tool support is usually provided for DSLs. *"Two types of DSL editors are UML tools that are configured via a profile or custom-made DSL-specific tools"* [SVC06].

Formal Models. A *formal model* in the context of MDSD is formulated using a DSL's concrete syntax in which meaning is obtained from the DSL's semantics, and is connected to a specific domain [SVC06].

Models and Transformations. *Models* are the main artifacts in the development of software applications in MDSD. The *transformation* of these models to the actual code is done automatically. Transformation may occur directly from model to code, or may include intermediate steps such as *model-to-model* transformations and then *model-to-code.*

However, Stahl & Völter [SVC06] recommend against *"explicitly visible and manipulable intermediate results"* of transformation. This means that if possible, other transformations (i.e. intermediate transformations such as model-to-model) must be hidden in order to avoid consistency problems.

[2]http://www.uml.org/
[3]http://www.omg.org/technology/documents/profile_catalog.htm

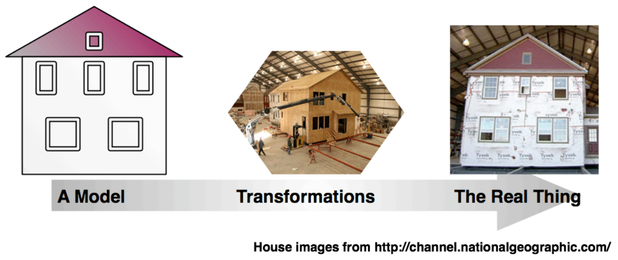


Figure 4.2: A real world example of models and transformations

Stahl & Völter [SVC06] also advise against source code to model transformations (i.e. reverse engineering). MDSD promotes *forward engineering* which means that modifications needed should be done to the model which is then automatically transformed to code and not the other way around. According to Stahl & Völter [SVC06] *"since architecture-centric MDSD models require real abstractions, reverse engineering is either not possible or does not make sense. Design changes have to be made to the actual design (i.e. model). Thus the model will always be consistent with the generated source code"*.

Domain Architecture. A *domain architecture* is the combination of the metamodel of a domain, a platform, the corresponding transformations, and the implemented idioms which are essential in order to transform (either partially or fully automated) the model into a product [SVC06].

Software System Families. A *software system family* is the set of products that can be made using a certain *domain architecture* [SVC06].

Software Product Line. A *software product line* refers to a system (i.e. methods, tools, techniques) that can be used to create a collection of similar software products (i.e. software system family) [SVC06][GSCK03].

4.1.2 MDSD Variants

As mentioned in the introduction, MDSD has different variants, or as Stahl & Völter [SVC06] put it, *flavors* which arose from different goals and approaches from people in the industry. The similarities between them lie in the *use of models as primary development artifacts and the automated process of conversion from these models to other forms* (e.g. another model or code).

Model-Driven Architecture

The *Model-Driven Architecture (MDA)* [OMG][KWB03] is an initiative from the Object Management Group (OMG) to standardize technologies (e.g. models, trasformations) related to MDSD in order to achieve interoperability. There are two types of formal models specified in MDA which are:

- **The Platform-Independent Model (PIM)** which defines domain-related concepts in the model and is independent of implementation. It uses UML constructs adapted via UML profiles.
- **The Platform-Specific Model (PSM)** which contains modeling constructs that use concepts from the target platform in order to describe the system. For example, if the target platform uses Java as the source code, the terms classes, attributes and methods may be used in the PSM. One can have multiple PSMs if there are many target platforms.

The typical sequence of transformation is usually from PIM to PSM, and then PSM to source code. The recommendation is that transformations should be done in several steps. This is to allow a series of refinement as each model is transformed from one form to next until it takes its true form which is the target code.

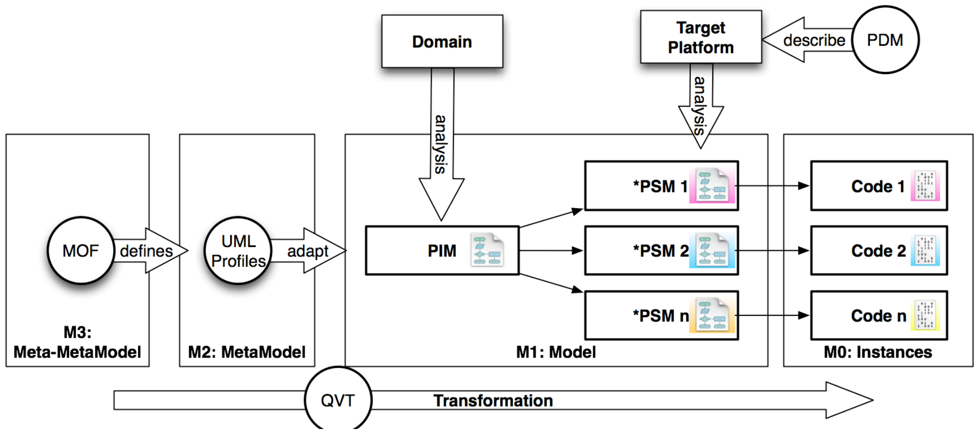


Figure 4.3: The MDA Approach showing some of the recommended technologies adapted from Stahl & Völter [SVC06] and Kleppe et al. [KWB03]. The asterisk (*) in the PSMs indicate that there may be other intermediate steps (other model-to-model transformations) before the final model-to-code transformation.

As mentioned, MDA aims to standardize specific types of technologies. Examples of such technologies are:

- The use of *Meta-Object Facility (MOF)*[4] for the metametamodel. DSLs are expected to be based on MOF.

[4]http://www.omg.org/mof/

- The recommended use of *UML profiles*[5] as the concrete syntax for the DSL.
- The specification of static semantics with the use of *Object Constraint Language (OCL)*[6] expressions.
- The use of *Query/View/Transformation (QVT)*[7] language to specify transformations between the different artifacts (i.e. models, code).
- The use of *Platform Description Model (PDM)* as a metamodel for the target platform.

The MDSD differs from MDA in a way that, instead of focusing on the technologies used, MDSD provides *"modules of software development* ***processes/methodology*** *that can be applied in the context of model-driven approaches"* [SVC06].

Domain-Specific Modeling

Domain-Specific Modeling [KT08] aims to reduce complexity in the modeling process by introducing domain-specific concepts in the actual model. It is strictly against the use of general purpose modeling approaches (e.g. UML) and tries to restrict development only to certain kinds of applications. This is because *"focusing on a narrow area of interest makes it possible to map the language closer to the actual problem space and makes full code generation realistic"* [KT08] .

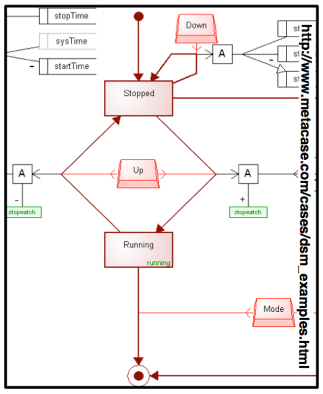
Some examples of domains that widely use DSM are automotive manufacturing, digital signal processing and development of consumer devices. Most DSM approaches are made in-house and typically less widely publicized [KT08] .

Nowadays, DSM techniques are characterized by a high level representation of models close to the target domain and support for full code generation. One example shown in figure 4.4a is the modeling of the application behavior for a wristwatch [KT08]. The term and graphical representation of a *button* is used to symbolize the watch button. The *states running or stopped* are used to indicate the state of a watch at a certain time. Another example shown in figure 4.4b is for modeling Symbian/S60 applications. It uses concepts such as messages, calling, etc. in order to model the whole application. Both of these examples were modeled using the MetaEdit+ modeler from [Meta].

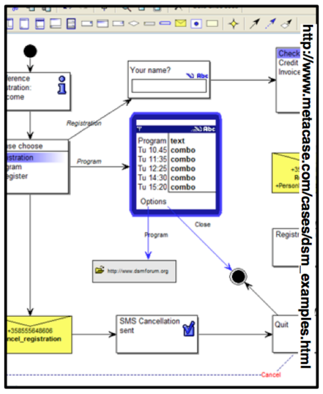
Software Factories

A *Software Factory* [GSCK03] is a software product line that aims to create complete software applications through the use of configurable tools that provide reusable templates, processes and content. These templates are based on a *software factory schema* which is like a *"recipe for building a family of software products"* [Gre04]. In order to develop

[5]http://www.omg.org/technology/documents/profile_catalog.htm
[6]http://www.omg.org/technology/documents/formal/ocl.htm
[7]http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF_QVT



(a) A watch model



(b) A mobile application model

Figure 4.4: Example DSM approaches (e.g. mobile application and wristwatch application) using the MetaEdit+ Modeling Tool from Metacase [Meta].

other variants of the product, existing components are automatically adapted, assembled and configured based on the new specifications. In this way, *"only a small part of the application needs to be developed from scratch"* [GSCK03].

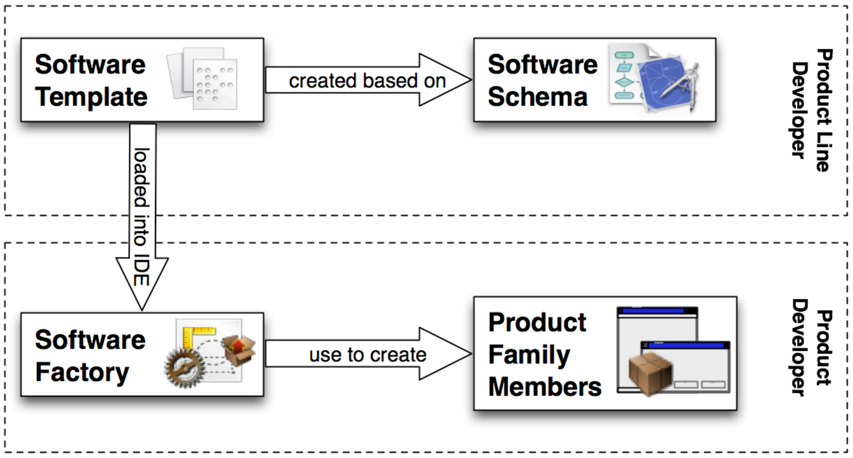


Figure 4.5: An overview of software factories adapted from Greenfield et al. [GS03]. The boxes shows the different types of developer roles involved in creating the artifacts for each phase.

A *software factory schema* includes a description of the categories and relationships between the different development artifacts (e.g. models, configuration files, manifest files, XML documents, etc.). A *software factory template* is composed of a collection of resources (e.g. code, metadata) that is configured and loaded into the IDE in order to allow the easy development of applications for a certain type of software family.

Software factories are related to MDSD in a way that it combines the MDSD approach together with ideas from software product lines and component-based development in order to easily create software applications in a cost effective way [GS03]. One of the focuses though is not just on the processes but also the *support of extensible tools* used to quickly create software factories for specific domains.

Generative Programming

Generative Programming (GP) [Cza05] aims to create complete software products from a set of specifications that are processed automatically to create the final product. One key characteristic of GP is the focus on the creation of software system families. Features of these products are modeled on the basis of the analysis of a certain domain via Domain-specific languages (DSL). The DSLs may be textual or graphical in form.

The GP process follows a *feature-oriented approach* in which models are created by capturing common and variable features of a system family. The whole process starts by analyzing the domain which the system family belongs to. Feature models are then created as a result of the analysis and are the basis for creating the *solution space* which consists of the architecture and components, and the DSLs which are used to describe the domain-specific concepts and features in the problem space (i.e. domain). The mapping of the problem space (models) to the solution space (components) is done through configuration and automatic generation from model to code [Cza05].

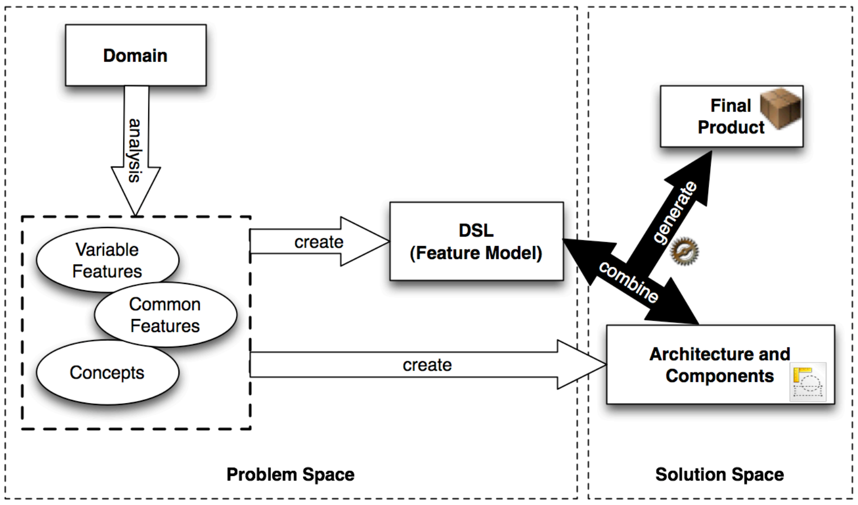


Figure 4.6: The generative programming overview adapted from Czarnecki [Cza05] and Stahl & Völter [SVC06] .

Because of the use of existing atomic components that are optimized for efficiency, GP boasts the advantages of having a faster development speed and better software quality, reusability and maintainability [SVC06].

Generative Programming is closely related to MDSD because of its goal to capture the important aspects of a system through models. It differs from MDSD in terms of its focus on system families. *"In MDSD, system families may be of interest but not regarded as necessity"* [Cza05] .

Model-Integrated Computing

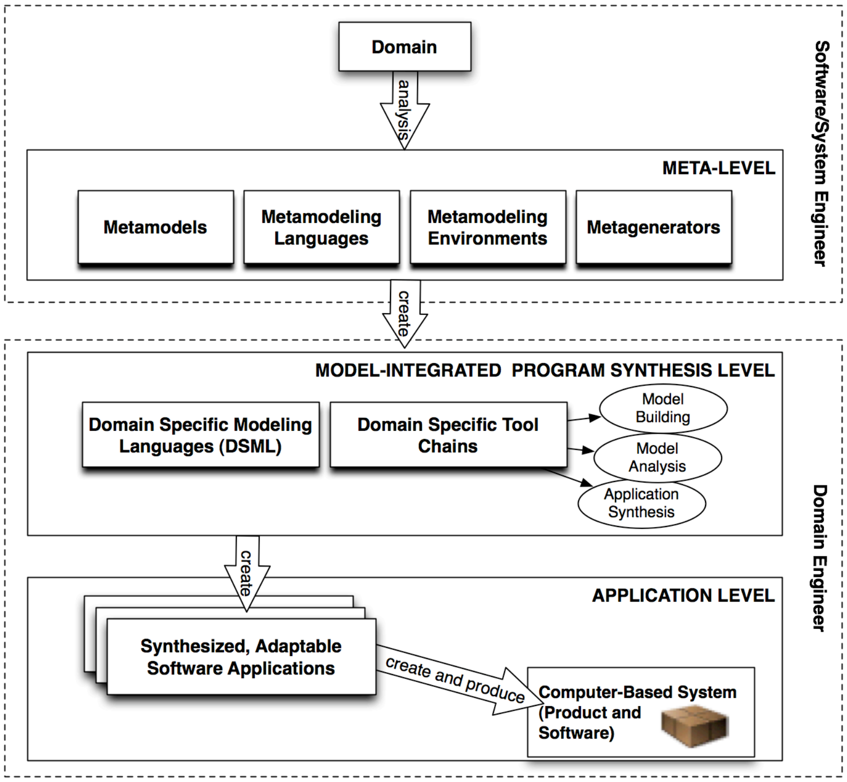

Domain
analysis
META-LEVEL
Metamodels
Metamodeling Languages
Metamodeling Environments
Metagenerators
Software/System Engineer
create
MODEL-INTEGRATED PROGRAM SYNTHESIS LEVEL
Domain Specific Modeling Languages (DSML)
Domain Specific Tool Chains
Model Building
Model Analysis
Application Synthesis
create
APPLICATION LEVEL
Synthesized, Adaptable Software Applications
create and produce
Computer-Based System (Product and Software)
Domain Engineer

Figure 4.7: The MIC Software and Development Process as adapted from [MIC] and Sztipanovits [SK97] . The figure shows the three levels of the MIC process including the different roles that deal with each level.

In *Model-Integrated Computing (MIC)*[8] [SK97], models are used to provide solutions to problems that deal with critical computer-based systems (e.g. avionics control systems, car brake systems). MIC differs from MDSD in a way that, models are not only used during the development phase but also in the analysis, verification, integration and maintenance phases of such systems. Because of this, model-to-model transformation is very important in order to produce models that are essential to each phase of the system lifecycle [SVC06].

Similar to DSM, the first phase of MIC is the analysis of the domain under study. This analysis gives rise to a set of metamodels and its corresponding languages, generators and environments which are then used to create domain specific environments. The use of

[8] MIC has been developed over the years by the Institute for Software Integrated Systems (ISIS) at Vanderbilt University: http://www.isis.vanderbilt.edu/research/MIC.

metamodels in order to describe the modeling tools that provide a well-suited environment to creating solutions for a specific domain, is what differentiates MIC from DSM. This first phase is usually done by the software or system engineers. The domain engineers will then use these environments in the second phase of the development in order to build and analyze domain models, and generate the applications. The tools available in this level also allows the formal analysis, verfication and transformation of models of the computer-based system product [SK97][MIC].

Architecture-Centric Model-Driven Software Development

Architecture-Centric Model-Driven Software Development (AC-MDSD) [SVC06] is another flavor of MDSD which perhaps is the closest in its goals to MDSD. It aims to provide a set of *formal architectural requirements* that would help in improving development efficiency, software quality and reusability of existing software artifacts. Unlike MIC for example, AC-MDSD does not focus on the tools/environments but on the engineering principle.

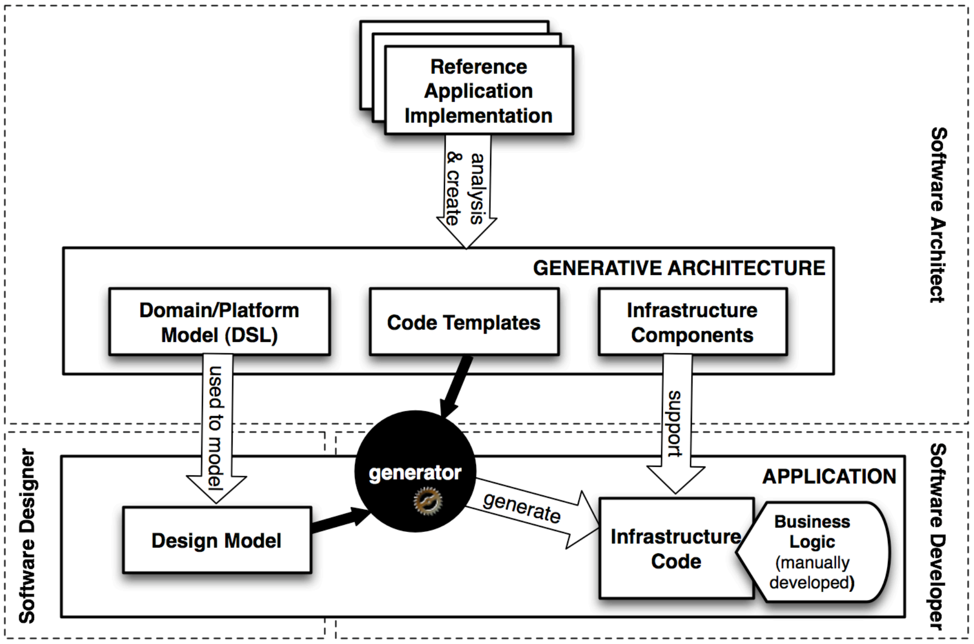


Figure 4.8: The basic idea behind AC-MDSD adapted from Stahl & Völter [SVC06] . It also shows the different roles that deals with the different activities.

As enumerated by Stahl & Völter [SVC06] , the main properties of AC-MDSD are the following:

- AC-MDSD focuses on creation of *software system families and not unique items*, and the reuse of *generative software architectures* of architecturally similar applications.

Generative software architectures use generators in order to automate the tedious schematic and repetitive application development process of an already well-defined family of applications. The input to such system is the domain model of the application, and the output generated is a complete infrastructure code of the application. This generated code is usually the one created via tedious copy/paste/modify process when manual methods are employed.

- AC-MDSD uses a single step model-to-code transformation if possible. When model-to-model transformations is needed for modularization purposes, the resulting intermediate models should be invisible to the application developer.

- AC-MDSD encourages the use of generator templates for code generation.

- AC-MDSD usually generates incomplete products[9]. Architectural infrastructure code is 100% generated, but the non-generated code (individual/domain-related aspects) is manually implemented in a target language. The integration of the generated and non-generated code are done using suitable design patterns.

- AC-MDSD avoids round-trip engineering which basically means that, changes are only made to the model and not the generated code. For parts that need to be implemented manually, some proposed methods by Frankel [Fra03][SVC06] can be used in order to obtain the desired code without round-trip engineering:

  - **Tagging the model.** Code-level decisions are moved to the model by tagging the model with implementation decisions. A negative effect is the contamination of the models with implementation concepts not derived from the domain which can be a potential source of error.
  - **Separation of code classes.** The target architecture is adapted such that manually created code must be written into classes.
  - **Tagging the code.** Some regions of the code are tagged as protected in order to prevent manual changes.

4.1.3 Comparison of the MDSD Approaches and the *Mobia Framework* Approach

Figure 4.9 shows a summary of the different MDSD approaches including MDSD itself. The figure shows the differences and similarities according to how the models are used, the types of transformations, the produced artifacts and other details that each approach might have.

[9] This is valid only for AC-MDSD and not for MDSD in general.

The Use of *Models*. All of the approaches use *models* one way or another. A prerequisite for the models is that they should be formally defined (i.e. assigned syntax and semantics with the use of DSLs) in order for automatic transformation to be attainable.

The approaches differ according to how the formal models are used. For instance, all other approaches except for *Software Factories* use formal models to describe either the framework or the final software application that needs to be created. For *Software Factories*, *DSM* and *MIC*, models are utilized in order to describe the tools and environment that will be used for development.

In the case of the *Mobia Framework*, models are also used as primary artifacts to creating domain-specific mobile applications. Details about the models (concrete and abstract syntax) of the *Mobia Framework* is discussed further in chapter 5.

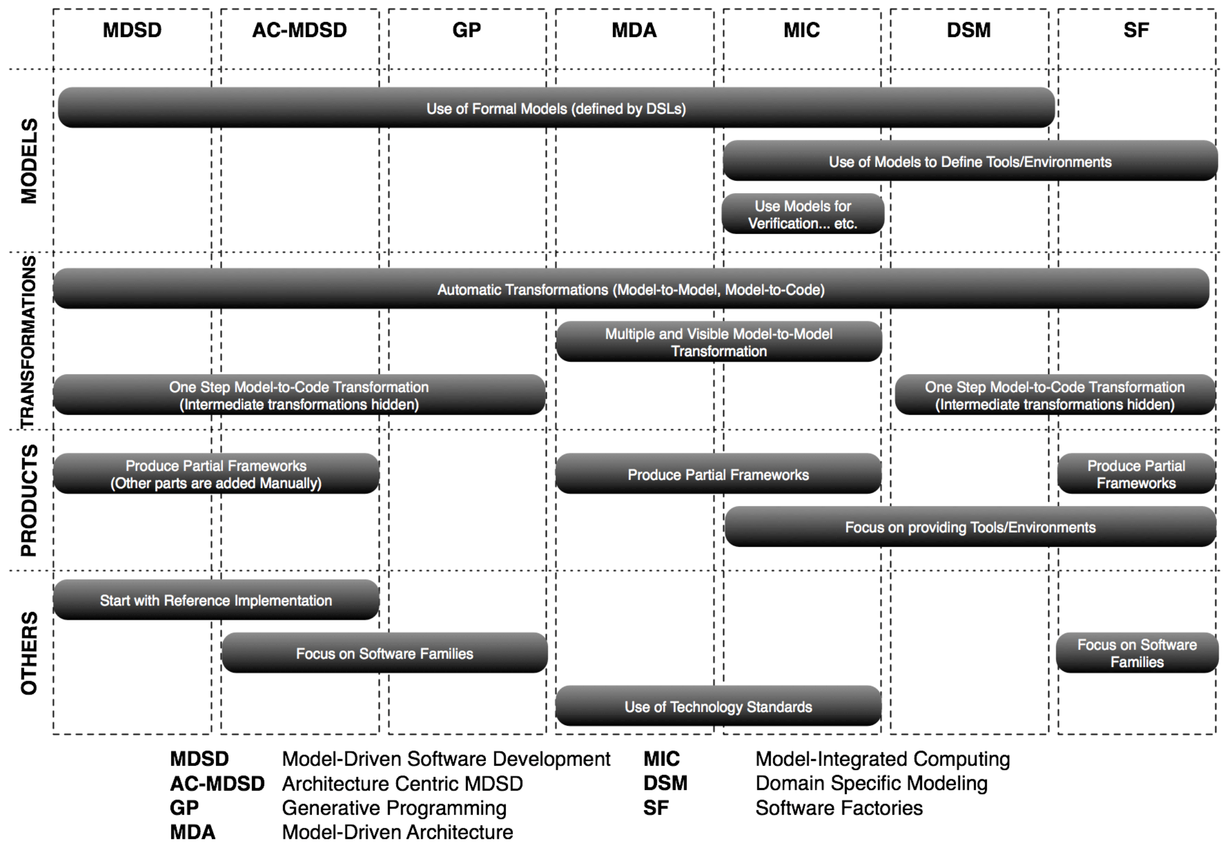

MDSD
AC-MDSD
GP
MDA
MIC
DSM
SF
MODELS
Use of Formal Models (defined by DSLs)
Use of Models to Define Tools/Environments
Use Models for Verification... etc.
TRANSFORMATIONS
Automatic Transformations (Model-to-Model, Model-to-Code)
Multiple and Visible Model-to-Model Transformation
One Step Model-to-Code Transformation (Intermediate transformations hidden)
One Step Model-to-Code Transformation (Intermediate transformations hidden)
PRODUCTS
Produce Partial Frameworks (Other parts are added Manually)
Produce Partial Frameworks
Produce Partial Frameworks
Focus on providing Tools/Environments
OTHERS
Start with Reference Implementation
Focus on Software Families
Focus on Software Families
Use of Technology Standards
MDSD Model-Driven Software Development
AC-MDSD Architecture Centric MDSD
GP Generative Programming
MDA Model-Driven Architecture
MIC Model-Integrated Computing
DSM Domain Specific Modeling
SF Software Factories

Figure 4.9: MDSD's relationship with respect to the variants discussed.

The Use of Technology Standards. The different approaches differ with regards to which technologies are used to create the models. Some approaches such as *MDA* and *MIC* recommend the use and standardization of specific technologies (e.g. UML, MOF), while others have their own representations of the models. According to Stahl & Völter [SVC06] , the use of these so-called industry standards can sometimes pose as a disadvantage because sometimes they are often unusable in the reason of lack of practical experiences in using such technologies, and also because they are overly complicated since the proposed standard must satisfy all the vendors who are part of the group that define them.

During the development of the *Mobia Framework*, technology standards were not used for the purpose of exploring new technologies that can be utilized to create the different parts of the framework.

The Basis for the Models. Approaches such as *MDSD* and *AC-MDSD* recommend the use of a *reference implementation* as the basis for the models. Other approaches base the model on the software family it belongs to.

For the *Mobia Framework*, the models were based on the combination of the problem domain being explored which is *mHealth*, the mobile constructs looked into (section 3.1.4) and the different systems offering similar goals for allowing development for mobile applications (section 3.1.2). The reference implementations were used as the *basis for the code templates for model-to-code transformation.*

The Transformations. All MDSD-related approaches are characterized by automatic transformation whether it is model-to-model or model-to-code. They differ however in terms of the *visibility of the intermediate transformations and produced artifacts (e.g. models, source code).*

The *MDA and MIC* encourage the use of multiple transformations resulting to visibly modifiable artifacts. In MIC, the purpose of having this multiple transformation is in order to utilize the models in the different phases of the system lifecycle such as analysis, verification, integration and maintenance. This multiple transformation is important especially when interoperability between different platforms is a concern. However, this *"may lead to potential consistency problems and is not as efficient as compared to one-step approaches (i.e. model to code)"* [SVC06]. In cases where it is not possible to have this one-step approach, intermediate steps and results should be hidden [SVC06].

Since the target users of the tools provided by the *Mobia Framework* are non-technical users, transformations are made hidden from the end-user as possible. From the end-user's point of view, the transformation is a one step process which includes importing the model to the processor and having the final deployable application as end-product. Transformations in the *Mobia Framework* will be discussed further in section 4.2.4.

The Produced Artifacts. In terms of the resulting products from the different MDSD approaches, most of them produce partial frameworks which can be completed by manually adding the non-generated artifacts, or by assembling the produced parts in order to create the complete application. However, 100% code generation is also possible but may be less flexible [SVC06]. Approaches like MIC, DSM and SF also focus on producing tools and environments from the models which is very helpful for developers and domain experts.

As already mentioned, since the target users of the *Mobia Framework* are non-technical users, the produced artifacts include a fully deployable application. However, in order to make the tools also usable to professional developers, the complete source code which can be imported to IDEs that supports the development of the target platform (e.g. Eclipse IDE with plugins for Android Development) is also made available so that additional enhancements can be made to the application when needed.

4.2 The *Mobia Framework*

The *Mobia Framework* is a domain-specific modeling framework for mobile applications. The two main parts of the *Mobia Framework* are the *Mobia Modeler* and the *Mobia Processor* (figure 4.10).

The *Mobia Modeler* acts as the front end of the *Mobia Framework*. It is a tool which is designed to easily create mobile applications through modeling. The *Mobia Processor* is responsible for transforming the underlying models to the target code. As proof of concept, the Android Framework[10] was chosen as the target platform in which the models are transformed into.

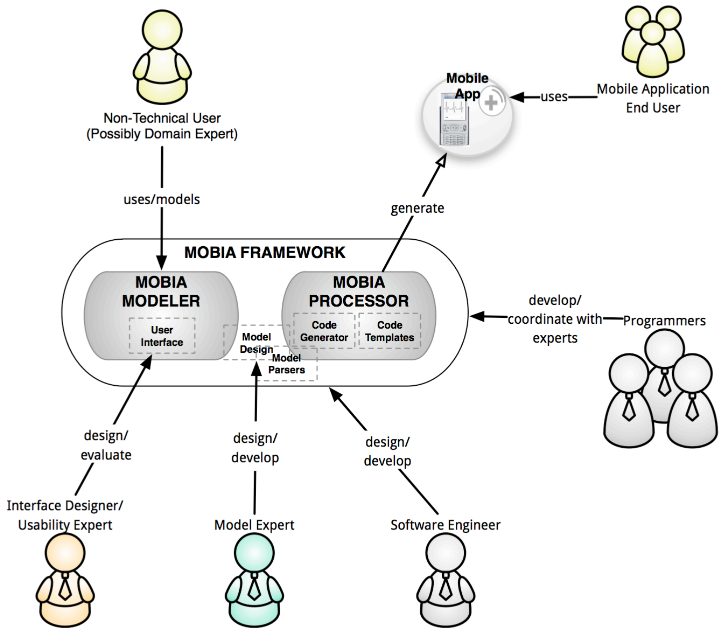


Figure 4.10: An overview of the *Mobia Framework* parts and its use cases.

[10] http://developer.android.com

4.2.1 *Mobia Framework* Use Cases

Figure 4.10 shows a use case diagram for the *Mobia Framework*. As seen in the figure, the top-left part shows the end-user of the modeling tool (i.e. the *Mobia Modeler*) who is a non-technical user (i.e. has little or no knowledge in programming). He uses the *Mobia Modeler* to model the mobile application he wants to create. The other type of user shown in the top-right part of the figure is the end-user for the mobile application generated by the *Mobia Processor*. The lower part of the figure shows other possible stakeholders and their roles in the creation of the *Mobia Framework*.

- **The Interface Designer** is responsible for researching and designing the appropriate user-interface for the *Mobia Modeler*. He works together with the programmer who will write the actual code for it. The user-interface designer also works together with the model expert in order to come up with designs for the model artifacts.
- **The Usability Expert** is responsible for doing the necessary user studies to evaluate the *Mobia Modeler*. He works closely with the interface designer in order to come up with usable designs for the user-interface and the interaction methods.
- **The Model Expert** is responsible for identifying the patterns for the application and translating it into models.
- **The Software Engineer** is responsible for designing the *Mobia Framework* and identify the different functional parts of the framework. He needs to ensure that the framework is designed such that it would be easy to extend later on. He works closely with the programmers in order to develop the whole framework.
- **The Programmers** are responsible for developing the different parts of the *Mobia Framework*. Some programmers may be assigned in the development of the *Mobia Modeler*, while the others are responsible for creating the *Mobia Processor*. Others who are experts in developing specific applications for specific mobile platforms may be responsible for developing the reference implementations of the Mobia components for that specific mobile platform. The code templates will then be created based on these reference implementations.

In order to allow the different people to work together, there must be a common artifact that they all get to work on in order to make everything consistent. In this case, the *model* is the central part of the whole framework. The idea of having *"models as the central hub"* was envisioned by Pleuss et al. [PVH07] in order to integrate the different expertise from stakeholders and the different views of the system. Having the model as the basis for the different phases of development ensures that the artifacts produced even by the different tools are consistent with each other [PVH07]. In the case of this research, the metamodels which will be discussed further in chapter 5, are the ones that will allow different people to work on the different parts of the *Mobia Framework* (i.e. *Mobia Modeler* and *Mobia Processor*).

4.2.2 An Application Example: *Health Monitor*

In order to easily explain the different aspects of the *Mobia Framework*, an example scenario and application will be given. The scenario is inspired by the scenario from Leijdekkers et al.[LG06] that uses Smart phones and Biosensors for monitoring patients with heart problems during rehabilitation [BFTH10].

The Application Scenario

Suppose a doctor wants to monitor his clinically obese patient by keeping track of the patient's nutrition, physical activities and heart rate. He wants to ensure that the patient is eating the right foods and doing the assigned exercises (e.g. walking for 30 minutes a day) by getting a daily update on the patient's food intake and physical activities. Since the patient just recently had a heart attack, the doctor wants to ensure that the patient's heart rate does not go over 120 bpm for the next 30 days. In case this happens, the doctor would like to be notified by receiving an SMS through his mobile device. He also configures the application to call an emergency number when the patient's heart rate goes up beyond 150 bpm [LG06]. Figure 4.11 shows an overview of the application scenario.

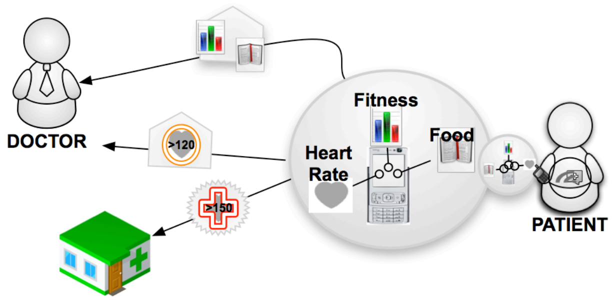


Figure 4.11: An example application that allows a patient to keep track of food intake, physical activities and heart condition

The Application Instance

The expected functionalities of the mobile application just described are the following:

- **Login Screen.** In order to verify the identity of the user (i.e. the right patient is using the application), a login screen is required. The patient should input his login name and password in order to use the application.
- **Fitness Monitor.** A diary which allows the user to input his daily fitness activities.
- **Nutrition Monitor.** A diary which allows the user to input his daily food intake.

- **Heart Monitor.** An application which is coupled with a heart rate monitor in order to keep track of the patient's heart rate. This application invokes certain actions (e.g. sending SMS, calling a number) depending on the heart rate data collected.

Figure 4.12 shows an example instance of the mobile application running on an Android emulator[11]. Instead of using an actual heart rate monitor and processing its inputs, the heart rate is simulated as a text field inside the mobile application where one can input an integer representing the current heart rate.

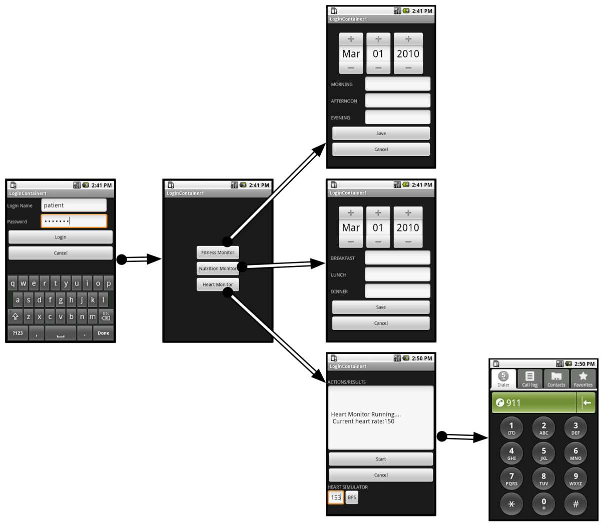


Figure 4.12: An example mobile application instance running on an Android emulator.

Application Usage. The doctor will require the patient to use this mobile application with its different functionalities in order to ensure the patient's health is well monitored even if he is not in the hospital.

The patient has to log in first using the user name and password the doctor has assigned to him. The doctor is the one to configure the application such that it would accept such input. Every time the user is done with a certain fitness activity, he logs this information using the *Fitness Monitor* in the application. After every meal, he also logs what he ate using the *Nutrition Monitor* application. The patient also needs to use a separate heart monitor sensor/device that works with the application. Together with the *Heart Monitor* application, the heart rate of the patient is always monitored such that it is within the healthy normal range which is set beforehand by the doctor during configuration.

The Application Model

A complete version of the model created using the *Mobia Modeler* is shown in figure 4.13.

[11] http://developer.android.com/guide/developing/tools/emulator.html

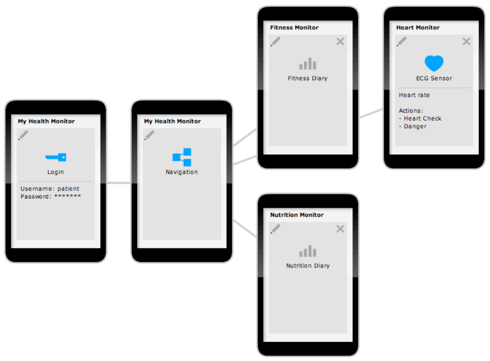


Figure 4.13: An example model created using the *Mobia Modeler* for the application scenario described in section 4.2.2.

4.2.3 The *Mobia Modeler*

Functional Requirements[12]

Expected Users. The expected users of the *Mobia Modeler* are non-technical people who may be experts in their domain. For instance, in the domain of *mobile health monitoring*, example users could be: doctors and nurses in the medical field, researchers in the medical research field, or just an ordinary person who wants to create his/her own *mobile health monitoring* application [BFH09a].

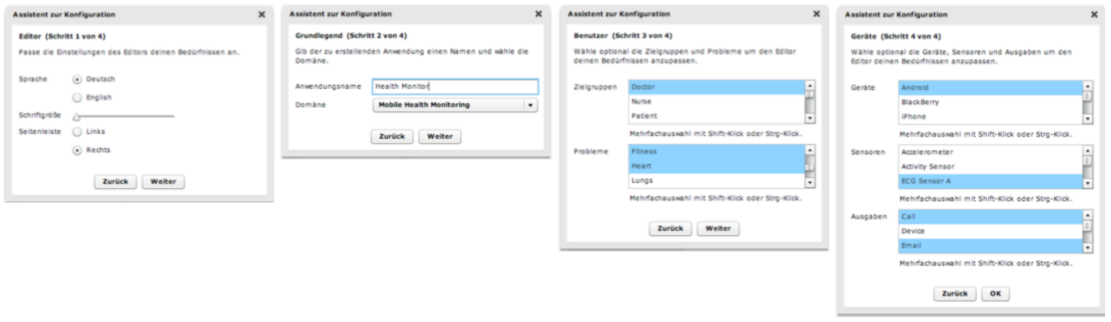
Application Requirements. As of the moment, the *Mobia Modeler* can only model applications in the domain of *mobile health monitoring*. Example applications in this area and the motivation to allow non-technical users create their own mobile domain-specific applications, and possible use cases in the area of personal care, medical research and health institutions (e.g. clinics, hospitals) [BFH09c] were initially discussed in the motivation section in chapter 1.

***Mobia Modeler* Inputs.** The inputs to the *Mobia Modeler* that are supplied by the end-user of the modeler are the following:

- **Information to configure the *Mobia Modeler* interface.** To give a concrete example, consider the example application discussed in section 4.2.2. The example wizard dialogs are shown in figure 4.14.
  Suppose the doctor in the said scenario speaks German. He might want to configure the interface of the *Mobia Modeler* to be in Deutsch. He would then select the target

[12] According to Sommerville [Som04], *functional requirements* specify the services a certain system should provide. This includes details about the inputs, outputs, exceptions and other details about the functionality of the system [Gli07].

domain to be *mobile health monitoring* and input the name of the application as *"Health Monitor"*.
He would then select the target user of the *Mobia Modeler* to be a *doctor*. The idea behind this is that, the *Mobia Modeler* would present more advance options[13] if the target user has more expertise in the field. As for the *patient problems*, he would select *fitness*, *heart* and *nutrition* since these are the important health issues he wants to monitor.
Since the doctor knows he would be providing the patient with a mobile device using the *Android Framework* and an ECG sensor for monitoring, he selects these options in the wizard. He also knows that he only needs the *call*, *email* and *sms* functions of the phone, so he selects these as well.


Assistent zur Konfiguration
Zurück
Weiter
Assistent zur Konfiguration
Zurück
Weiter
Assistent zur Konfiguration
Zurück
Weiter
Assistent zur Konfiguration
Zurück
OK

Editor (Schritt 1 von 4)
Passe die Einstellungen des Editors deinen Bedürfnissen an.
Sprache: [✓ Deutsch] English
Schriftgröße
Seitenleiste: Links [✓ Rechts]

Grundlegend (Schritt 2 von 4)
Gib der zu erstellenden Anwendung einen Namen und wähle die Domäne.
Anwendungsname: Health Monitor
Domäne: **Mobile Health Monitoring**

Benutzer (Schritt 3 von 4)
Wähle optional die Zielgruppen und Probleme um den Editor deinen Bedürfnissen anzupassen.
Zielgruppen: [✓ Doctor] Nurse Patient
Mehrfachauswahl mit Shift-Klick oder Strg-Klick.
Probleme: [✓ Fitness] [✓ Heart] Lungs
Mehrfachauswahl mit Shift-Klick oder Strg-Klick.

Geräte (Schritt 4 von 4)
Wähle optional die Geräte, Sensoren und Ausgaben um den Editor deinen Bedürfnissen anzupassen.
Geräte: [✓ Android] BlackBerry iPhone
Mehrfachauswahl mit Shift-Klick oder Strg-Klick.
Sensoren: Accelerometer Activity Sensor [✓ ECG Sensor A]
Mehrfachauswahl mit Shift-Klick oder Strg-Klick.
Ausgaben: [✓ Call] Device [✓ Email]
Mehrfachauswahl mit Shift-Klick oder Strg-Klick.

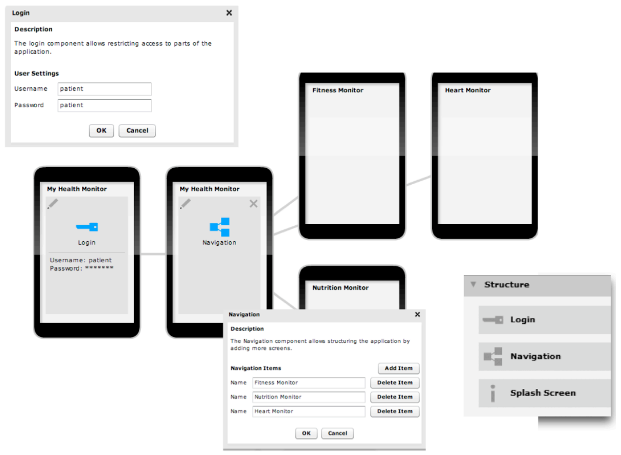
Figure 4.14: Example input to the initial configuration wizard.

- **The selected model components and their configuration.** To give a concrete example, consider the example application discussed in the previous chapter (section 4.2.2). The example components and their respective configurations are shown in figure 4.15.
  The doctor needs a model component that represents a *login* dialog in the application. In order to do this, he chooses the component in the sidebar where all available *Mobia Modeler* components are displayed. After the component is added to the model, the doctor has to add configuration information to the *login component* such as the *user name* and *password* in order to fully configure the component.

***Mobia Modeler* Basic Functionality.** The following points describe the supported features of the the *Mobia Modeler*.

- **Initial Configuration Wizard.** The *Mobia Modeler* presents a configuration wizard during the initial start of the application. This allows the user to configure the interface of the *Mobia Modeler*, select the domain, supply the application name

[13] This type of adaptation for the *Mobia Modeler* interface is currently not implemented in the *Mobia Modeler*.



Login

Description

The login component allows restricting access to parts of the application.

User Settings

Username: patient
Password: patient

Username: patient
Password: *******

Navigation

Description

The Navigation component allows structuring the application by adding more screens.

Navigation Items

Name: Fitness Monitor
Name: Nutrition Monitor
Name: Heart Monitor

Figure 4.15: Example model components and their configuration.

and other information about the application he wants to model (e.g. target users, problems, devices, sensors, outputs).

- **User Interface Customization.** The *Mobia Modeler* allows the user to customize the user interface of the modeling environment. The supported customization features are font size adjustment, sidebar location (where the available components are shown) and language (German or English).

- **Content Adaptation.** The *Mobia Modeler* has currently 20 model components available and are displayed on the sidebar. However, in the future, as the number increases, this will pose a problem in finding the right component to use [Taf09] for modeling. In order to avoid this problem, the modeler is able to adapt which components to display depending on the information inputted for the *Domain*, *Problems* and *Sensors* during the configuration wizard which is displayed when the *Mobia Modeler* is first started.

- **Adding Components through drag-and-drop** The user can add components to the model by dragging components from the sidebar to the main area. Model components can only be dropped on screen instances in the model. When the user attempts to drag a component on an illegal area in the modeler (e.g. not on a screen, or the screen is already full of components), the modeler simply ignores the action.

- **Deleting Components.** The user can delete a component in the model simply by clicking on cross icon (X) at the top-right corner of a component.

- **Configuring Components in a Model.** Once a component is added to a screen, the user can add configuration information to the component by clicking on the pen icon at the top-left corner of a component.

- **Saving the Model.** The model created using the *Mobia Modeler* can be saved in various ways such as:
  - Save on the server where the *Mobia Modeler* is located by pressing on File→Save.
  - Save by downloading the model to the local drive by pressing File→Download.
  - Save by exporting the model by pressing File→Export, and then copying the text on the Export dialog and saving it to a file.
- **Loading the Model.** A previously created model can be loaded to the *Mobia Modeler* in various ways such as:
  - Loading from the server by pressing File→Load and choosing the filename of the model in the *Server* tab of the load dialog.
  - Loading from a local file by pressing File→Load, pressing the *Local* tab in the load dialog, and pasting the model code in the dialog.

***Mobia Modeler* Outputs.** The output of the *Mobia Modeler* which is supplied to the *Mobia Processor* for further processing is:

- **The *Mobia PIM* File.** This is the XML form of the graphical model created in the *Mobia Modeler*. *PIM* actually stands for *platform-independent model* which is a term borrowed from the Model-Driven Architecture (MDA). This represents models that does not have constructs specific to one platform [KWB03][SVC06]. The metamodel of the *Mobia PIM* will be presented later in chapter 5.

Design and Implementation

In chapter 3, the user-centered iterative design, development and evaluation process that were carried out in the development of the *Mobia Modeler* were presented. Particularly in section 3.2.5, the design approach, implementation and other functionalities of the *Mobia Modeler* including the evaluation results were shown.

4.2.4 The *Mobia Processor*

Functional Requirements

Supported Platforms. For the current version of the *Mobia Processor*, the only supported output target platform is the *Android Framework*.

***Mobia Processor* Inputs.** The inputs to the *Mobia Processor* are the following:

- **The *Mobia Metamodel*.** The *Mobia Metamodel* which is in the form of an XML Schema Document (XSD) describes the structure of the *Mobia PIM*. This is needed in order to allow the *Mobia Processor* to know the structure of the *Mobia PIM* in order to process its contents.
- ***Mobia Processor* Configuration Files.** A set of configuration files that contain information about the files and folder locations needed by the *Mobia Processor*. A complete view of the configuration files can be found in appendix A.3.
- ***Android Framework* Code Templates.** A set of code templates that are used as input for generating the final code in the *Android Framework*.
- **The *Mobia PIM*.** The *Mobia PIM* which contains information about the application model and is the main input to the *Mobia Processor*. This is exported from the *Mobia Modeler* and imported to the *Mobia Processor*.

***Mobia Processor* Basic Functionality.** The following points describe the supported features of the the *Mobia Processor*.

- **Transforms the *Mobia PIM* to *Mobia PSM*.** The *Mobia PIM* is processed and information about the model is extracted and stored to an object-equivalent of the *Mobia PIM* which is the *MobiaPimObject* (information about the classes for the *Mobia Processor* can be found in appendix A.4). This information is then stored to a *Mobia PSM* object inside the *Mobia Processor*. *PSM* is actually short for *"platform-specific model"* and is borrowed from the Model-Driven Architecture (MDA). This represents a model that contains constructs specific to a certain platform [KWB03][SVC06]. In this case, the *Mobia PSM* is specific to the Android platform.
- **Transforms the *Mobia PSM* to Android source code.** The *Mobia Processor* transforms the *Mobia PSM* objects to Android source code by merging the information from the *Mobia PSM* to the available code templates.
- **Compiles the generated source code.** The *Mobia Processor* together with the help of additional tools (e.g. Android compiler, Apache Ant) compiles the source code to an installable **Android APK file**.
- **Deploys the compiled mobile application to an emulator.** The *Mobia Processor* provides a way for the user to automatically install the application on the mobile device or emulator[14] through a click of a button.

[14] The emulator must be already be deployed before running the *Mobia Processor*.

***Mobia Processor* Outputs.** The output of the *Mobia Processor* are the following:

- **An installable Android .apk file.** The final output that is visible to the user of the *Mobia Modeler* and *Mobia Processor* is the compiled mobile application which is an installable Android **.apk file**.
- **An Android project.** Other output files which does not concern the end-user of the modeler are the project source files for the Android mobile application. This can be used by an expert user (e.g. programmer) to modify the application by importing the project to an IDE that supports the Android platform (e.g. Eclipse IDE). A complete list of the files can be seen in appendix A.2.

Design

The idea behind the development of the *Mobia Processor* is that we want a framework that abstracts all the technicalities from the end-user as much as possible through the support of full code generation. According to Kelly and Tolvanen [KT08], the success of shifts to current programming languages is because of completely generated executable code without additional manual effort. The same thing is the goal of DSM [KT08][Meta], and which is what we want to achieve with the *Mobia Processor*.

The *Mobia Processor* is designed such that its different parts are separated based on their purpose/function. Figure 4.16 shows the parts of the processor and their links to each other. How the different parts work together will be made clearer as we discuss the transformations in section 4.2.4.

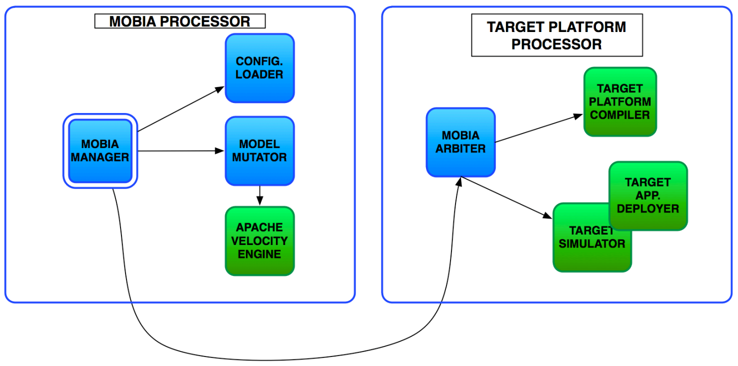


Figure 4.16: The *Mobia Processor* and its parts.

In order to invoke the different parts of the *Mobia Processor*, there should be a part that acts like a manager or control center. In the *Mobia Processor*, such part is called the *Mobia Manager*. It is responsible for assigning all tasks to the subcomponents inside the *Mobia Processor* then loads the information from the model into the runtime system of the processor.

The *Configuration Loader* on the other hand is responsible for loading and processing other information not built into the *Mobia Processor* such as configuration information and other external files.

The core part of the *Mobia Processor* is the *Model Mutator* which is responsible for transforming the model created using the *Mobia Modeler* to its object form in order to allow further transformation into a more platform-specific form of the model.

The *Apache Velocity Engine* is a helper platform used by the *Mobia Processor* to merge information from the model to the code templates in order to generate the final code.

Finally, the *Mobia Arbiter* together with the help of platform-specific tools (e.g. compiler and deployer/simulator) is responsible for compiling the source code and/or deploying the final application.

Implementation

The Technologies Used. The implementation for the *Mobia Processor* uses the *Java SE Platform 6.0*[15] as the main programming platform and *JDOM*[16] for parsing the input files (e.g. XSD files for the metamodel, PIM expressed in XML). In order to merge the code templates with the information from the model, the *Apache Velocity Engine* 1.6[17] is used. A detailed list of the classes and packages for the *Mobia Processor* is listed in appendix A.4.

Currently, the *Mobia Metamodel* which is expressed in *XML Schema Document (XSD)* is used in order to determine the structure of the input *Mobia PIM file* which is exported by the *Mobia Modeler*.

For building and deploying the generated source code, *Android 1.6 release 1*[18] and *Apache ANT 1.7*[19] were used.

The Transformations. The following points will present the different processes inside the *Mobia Processor* which transforms the model created with the *Mobia Modeler* to its final executable code (i.e. Android apk file).

- ***Mobia PIM* and Configuration Importation.**

  The work of the *Mobia Processor* starts when the *Mobia PIM* which is generated by the *Mobia Modeler* is imported into the *Mobia Processor*.

  The *Mobia Manager* loads the information from the *Mobia PIM* into the runtime system of the processor.

[15] http://java.sun.com/javase
[16] http://www.jdom.org
[17] http://velocity.apache.org
[18] http://developer.android.com/sdk/android-1.6.html
[19] http://ant.apache.org

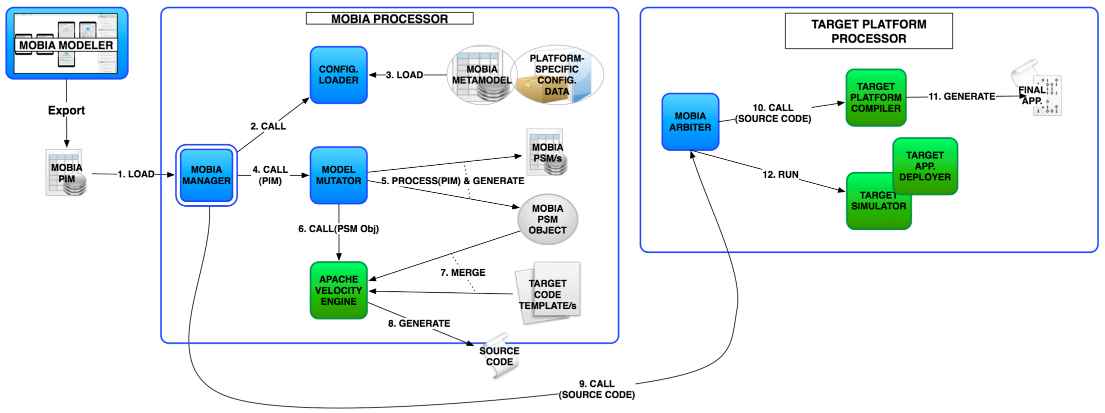


Figure 4.17: From model to code: the *Mobia Processor* in action.

It then calls the *Configuration Loader* which is responsible for loading platform specific information which are stored in the configuration files (see appendix A.3 for details about the configuration files) and also loads the *Mobia Metamodel*.

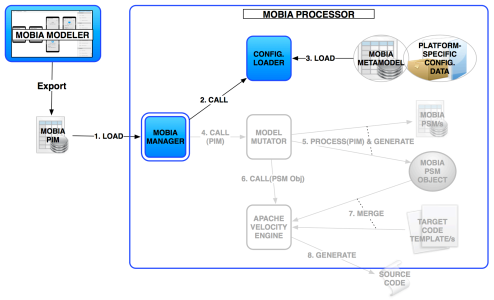


Figure 4.18: *Mobia PIM* and Configuration Importation.

The *Mobia Metamodel* is used by the *Mobia Processor* in order to know the structure of the *Mobia PIM* and prepare an object version of the *Mobia PIM* called the *MobiaPimObject*. Once the structure of the *Mobia PIM* is known, the data from the *Mobia PIM* is extracted and stored inside the *MobiaPimObject*.

- ***Mobia PIM* to *Mobia PSM* Transformation.**

  Instead of directly transforming the *Mobia PIM* to code, we follow the recommended approach from the MDA in which transformation is first done from PIM to PSM [SVC06]. The reason for this is so that we are not tied to one specific target platform [KWB03][OMG]. However, we also follow the recommendation from Stahl & Völter [SVC06] which states that *"explicitly visible and manipulable results should be avoided"*. In the case of the *Mobia Processor*, this means that the *Mobia PSMs* cannot be directly manipulated and only exists inside the running *Mobia Processor*.

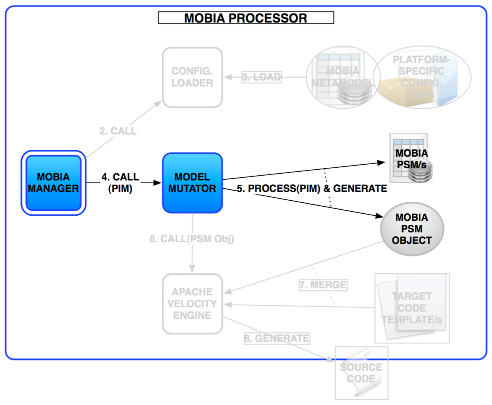


Figure 4.19: The *Mobia PIM* to *Mobia PSM* Transformation.

The *Model Mutator* is responsible for transforming the object form of the *Mobia PIM* (i.e. *MobiaPimObject*) to *Mobia PSM*. Transformation in this context simply means transferring information from the *MobiaPimObject* to an object form of an *Mobia PSM* which is called *MobiaAndroidPsm*.

The *MobiaAndroidPsm* contains information specific to the *Android Framework* such as class names, packages, etc. This is the time where components described in the *Mobia PIM* are assigned their own class names and the package they belong to. This also stores information about which code templates would be assigned to a specific component in the model.

- ***Mobia PSM* to Android Code Transformation.**

  After acquiring information from the *MobiaPimObject* and storing it into the *MobiaAndroidPsm*, the *Model Mutator* then passes the PSM object to the *Apache Velocity Engine*[20] which merges the information from the *Mobia PSM* and the code templates to generate the final source code (figure 4.20).

  This template approach for code generation is suitable specifically for the component-based modeling approach of the *Mobia Framework* because of the following reasons:

  – In the component-based model design employed by the *Mobia Modeler*, there is a *direct mapping between a component and a specific code template*.
  – It is *easier to extend or add more components* by simply adding a code template that would correspond to a specific component. This would not affect the size of the application itself since only the components that will be used are generated.
  – It would be *easier to test the generated code since the code templates would correspond to small but fully functional components*. Before creating the templates, the actual source code for a specific template must first created and tested if it is

[20]http://velocity.apache.org/

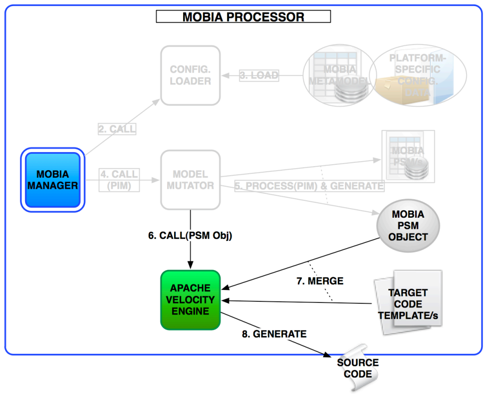


Figure 4.20: The *Apache Velocity Engine* merges information from the *Mobia PSM* object and the code templates to generate the final code.

both syntactically and semantically correct. The template will then be derived from this fully functional and tested piece of code.

Similar frameworks that use template-based code generation include the work from Holleis et al. [HS08] and the SMS Framework from Bartolomeo et al. [BMC+06].

- **Compilation and Deployment.**

  After the code has been generated, the *Mobia Manager* calls the *Mobia Arbiter* (figure 4.21) in order to compile the generated code to the final application. The *Mobia Arbiter* mainly consists of scripts that are necessary in order to run the compilation and deployment tools without the user having to type a command. The *Mobia Arbiter* is also responsible for deploying the application on the mobile device or emulator.

  The *Mobia Arbiter* gets the information with regards to the location of the generated files from the *OutputFolderList.config* configuration file, and the location of the tools used in order to compile the application from the *MobiaConfig.mainconfig* file. The commands used for the scripts are taken from the *MobiaConfig.supportTools* configuration file.

4.3 The *Mobia Framework* Evaluation

It is important to define which quality attributes or non-functional requirements will be used in order to evaluate a software architecture or framework [BZJ04]. However, according to Glinz [Gli07] and Chung et al. [CL09], there is a problem in terms of the concrete *definition*, *classification* and *representation* of non-functional requirements in software engineering. Because of this, it is important for us to choose which ones to adopt in order to be consistent with the chosen definition and classifications [CL09]. In this section, we will

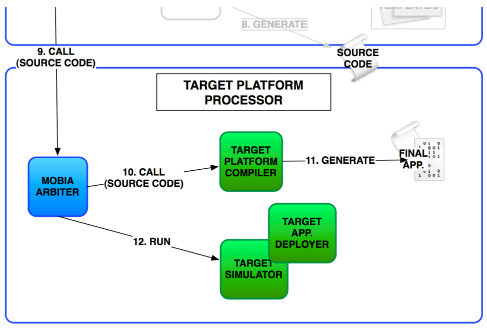


Figure 4.21: Compilation and Deployment.

first define what *non-functional requirements* are and which classifications will be used in order to evaluate the *Mobia Framework*.

4.3.1 Definition of Non-Functional Requirements.

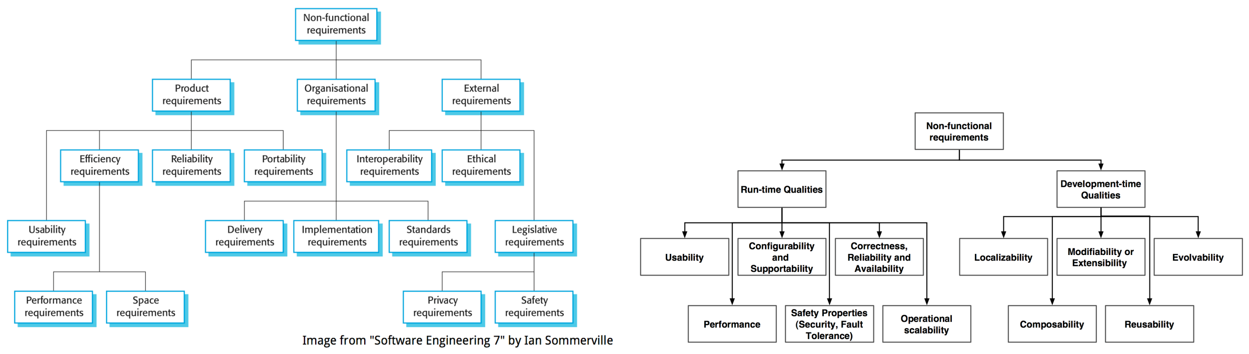
Non-functional requirements are the *constraints (i.e. restrictions or limitations)* and *qualities (i.e. properties, characteristics or attributes)* of a system's functions and services [Som04][Gli07][MB01] that the system *stakeholders will care about and thus affect their degree of satisfaction* [MB01]. Non-functional requirements are very important [Gli07] and may be *"more critical than functional requirements since if they are not met, this may render the system to be useless"* [Som04]. According to Chung et al. [CL09], many of the *"real-world problems are more non-functionally oriented (e.g. poor productivity, slow processing, unhappy customer, etc.)"*.

4.3.2 Classification of Non-Functional Requirements.

Figures 4.22a and 4.22b show classifications of non-functional requirements from Malan [MB01] and Sommerville [Som04]. As seen in the two classification trees, there are similarities between *run-time qualities* [MB01] and *product requirements* [Som04], and also between *development-time qualities* [MB01] and *organizational requirements* [Som04].

For this research, we will put aside the *external requirements* [Som04] which covers those that are derived from factors *external* to the system and its development process (e.g. interoperability with systems from other organizations, legislative requirements, ethical requirements).

Product Requirements or Run-time Qualities. *Product requirements* specify *product behavior* [Som04]. This is similar to the definition from Malan [MB01] which defines *Run-time Qualities* to specify "how well" an *executing system* (i.e. the product) performs relative to the users of the system. In the case of the *Mobia Framework* this pertains to the



(a) (b)

Figure 4.22: (a) The types of non-functional requirements from Sommerville [Som04]. Image taken from [Som04]. (b) The types of non-functional requirements from Malan et al. [MB01]. Image adapted from [MB01].

Mobia Modeler which serves as the front-end of the framework. The non-technical people are defined to be the users of the system.

The following requirements are taken from the classification [Som04] and will be addressed for the *Mobia Modeler*:

- **Usability.** This describes how easy it is to use the system [Som04]. In the case of the *Mobia Modeler*, usability is a big factor since we want the end-users to be able to develop their own mobile applications with ease with the use of this tool. More about usability was discussed in section 3.1.6.
- **Efficiency.** This property includes performance (e.g. how fast the system executes) or space requirements (e.g. how much memory is required) [Som04].
- **Reliability.** This property describes how a system reacts to unexpected interactions by being tolerant to failure [Som04].
- **Portability.** This property describes how a system can easily be used in different platforms other than the one where it is made in (e.g. number of target systems) [Som04].

Organizational Requirements or Development-time Qualities. *Organizational requirements* are derived from policies and procedures used by both the customer and the developers' organization [Som04]. *Development-time Qualities* on the other hand is similar, which specifies the qualities of the work products (e.g. architecture, design and code) and is driven by the development organization's goals [MB01].

In the case of the *Mobia Framework*, it is more applicable to focus on the *implementation* requirements from [Som04]. The *standards* (e.g. process standards that should be used in development) and *delivery* requirements are not really applicable to the research prototypes. The other qualities in the list from [MB01] will be addressed except for the *reusability* (reuse system for future systems) quality. In our case *modifiability or extensibility* is more applicable than reusing the current system for a totally different purpose.

- **Implementation.** This describes the design methods or programming languages used during development [Som04].
- **Localizability.** This describes the ability to make adaptations to the system due to regional changes (e.g. language) [MB01].
- **Modifiability or Extensibility.** This describes the ability to easily add new functionality [MB01].
- **Evolvability.** This describes the ability to add new capabilities or use new technologies [MB01].
- **Composability.** This describes the ability to easily add new components to the system (i.e. plug and play) [MB01].

4.3.3 *Mobia Framework* Evaluation against Non-Functional Requirements

Product Requirements or Run-time Qualities

The front-end of the *Mobia Framework* in which the users mostly interact with is the *Mobia Modeler*. In this section, we describe the product/run-time qualities for the *Mobia Modeler*.

- **Usability.** The usability of the *Mobia Modeler*[21] was evaluated via a user study in which the users were given tasks to perform. Both quantitative information (e.g. task times) and qualitative information (e.g. feedback from the participants) were collected. The details about the usability evaluation and results of the *Mobia Modeler* was described in detail in section 3.2.5.
- **Efficiency.** The efficiency of the *Mobia Modeler* is relative to the system/device (i.e. computer) where it is running.
- **Reliability.** The *Mobia Modeler* is fault-tolerant and ignores any unexpected action from the user.
- **Portability.** The current *Mobia Modeler* is implemented using the Adobe Flex Framework, and can be run on any Flash-enabled web browser.

[21] This does not include the *Mobia Proto-Go* prototype which is not yet evaluated via user study. This tool was developed as proof of concept for the extensibility of the *Mobia Framework*.

Organizational Requirements or Development-time Qualities

For the organizational/development-time requirements, we will discuss the *Mobia Framework*

- **Implementation.** The development of the *Mobia Framework* involved different technologies, processes, and different people (i.e. developers) who carry out the development of the different parts. It is important to separate the development of the two major parts (the *Mobia Modeler* and the *Mobia Processor*) in order to easily divide the work among the developers, and also to allow making use of different technologies that is suitable for each part. What is important is that there is a common underlying model that the different people agrees on in order to allow the model to be translated later on to the target code.

  For the development of the *Mobia Modeler* for example, it involved a more user-centered iterative design and development (see chapter 3 for details). The development technologies used were web-based (e.g. Flash for the initial prototypes [Örg09] and then later on Adobe Flex Framework for the current version [Taf09]). These technologies allowed the creation of a richer and more user-friendly looking interface which can be accessed anywhere. Also the people who created the *Mobia Modeler* were more familiar with the technologies, therefore were more comfortable in developing the prototypes.

  For the *Mobia Processor* part of the framework, the *Java Platform* was used as the major programming platform. Since the design of the code generator relied on a template-based approach, the *Apache Velocity Engine* which is a Java-based template engine was used.

- **Localizability.** The current languages supported by the *Mobia Modeler* are German and English. This requirement is not necessary for the *Mobia Processor* part of the framework since the models created and processed are independent of the localized language used during modeling.

- **Modifiability or Extensibility.** There are different levels in which the extensibility or modifiability of the *Mobia Framework* can be validated. This includes: (1) extensibility with respect to the EUD tools that can be added to the framework to be used by the end users (e.g. EUD tool such as the *Mobia Modeler* and *Mobia Proto-Go*), (2) extensibility with respect to the underlying model, (3) extensibility with respect to additional target mobile platforms for code generation, and (4) extensibility with respect to additional supported model components.

  For the first two, it is easier to explain the *Mobia Framework*'s extensibility through an example implementation as proof of concept. Let us take the *Mobia Proto-Go* tool which was presented in section 3.2.6 as an example.

EUD Tools. The *Mobia Proto-Go* is a supplement mobile-based tool that allows users to create their own mobile applications. The *Mobia Proto-Go* differs from the *Mobia Modeler* in a way that it allows users to customize their own user-interface on the target mobile device itself.

This illustrates the *Mobia Framework*'s extensibility in terms of the tools that can be used together with the *Mobia Processor* part of the framework. The *Mobia Processor* is totally independent of the modeling tool used, as long as the modeling tool follows the same format for exporting the model (i.e. same metamodel). This gives developers of the front-end tools (i.e. EUD tools) the freedom to create tools which may differ in designs and even the running platform as long as it follows how the model should be exported for processing.

Models. Unlike in the *Mobia Modeler*, the *Mobia Proto-Go* allows users to customize their own UI. Because of this, the additional UI information has to be saved in the model in order for the processor to take the UI into account during processing of the model to code. In order to do this, additional *layout* data section is added to the *screen* section of the model without introducing any drastic changes to the model (presented in section 3.2.6 in figure 3.38), which illustrates the extensibility of the models in the *Mobia Framework*.

Listing 4.1: A snippet of the code used to check if layout information should be taken from the model or the default code templates.

```
if( screen.get(Screen.Property.screenLayout) != null ){
  activity.setGeneralProperty(  AndroidActivityContainer.Property.
      optionalLayout,
        screen.get( Screen.Property.screenLayout ).get(0).toString(); //layout
             data
       );
}
```

In terms of generating code for the additional layout, only a few lines of code were incorporated to the *Model Mutator* (inside the class *MobiaAndroidPsmInstanceLoader*) such as shown in listing 4.1 in order to inform the *Apache Velocity Engine* and *Mobia Arbiter* later on to use the new layout instead of the available code templates. Additional code to the *Apache Velocity Engine* and *Mobia Arbiter* also had to be added to take into account the new layout files.

Target Platforms. As for extending the *Mobia Processor* for other target platforms, the *Mobia Framework* can be extended by extending the *mobia.psm.Mobia PsmInstanceLoader* for the new target platform (e.g. iOS, JavaME), creating specific containers depending on the structure of the target platform and providing the source code templates for the new target platform. Examples of this are the classes inside the *mobia.psm.android* package which contains Android-specific classes (see appendix A.4 for descriptions of the classes and packages).

New Components. In the current implementation of the *Mobia Framework*, extending the framework in order to accommodate new types of components can be done by manually adding the definition to the XML Schema Document that contains the model definition, and also adding code to the *Mobia Modeler* to show the component in the *Mobia Modeler*'s interface. A proposed approach on how to easily extend the capabilities of the framework will be discussed in the future work section 6.3.2.

- **Evolvability.** Since there is a clear separation between the *Mobia Modeler* and *Mobia Processor* implementations, it is possible to make use of new technologies or platforms for future implementations as long as it follows the same underlying model as illustrated in the *Mobia Proto-Go* tool example.
- **Composability.** The basic idea of having configurable components in the design of the model components in the *Mobia Framework* would satisfy the *composability* property if there was an easier way of adding new components without having to modify the underlying code. A proposed approach in order to do this is discussed in the future work in section 6.3.2.

4.4 Summary

In this chapter, a detailed view into the *Mobia Framework* and its different parts was presented. An overview about MDSD and related approaches was first presented to give background to the paradigm in which the *Mobia Framework* is based on.

The different use cases of the *Mobia Framework* in which roles of possible stakeholders (e.g. developers, interface designers, model experts, non-technical end user, etc.) were presented. It was emphasized that in order for the different stakeholders to work together with the different parts of the framework, a common artifact which in this case is the model [PVH07] should act as the center of all development efforts.

For the different parts of the *Mobia Framework* such as the *Mobia Modeler* and *Mobia Processor*, details about the functional requirements (e.g. expected users, inputs, outputs, etc.), the design and implementation were presented in sections 4.2.3 and 4.2.4 respectively. For the *Mobia Processor* in particular, the transformation process that a model undergoes in order to generate code was discussed.

Finally, an evaluation of the *Mobia Framework* against non-functional requirements such as usability, extensibility, composability, etc. was presented.

Chapter 5

The Mobia Models

This chapter will give a detailed look into the *Mobia Framework* models. This includes a discussion about the purpose and design decisions for the model components, the models' concrete syntax (i.e. graphical representation inside the *Mobia Modeler* and its XML form), the metamodel, and the semantics for code generation.

Contents

5.1 Model Discussion: An Overview

This chapter will present details about the models in the *Mobia Framework*. The scenario presented in section 4.2.2 will be used as an example to easily explain the models. The details about the models will be presented as follows (i.e. a guide for model discussion):

- Each discussion will start off with a *description (i.e. purpose)* of a model type and the *decisions influencing its design*.
- The *concrete syntax*[1] of a model will then be presented, both its graphical form which is visible in the *Mobia Modeler* and its XML form which is the one exported from the *Mobia Modeler*[2] to be processed later on by the *Mobia Processor*.
- The *metamodel* for the XML form of the component will then be discussed. Although the metamodel is expressed in XML Schema Document (XSD), the UML model of the metamodel will be shown for easier reference. A view of the overall *Mobia Metamodel* can be found in appendix B.
- Finally, the *semantics of the model* will then be presented particularly the mapping of the model elements to the generated code fragments. As proof of concept, code was generated for the *Android Framework*[3]. In order to understand the mapping between some of the model elements to the Android code, some background information on parts of an Android application relevant to the discussion will be given beforehand.

5.2 Application Requirements

In this section, the mobile application functionalities which are supported by the *Mobia Framework* will be presented. The purpose of this is to make it clear to potential users about what can be modeled and generated using the framework, and what the limitations are.

Supported Functionalities. The following mobile application functionalities can be modeled and generated using the *Mobia Framework*:

- Application functions that allow the user to *input information* such as typing in text, selecting from lists, activating functionalities by pushing buttons, etc. In the domain of *mobile health monitoring*, some examples of such functionalities are:

[1] The *concrete syntax* (e.g. may be in the form of textual or graphical constructs) is the realization of the abstract syntax , and is the interface to the modeler. The *"quality of the concrete syntax decides what degree of readability the models have"* [SVC06].

[2] This refers to the latest version of the *Mobia Modeler* with the configurable component-based design, which was designed and developed together with Max Tafelmayer [Taf09][BFTH10].

[3] http://developer.android.com/index.html

  - The input of personal health information such as weight, height, age, etc. which can be used by the application later on to compute for health status (e.g. body mass index which is based on the weight and height). Similar information related to health that can be logged are food intake, physical activities, blood sugar concentration (blood glucose levels), etc.
  - The input of user/patient information such as username and password in order to identify the user/patient using the application in order to load/save user-specific data.

- Application functions that allow processing of simple control logic. The application can process certain *inputs* and do specific *actions (device-specific functions such as calling, sending SMS, etc.)* based on specified *conditions* [Taf09][BFTH10]. In the domain of *mobile health monitoring*, examples of such are used for:
  - Informing the user when his health status is out of the normal range (e.g. blood pressure level is greater than 140) in order for the user to take the necessary actions (e.g. take medicine or go to the doctor).
  - Automatic (condition-triggered) actions done by the application such as calling a special number during an emergency when the patient's health status is on a dangerous level (e.g. epilepsy attack [Taf09][BHSW07]).

- Applications that interact with external devices (e.g. sensor) processing their data and providing visualization of such devices and their information in the modeling environment. In the domain of *mobile health monitoring*, examples of such devices that collect physiological data that can be used by the application are [BFH09c]:
  - A heart monitor such as the one from Alive Technologies [Ali] that collects such physiological information from the user and transmits this data remotely via bluetooth to a mobile device (or desktop).
  - The accelerometer such as the one Alive Technologies [Ali] to monitor body movement. This was used in applications such as MOPET from Buttussi et al. [BC08] and MPTrain from Oliver et al. [OFM06b].

  Take note however, that the implementation for communicating with a specific device and processing of the raw sensor data collected from these devices is not a primary concern for this research. The APIs that contain the implementation for such processing (e.g. MobHealth Framework [Moba][Mobb]) will be used by the *Mobia Framework* combined with the available code templates in order to make use of such functionalities.

Limitations. The following are not supported or cannot be done using the current version of the *Mobia Framework*:

- Customization of the user interface (i.e. add and layout of UI elements). However, a tool such as the *Mobia Proto-Go* can be used as a supplementary tool for modeling platform-specific and UI customizable mobile applications (section 3.2.6).

- Modeling and generating code for highly interactive applications with rich graphics such as multimedia games.

- Modeling application response to device-specific interaction methods such as *gestures*[4] [5] (e.g. tap, swipe, long press, etc.). Although, such support can be implemented in the code templates if necessary.

5.3 Structure Components: Adding Screen Instances and Application Flow

Structure components are used in order to add screen elements and application flow (i.e. transition from one screen to the next) to a model in the *Mobia Modeler*.

5.3.1 Design Decisions

In the previous *Mobia Modeler* [Örg09] prototypes, adding application flow to the model consisted of two steps: (1)adding a screen to the model, and then (2)dragging arrows from one screen to the next to signify the flow. There was also no way of adding conditions to the flow such that the transition is made only if the condition is satisfied.

For the current version of the *Mobia Modeler*, *structure components* are used to automatically create screens, application flow and add conditions for the transitions.

The basis for such a design is to simplify an otherwise multiple-step process of adding screens and application flow as the previous prototypes. This is because when a *structure component* is added to the model, it automatically creates another screen. For the code generation part (i.e. *Mobia Processor*), validation of the conditions specified in the *structure component* is also easier since the model will have predefined conditions specific to a certain type of *structure component*.

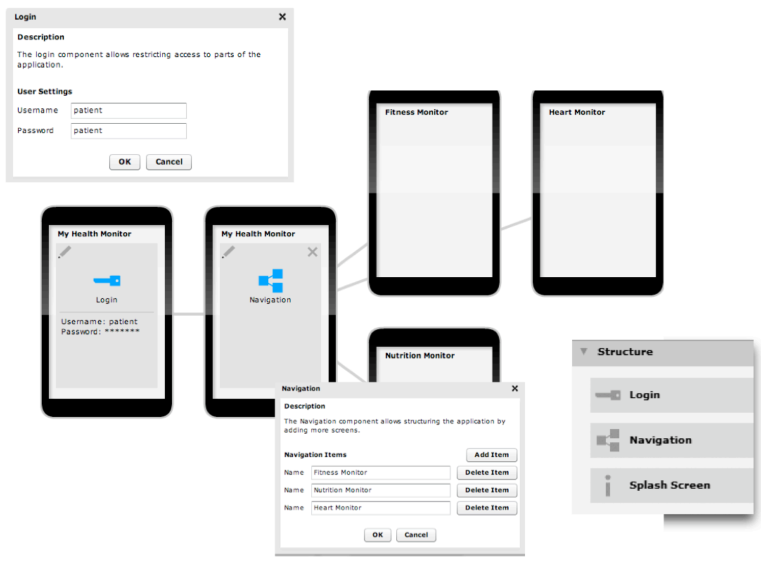
5.3.2 Concrete Syntax

In the example *health monitor application*, an initial screen which allows the patient to log in to the application is needed. To model this in the *Mobia Modeler*, a *login component*

[4] http://developer.android.com/resources/articles/gestures.html

[5] http://developer.apple.com/iphone/library/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/GestureRecognizers/GestureRecognizers.html

is added to the model. Adding a *login component* to the model automatically creates a second screen. The component can then be configured to accept a specific username and password. For this version, we only allow one login information to be stored. However, this can be extended later on to allow storing of information for multiple users. Figure 5.1a shows the *login component* inside the *Mobia Modeler* and its configuration dialog. Figure 5.1b shows the equivalent XML form when exported from the modeler.



The login component allows restricting access to parts of the application.

Username: patient
Password: patient

Username: patient
Password: *******

The Navigation component allows structuring the application by adding more screens.

Name: Fitness Monitor
Name: Nutrition Monitor
Name: Heart Monitor

(a)

```
<component id="1" xsi:type="login" screenRef="1">
   <data>
     <item>
       <username>patient</username>
       <password>patient</password>
       <targetScreen>2</targetScreen>
     </item>
   </data>
</component>
<component id="2" xsi:type="navigation" screenRef="2">
   <data>
      <item>
        <screenTitle>Fitness Monitor</screenTitle>
        <targetScreen>3</targetScreen>
      </item>
      <item>
        <screenTitle>Nutrition Monitor</screenTitle>
        <targetScreen>4</targetScreen>
      </item>
      <item>
        <screenTitle>Heart Monitor</screenTitle>
        <targetScreen>5</targetScreen>
      </item>
    </data>
</component>
```

(b)

Figure 5.1: (a) An example of the *structure components* in action based on the Health Monitor application. (b) The equivalent XML form of the *structure components*.

Another example of a *structure component* is the *navigation component*. Unlike the *login component* which only adds one screen to the model, the *navigation screen* allows multiple screens to be added to the model. In the *health monitor application* for example, we need a *navigation component* in order to allow other components to be added to our model. This acts like a menu screen where the user can choose which application (e.g. fitness monitor, nutrition monitor, heart monitor) he wants to run. Figure 5.1a shows the *navigation component* inside the *Mobia Modeler* and its configuration dialog where one can add multiple instances of the screen. Figure 5.1b shows the equivalent XML form when exported from the modeler.

5.3.3 The Metamodel

Structure components inherit attributes from the *abstractComponent* element which are: the component *id* and a reference to the screen (i.e. *screenRef*) containing them. All *structure components* are made up of one element called the *data*. The *data* element contains one or more *item* elements (see figure 5.2a).

Different types of *structure components* can have different types of *ItemType* elements as shown in figure 5.2b. For example, for the *login component*, the items that it can contain are the items *username* and *password*. However, one *ItemType* that is common to all *structure components* is the *targetScreen* element. This stores the *screenRef (i.e. screen id)* of the screen that will be displayed if a certain condition in the *structure component* is satisfied. For example, for the *login component*, a successful login (i.e. user name and password is in the user database) will lead to displaying the screen with the particular *screen id*.

5.3.4 Mapping Model Data to Code

An Android *Activity*. An activity in Android represents a visual interface to the user and is implemented as a subclass to the *Activity* [6] base class. An application may consist of multiple activities. For example, for an email application, one *Activity* might be used to display the list of emails received (i.e. Inbox), another *Activity* might be used for composing an email, and so on.

Android Activities and the *Mobia Framework*. In the *Mobia Framework*, a model component is represented by one Android *Activity* (i.e. one Android class). This implies that since a model component is an *Activity*, it always has an equivalent user interface in the application.

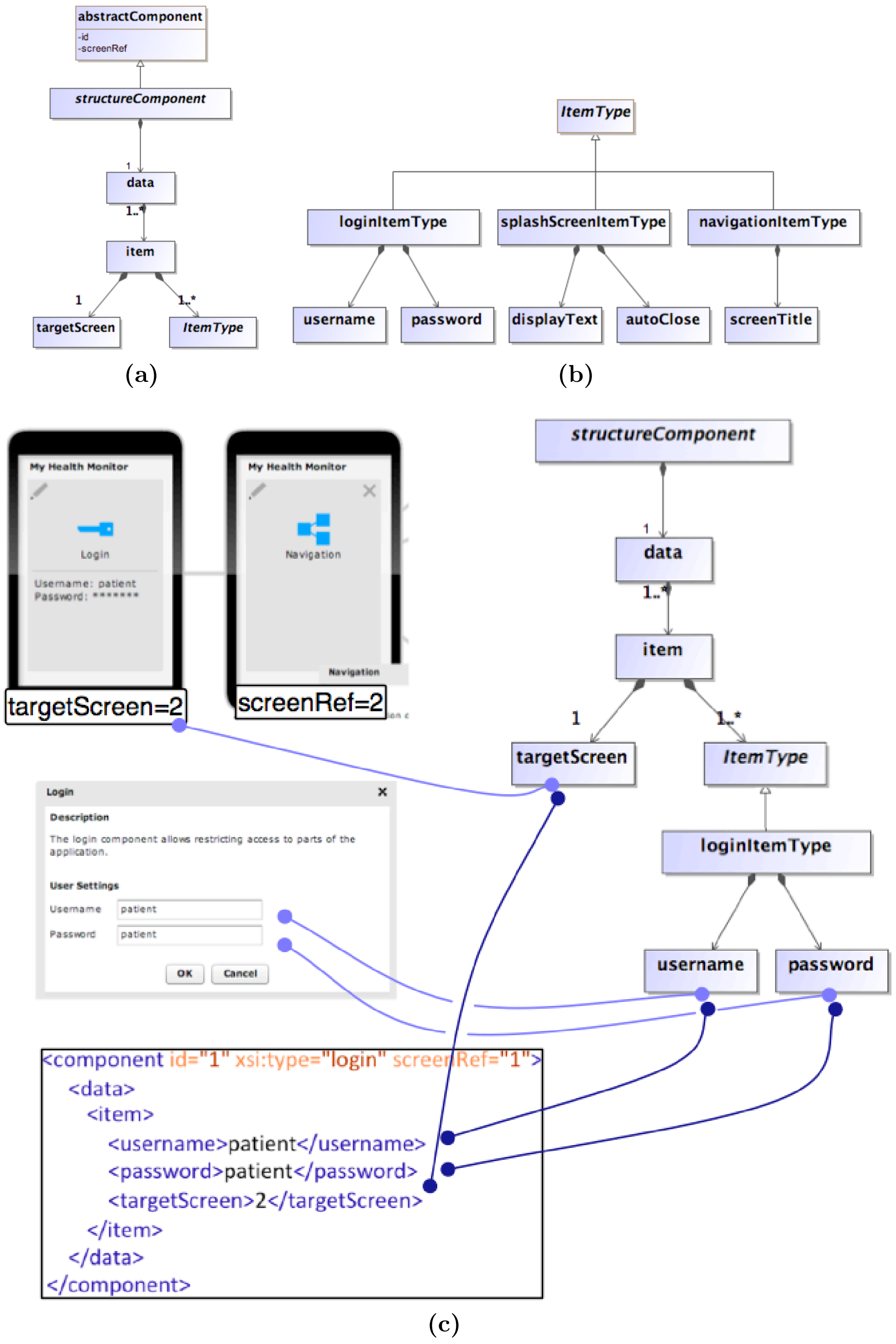
In order to separate the control logic from the user interface definition, two files that represent an activity are needed: the Java class file where the main control logic and processing is stored, and the XML layout file[7] [8].

The class name of each activity in the application is derived from the application's name excluding the white spaces in order to form a valid identifier, and a unique number. For example, if the name of the application is *My Health App* and the application contains five classes, the classes will be named *MyHealthApp1, MyHealthApp2,* etc. The layout name is derived from the type of component such as *login* or *navigation* including a unique number to identify multiple instances of the same component type in the model. For example, if there are two components of type *navigation* in the model, the layout files will be named *navigation1.xml* and *navigation2.xml* respectively.

[6] http://developer.android.com/reference/android/app/Activity.html

[7] http://developer.android.com/guide/topics/ui/index.html#Layout

[8] It must be noted, that it is possible to instantiate the layout of the user interface during runtime (http://developer.android.com/guide/topics/ui/declaring-layout.html). However in order to provide more flexibility to the layout (e.g. modify the layout or look-and-feel of the component), the use of the XML layout file is adapted to specify the layout.



Login

Description

The login component allows restricting access to parts of the application.

User Settings

Username	patient
Password	patient

```
<component id="1" xsi:type="login" screenRef="1">
  <data>
    <item>
      <username>patient</username>
      <password>patient</password>
      <targetScreen>2</targetScreen>
    </item>
  </data>
</component>
```

Figure 5.2: (a) The metamodel for structure components. (b) The possible item types for the different structure components. (c) An example showing the relationship between concrete syntax and metamodel for the *login component*. The value of *targetScreen* is internally assigned in the *Mobia Modeler*.

Invoking Android Activities via Intents. An Android activity is started from the currently running activity by calling the *startActivity(Intent)*[9] method with an instance of an *Intent*[10] that describes the activity to be executed (e.g. contains information about the name of the Activity). Listing 5.1 shows how to start a new activity.

Listing 5.1: Starting an *Activity* with the use of Intents.

```
        Intent myIntent = new Intent();
        myIntent.setClassName(targetActivityPackage,targetActivity);
        startActivity(myIntent); //method from the Activity base class
```

Modeling Activity Invocation via *Structure components*. The purpose of *structure components* as already mentioned is that it allows the user to model simple application flow. In the context of an Android application, this simply means that it allows invoking another activity from the current activity.

To give a concrete example, given the following snippet from a code template for a *structure component* in listing 5.2. The name of activity (i.e. class name) including the package where it belongs to are declared as variables in the code template[11]. The code in listing 5.1 is then used in order to start a specific activity from the current activity.

Listing 5.2: A snippet of the code template expressed in Velocity Template Language (VTL). This shows the template declaring the target activity information such as the name and package of the target activity. Take note that this target activity information is generated by the *Mobia Processor* during the processing of the *Mobia PIM* to *Mobia PSM* (section 4.2.4)

```
        //Target Activity
        private String targetActivityPackage = "$packagename";
        private String targetActivity = "$packagename"  + "$activity.get("targetScreen")";
```

Depending on the type of *structure component* there is, invocation of the next activity depends if the conditions for a certain component type are satisfied. More on this will be discussed next.

Component-Specific Data. As mentioned, different *structure components* have different information and has its equivalent variable in the code templates. For example, for the *login component*, the most essential code snippet for the component that stores user information to the database is shown in listing 5.3.

Listing 5.3: A snippet of the code template that stores login information to the database.

```
        private String  loginNameData = "$activity.get("username")";
        private String  loginPwdData    = "$activity.get("password")";
        ...
```

[9] http://developer.android.com/reference/android/app/Activity.html#StartingActivities

[10] http://developer.android.com/reference/android/content/Intent.html

[11] The identifiers beginning with a dollar ($) sign symbolizes a variable in the Velocity Template Language (VTL)(http://velocity.apache.org/engine/releases/velocity-1.6.4/user-guide.html) which is what is used by the *Apache Velocity Engine*.

```
        //insert login data to the database
        AppDB   db = new AppDB(this);
        db.insertToAccountsTable(loginNameData, loginPwdData);
```

Listing 5.4 shows the code for verifying the identity of the user. If this returns a *true* value, this will result to an invocation of the next activity (i.e. call a new Activity instance) as shown in listing 5.1. Take note the the only information that we need from the model in the given example are the *userame* and *password* extracted from the *login component*.

Listing 5.4: A snippet of the code template for verifying information stored in the database as shown in listing 5.3. The database is searched in order to verify if the username and password pair is in the database.

```
        //find if user and password combination exists in database
        for( int i=0; i<resultsList.size(); i++ ){
                AppDB.AccountsTableRow row = resultsList.get(i);
                if( username.equalsIgnoreCase(row.username) ){
                        if( password.equalsIgnoreCase(row.password)){
                                loginResult = true;
                                break;
                        }
                }
        }
```

5.4 Basic Components: Adding Default Applications

Basic components are used in order to represent applications or services that are already available in a certain mobile device (e.g. notes application, camera, voice recorder, etc.). Adding a *basic component* to the model in the *Mobia Modeler* signifies invoking these applications or services from the modeled application.

5.4.1 Design Decisions

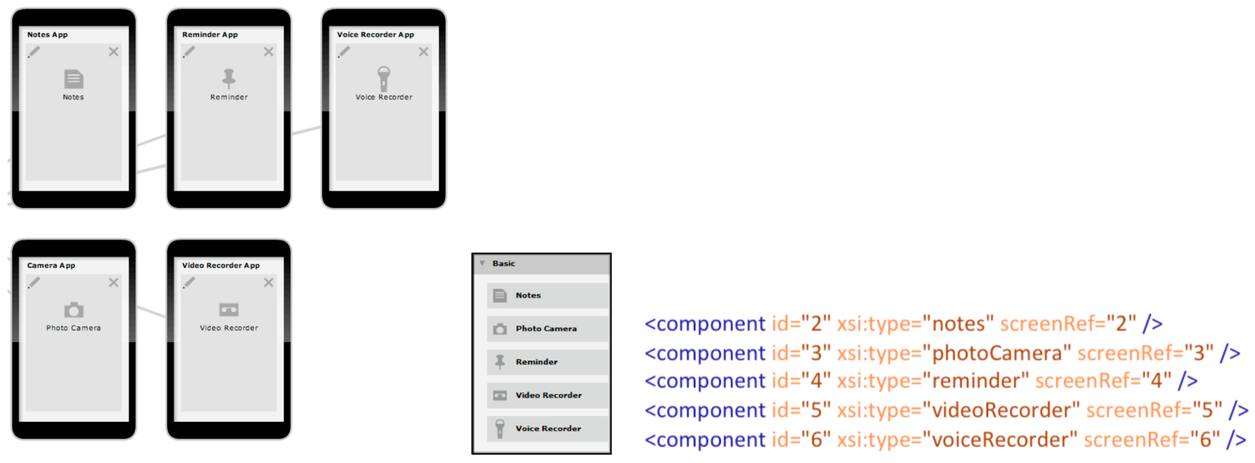
With the advent of modern mobile frameworks such as *Android*, launching other applications (provided they are available or supported by the platform) within a running application is possible. This is in order to allow developers to make use of such applications that already available in the current platform instead of reinventing the wheel by doing additional programming.

For the current version of the *Mobia Modeler*, such functionality can be modeled by adding *basic components* to the model.

5.4.2 Concrete Syntax

Since in the featured example in section 4.2.2, there are no *basic components* needed in the application, we make a new model just to show the concrete syntax of *basic components*.

Figure 5.3a shows the different *basic components* inside the Mobia Modeler, while figure 5.3b shows the equivalent XML form when exported from the modeler.



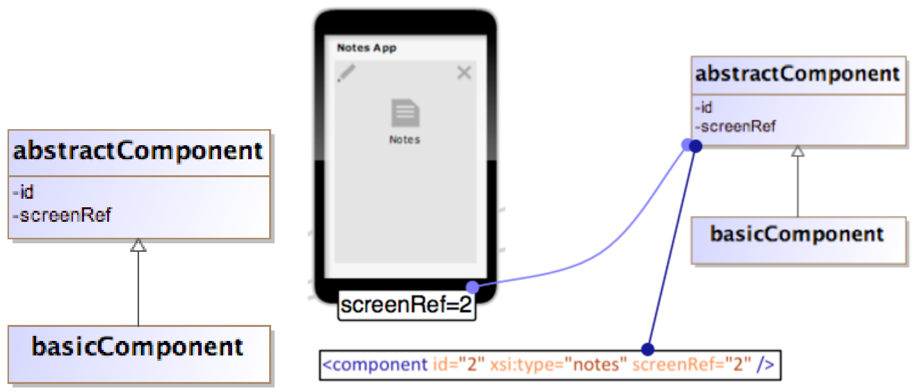
```
<component id="2" xsi:type="notes" screenRef="2" />
<component id="3" xsi:type="photoCamera" screenRef="3" />
<component id="4" xsi:type="reminder" screenRef="4" />
<component id="5" xsi:type="videoRecorder" screenRef="5" />
<component id="6" xsi:type="voiceRecorder" screenRef="6" />
```

(a) (b)

Figure 5.3: (a) An example of the *basic components* in the *Mobia Modeler*. (b) The equivalent XML form of the *basic components*.

5.4.3 The Metamodel

All *basic components* contain only information inherited from the *abstractComponent* element which are: the component *id* and a reference to the screen (i.e. *screenRef*) containing them.



```
<component id="2" xsi:type="notes" screenRef="2" />
```

(a) (b)

Figure 5.4: (a) The metamodel for *basic components*. (b) An example showing the relationship between concrete syntax and metamodel for the *notes component*.

5.4.4 Mapping Model Data to Code

Invoking Android Applications (i.e. Activities). In section 5.3.4, we already discussed how an activity can be invoked by a currently running activity with the use of *Intents* [12]. In principle, an application in Android is made up of activities, and therefore, invoking an application is just like invoking an activity. One just has to know which parameters should be set inside the *Intent* object in order to invoke that certain application. An example of invoking the camera is shown in listing 5.5.

Listing 5.5: Starting up the camera/video recorder.

```
        // use MediaStore.ACTION_VIDEO_CAPTURE for video
        Intent myIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        callingActivity.startActivity(myIntent);
```

However, other applications may need additional initializations rather than just using the *startActivity(Intent)* method. One example is invoking the *voice recorder* (i.e. audio recorder). A code snippet can be found in listing 5.6 (source code taken from the MediaRecorder documentation[13]).

Listing 5.6: Source code to record audio files.

```
        MediaRecorder recorder = new MediaRecorder();
        recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
        recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
        recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
        recorder.setOutputFile(PATH_NAME);
        recorder.prepare();
        recorder.start();   // Recording is now started
        ...
        recorder.stop();
        recorder.reset();   // You can reuse the object by going back to
                                // setAudioSource() step
        recorder.release(); // Now the object cannot be reused
```

Modeling Android Application (i.e. Activities) Invocation via *Basic components.* In the *Mobia Framework*, since the definitions of these *basic components* are similar no matter what type of model there is, one possible implementation would be to create a class (e.g. AndroidBasicComponents class in listing 5.7) which has methods that contain the implementation for invoking the different types of *basic components* available. During code generation, depending on the *basic component* added in the model, the corresponding method calls are added to the source code.

Listing 5.7: A class containing all implementations for invoking the different types of *basic components.*

```
        public class AndroidBasicComponents{
                public void launchCamera(Activity callingActivity){...}
                public void launchVideoCam(Activity callingActivity){...}
                public void launchAudioRecorder(Activity callingActivity){...}
```

[12] http://developer.android.com/guide/topics/intents/intents-filters.html
[13] http://developer.android.com/reference/android/media/MediaRecorder.html

```
                ....
        }
```

5.5 Special Components: Adding Domain-Specific Applications

Special components are used in order to represent applications or services that are installed on a mobile device and are specific to a certain type of domain. In the domain of *mobile health monitoring* for instance, examples of *special* components are the *fitness diary* and *nutrition diary* components which can be used by patients to monitor different aspects of their health. Other examples are functions such as *call-for-help* which calls a special number (e.g. 112 or 911) to call emergency services. Adding a *special component* to the model in the *Mobia Modeler* signifies invoking these applications or services on the mobile device.

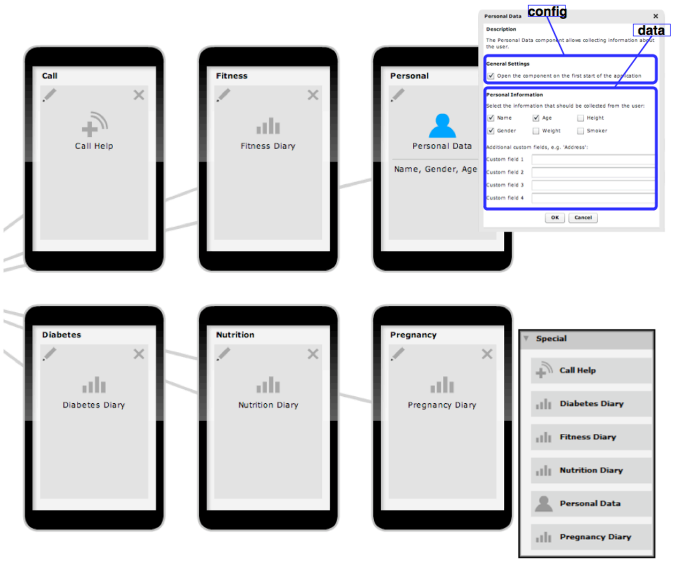
5.5.1 Design Decisions

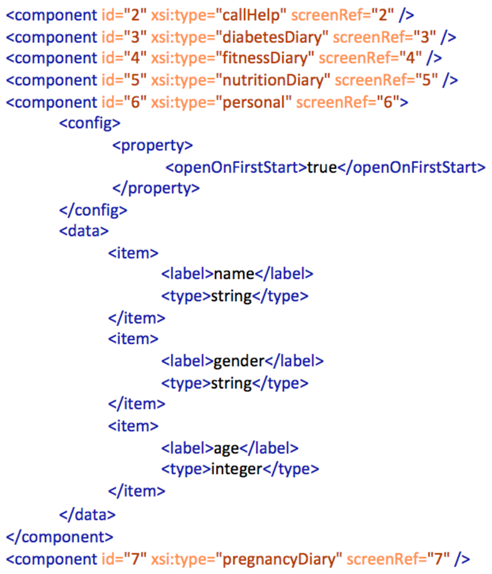
The idea behind the design of *special components* is actually similar to that of *basic components*. This is order to represent invocation of applications or services that are supported by a certain mobile platform. However, as already mentioned, *special components* are more specific to a certain type of domain.

5.5.2 Concrete Syntax

In the example in section 4.2.2, we need diary-type applications that would allow the patient to monitor his daily food intake and fitness activities. To model this in the *Mobia Modeler*, two types of *special components* can be used which are the *Nutrition diary* and the *Fitness diary*. In order to show the other types of *special components* in the *Mobia Modeler*, we created an alternative model containing all the *special components* as shown in figure 5.5a. The XML form of the models are shown in figure 5.5b.

To show that *special components* can also be configured, the *Personal Data* component together with its configuration is shown (5.5a). This *special component* simply collects information about the user of the application. The fields that would appear (e.g. name, age) in the application are marked in the configuration dialog. The setting *"Open the component on the first start of the application"* means that, this *special component* would be the first screen to appear when the application is used for the first time. The idea for this is because, typically, such personal information is only inputted once by the user. Modifications to the information can still be done later on if the user wants to change it.





```xml
<component id="2" xsi:type="callHelp" screenRef="2" />
<component id="3" xsi:type="diabetesDiary" screenRef="3" />
<component id="4" xsi:type="fitnessDiary" screenRef="4" />
<component id="5" xsi:type="nutritionDiary" screenRef="5" />
<component id="6" xsi:type="personal" screenRef="6">
        <config>
                <property>
                        <openOnFirstStart>true</openOnFirstStart>
                </property>
        </config>
        <data>
                <item>
                        <label>name</label>
                        <type>string</type>
                </item>
                <item>
                        <label>gender</label>
                        <type>string</type>
                </item>
                <item>
                        <label>age</label>
                        <type>integer</type>
                </item>
        </data>
</component>
<component id="7" xsi:type="pregnancyDiary" screenRef="7" />
```

(a) (b)

Figure 5.5: (a) An example of the *special components* in the *Mobia Modeler*. (b) The equivalent XML form of the *special components*. Take note that currently, not all *special components* are implemented in the *Mobia Modeler* (i.e. only the *personal* component exports with some configuration information).

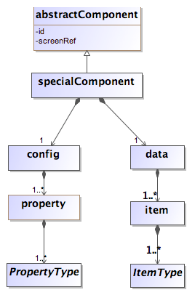
5.5.3 The Metamodel

Special components inherit attributes from the *abstractComponent* element which are: the component *id* and a reference to the screen (i.e. *screenRef*) containing them. All *special components* are made up of two elements called the *config* and *data*.

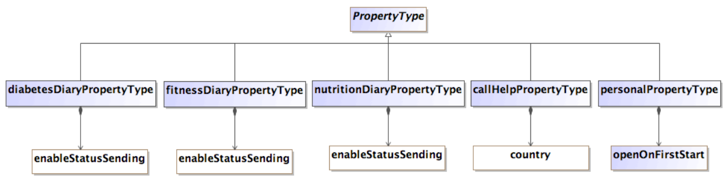
Figure 5.6 shows the metamodel for the *special components*. The unshaded boxes (white boxes) mean that they are still currently unimplemented in the *Mobia Framework* and are just shown for example purposes.

The *config* element can contain one or more *property* elements which differs in terms of the type of *special component* there is. For example, for the diary-type *special components* shown in figure 5.6b, the property *enableStatusSending* means that diary data can be sent. For the *callHelp special component*, the *country* property is needed since different countries or areas have different emergency numbers (e.g. the emergency number for EU countries is 112).

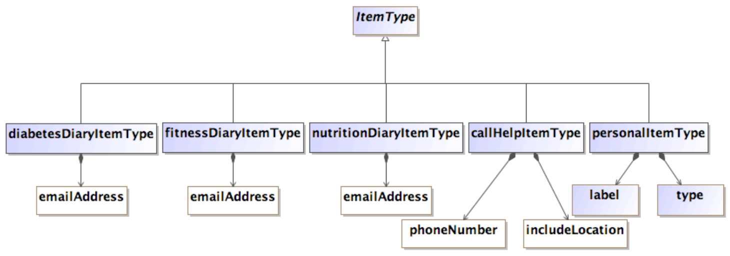
The *data* element contains one or more *item* elements (see figure 5.6a). The *ItemType* is basically related to the *PropertyType* in the *config* element. For example (see figure 5.6c), if the *enableStatusSending* is present in the model, then the *emailAddress* should contain the information on where the status will be sent.



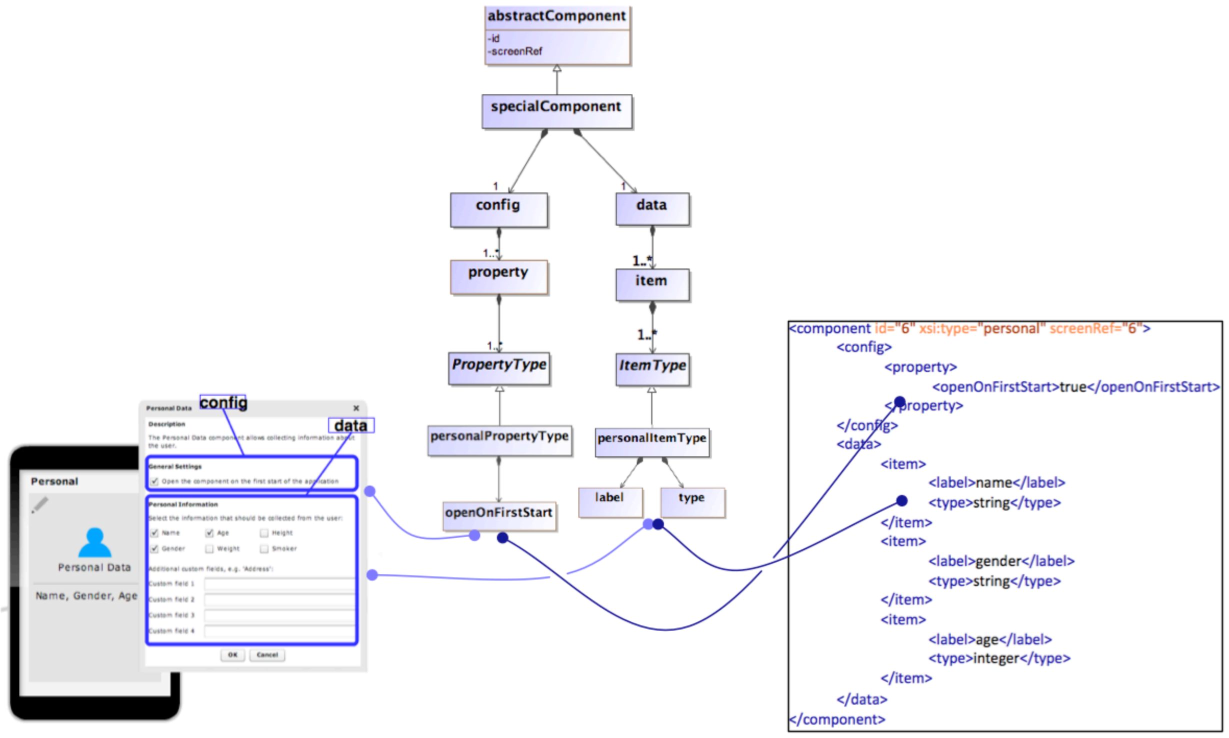
(a)



(b)



(c)



Personal Data

Description

The Personal Data component allows collecting information about the user.

General Settings

☑ Open the component on the first start of the application

Personal Information

Select the information that should be collected from the user:

☑ Name ☑ Age ☐ Height

☑ Gender ☐ Weight ☐ Smoker

Additional custom fields, e.g. 'Address':

Custom field 1

Custom field 2

Custom field 3

Custom field 4

```
<component id="6" xsi:type="personal" screenRef="6">
        <config>
                <property>
                        <openOnFirstStart>true</openOnFirstStart>
                </property>
        </config>
        <data>
                <item>
                        <label>name</label>
                        <type>string</type>
                </item>
                <item>
                        <label>gender</label>
                        <type>string</type>
                </item>
                <item>
                        <label>age</label>
                        <type>integer</type>
                </item>
        </data>
</component>
```

(d)

Figure 5.6: (a) The metamodel for *special components*. (b) The possible property types for the different *special components*. (c) The possible item types for the different *special components*. (d) An example showing the relationship between concrete syntax and metamodel for the *personal component*.

5.5.4 Mapping Model Data to Code

Modeling Domain-Specific Android Application (i.e. Activities) Invocation via *Special components*. The same principle as generating code for the *basic components* (section 5.4.4) is applied for *special components*. It is just important to know the name of the class (i.e. Activity) that has the implementation for the *special component* and the package it belongs to in order to invoke it via the *startActivity(Intent)* method.

The initializations for the *special component* class are done inside the *onCreate()* method. The initialization information are taken from the model elements *config* and *data*. An example code template for the *Personal* component is shown in listing 5.8.

Listing 5.8: An example code template for the *Personal special component* class.

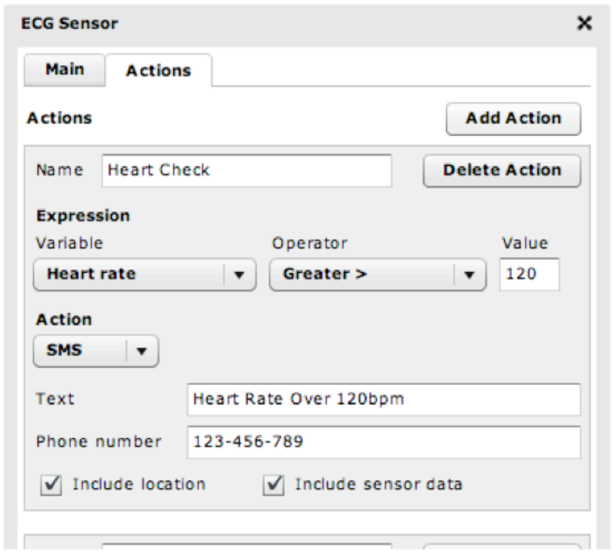
```
    public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.personalinfo);

        //do the necessary initializations: CONFIG info
        #foreach( $config in $configList )
                #if( $config.getConfigVariableValue() == "openOnFirstStart")
        this.setOpenOnFirstStart( true );
                ...
                #end
        #end

        //do the necessary initializations: DATA info
        #foreach( $field in $DATAVARMAP )
                 #set( $result = false )
                 #set( $result = $field.get("label") )
                 #if( $result )
                        createInputField("$field.get("label")");
         #end
        #end

        ....
        }

        public void createInputField(String fieldName){
        ....
```

5.6 Sensor Components: Adding Complex Application Logic

Sensor components are used in order to represent real-world sensors that collect various information and transmit this information to the mobile device. *Complex application logic* (see section 3.1.4 for complete description) can also be added to the model through the use of this type of component.

5.6.1 Design Decisions

In the *Mobia Modeler*, external devices are represented as one type of *sensor component* in the model (e.g. ECG Sensor Component). Complex application logic can be added by configuring the *Actions* of the *sensor component*.



Name: Heart Check
Variable: Heart rate
Operator: Greater >
Value: 120
Action: SMS
Text: Heart Rate Over 120bpm
Phone number: 123-456-789
✓ Include location
✓ Include sensor data

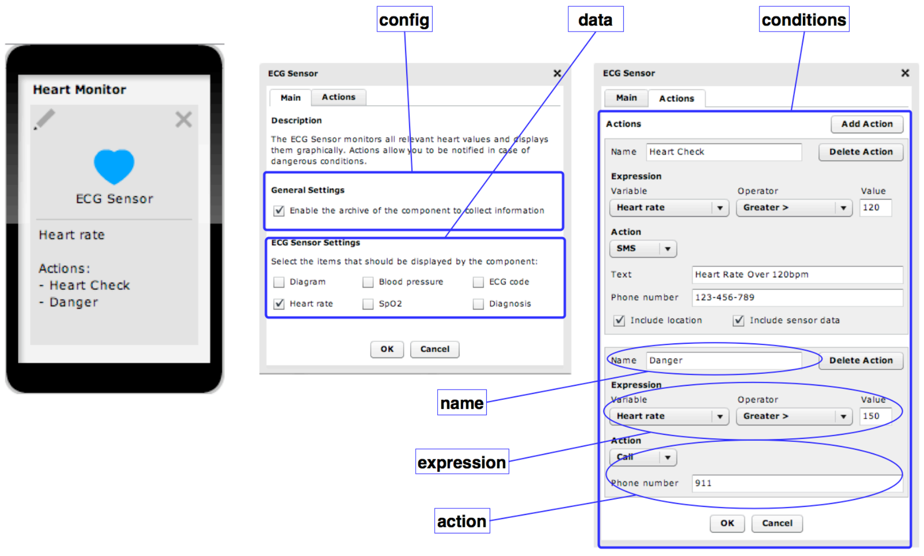
Figure 5.7: Example configuration for a *sensor component*.

Please refer to figure 5.7 for the following explanation. The information from device is simply treated as an *input variable* in the configuration dialog of the model component (e.g. *Heart rate* for the ECG Sensor Component). Relational operators (e.g. $= > <$) are used in order to compare the values of such variables to another input value provided by the user (e.g. value which says 120).

In order to model the response of the mobile application to the conditions stated, such response are represented as *actions* in the model. Actions are basically functions that the mobile device is capable of such as calling, sending SMS, vibrating, etc. This is modeled by simply selecting the *action* in the configuration dialog shown in figure 5.7 and adding the necessary information for that action (e.g. supply phone number and message for the *SMS* action).

5.6.2 Concrete Syntax

In the example in section 4.2.2, we need some way to monitor the heart rate and do the necessary actions depending on the value of the patient's heart rate. In the *Mobia Modeler*, we model this by using the *ECG Sensor* component. Figure 5.8a shows the *ECG Sensor* and the configuration information added in order to satisfy the conditions stated in the example application. The equivalent XML form of the model is shown in figure 5.8b.



Heart Monitor

ECG Sensor

Heart rate

Actions:
- Heart Check
- Danger

ECG Sensor

Description

The ECG Sensor monitors all relevant heart values and displays them graphically. Actions allow you to be notified in case of dangerous conditions.

General Settings

✓ Enable the archive of the component to collect information

ECG Sensor Settings

Select the items that should be displayed by the component:

☐ Diagram	☐ Blood pressure	☐ ECG code
✓ Heart rate	☐ SpO2	☐ Diagnosis

ECG Sensor

Actions

Name: Heart Check

Expression

Variable	Operator	Value
Heart rate	Greater >	120

Action: SMS

Text: Heart Rate Over 120bpm

Phone number: 123-456-789

✓ Include location ✓ Include sensor data

Name: Danger

Expression

Variable	Operator	Value
Heart rate	Greater >	150

Action: Call

Phone number: 911

(a)

```
<component id="4" xsi:type="ecgMedget" screenRef="5">

   <config>
     <property>
       <enableArchive>true</enableArchive>
     </property>
   </config>


   <data>
     <item>
       <status>heartRate</status>
     </item>
   </data>

<conditions>
  <condition>
    <name>Heart Check</name>
    <expressions>
      <expression>
        <variable>heartRate</variable>
        <operator>greater</operator>
        <value>120</value>
      </expression>
    </expressions>
    <actions>
      <action xsi:type="sms">
        <property>
          <text>Heart Rate Over 120bpm</text>
          <phoneNumber>123-456-789</phoneNumber>
          <includeLocation>true</includeLocation>
          <includeSensorData>true</includeSensorData>
        </property>
      </action>
    </actions>
  </condition>

  <condition>
    <name>Danger</name>
    <expressions>
      <expression>
        <variable>heartRate</variable>
        <operator>greater</operator>
        <value>150</value>
      </expression>
    </expressions>
    <actions>
      <action xsi:type="call">
        <property>
          <phoneNumber>911</phoneNumber>
        </property>
      </action>
    </actions>
  </condition>
</conditions>

</component>
```

(b)

Figure 5.8: (a) An example of a *sensor components* in the *Mobia Modeler* configured to satisfy the conditions in the example *health monitor application*. (b) The equivalent XML form of the *sensor component*.

5.6.3 The Metamodel

Sensor components inherit attributes from the *abstractComponent* element which are: the component *id* and a reference to the screen (i.e. *screenRef*) containing them. All *sensor components* are made up of three elements called the *config*, *data* and *conditions*. Figure 5.9a shows the metamodel for the *special components*.

The *config* element can contain one or more *property* elements which differs in terms of the type of *sensor component* there is. An example *propertyType* (figure 5.9b) for the *ECG Sensor special component* would be the *enableArchive* property which means all information collected by the certain sensor will be stored. For the *Activity Medget special component*, the property *runOnStart* means that the sensor should immediately start collecting data once the application is launched.

The *data* element contains one or more *item* elements. The different *sensor components* basically store the same *ItemType* (see figure 5.9c) which is the *status*. This stores which type of status the sensor should collect (i.e. what is displayed in the application). For example, for the *ECG Sensor* component, the user could choose in the configuration dialog which status information should be collected such as heart rate, blood pressure, etc. (see figure 5.8a).

The *conditions* element can contain zero or more *condition* elements (figure 5.9a). Each *condition* element contains three elements which are the *name*, *expressions* and *actions*.

The *name* contains a string that represents an identifier to the condition.

The *expressions* element can contain one *expression* element that contains a variable, operator and value that needs to be evaluated. As of the moment, the *Mobia Framework* only supports one expression. In the future though, it is possible to extend it such that multiple expressions together with logical operators (e.g. AND, OR) can be evaluated.

The *actions* element can contain one *action* element that contains information about the actions the application will carry out when the expression evaluates to *true*. As of the moment, there is a one-to-one mapping between an expression and an action. This means that only one type of action can be carried out for each expression (e.g. if(heartRate > 100) -> action(call 911)).

Depending on the *ActionType* there is, different *PropertyType*s are available for each action (see figures 5.10b and 5.10c). For the *email* action type for example (figure 5.10c), the available *PropertyType*s are:

- The *text* which contains the text of the email.
- The *includeSensorData* which when selected will attach the collected sensor data to the email.
- The *includeLocation* which when selected and the device is capable of getting the location information of the user would also include this in the email.

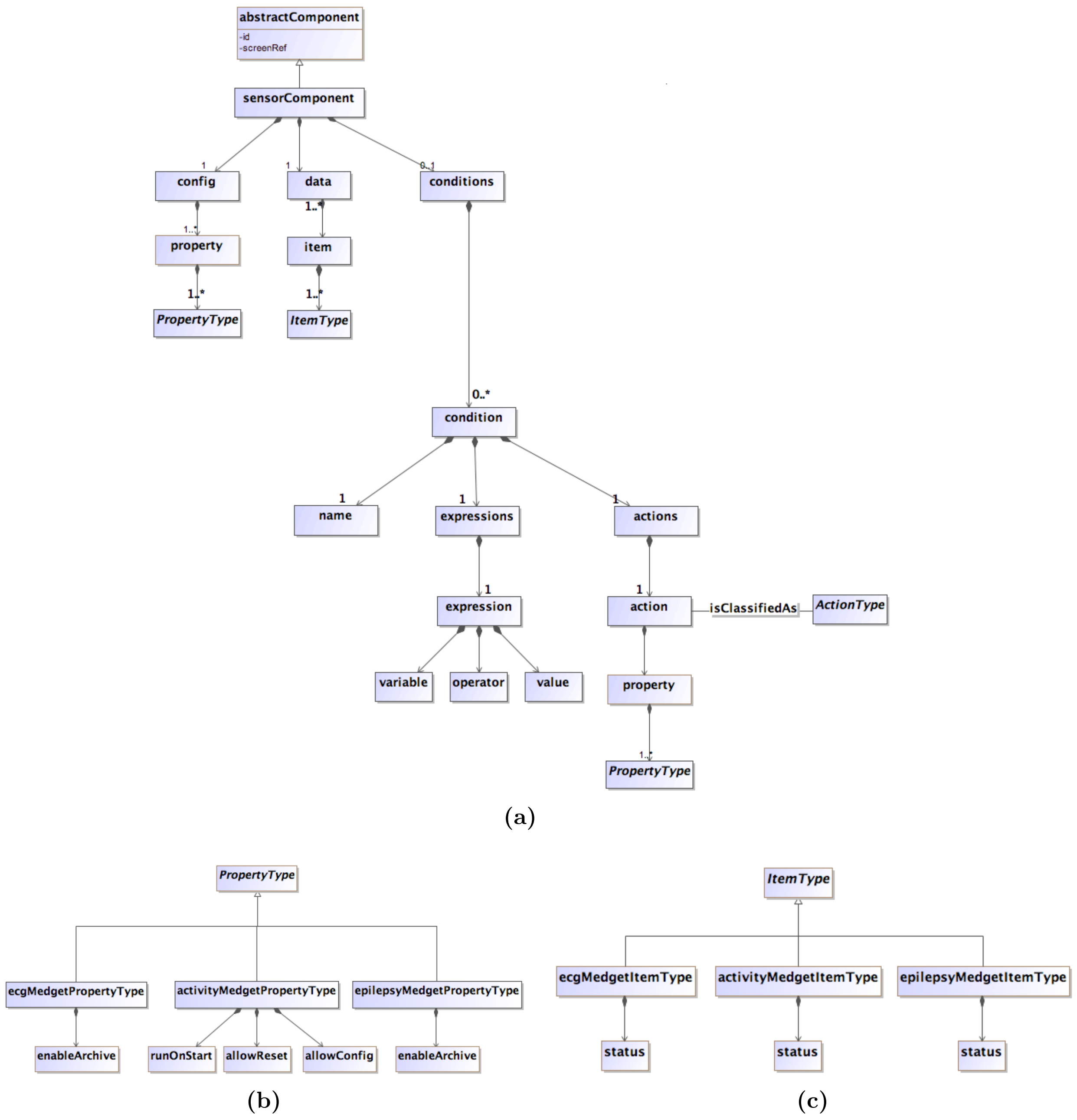


Figure 5.9: (a) The metamodel for *sensor components.* (b) The possible property types for the different *sensor components.* (c) The possible item types for the different *sensor components.*

- The *emailAddress* which contains the address of the recipient.

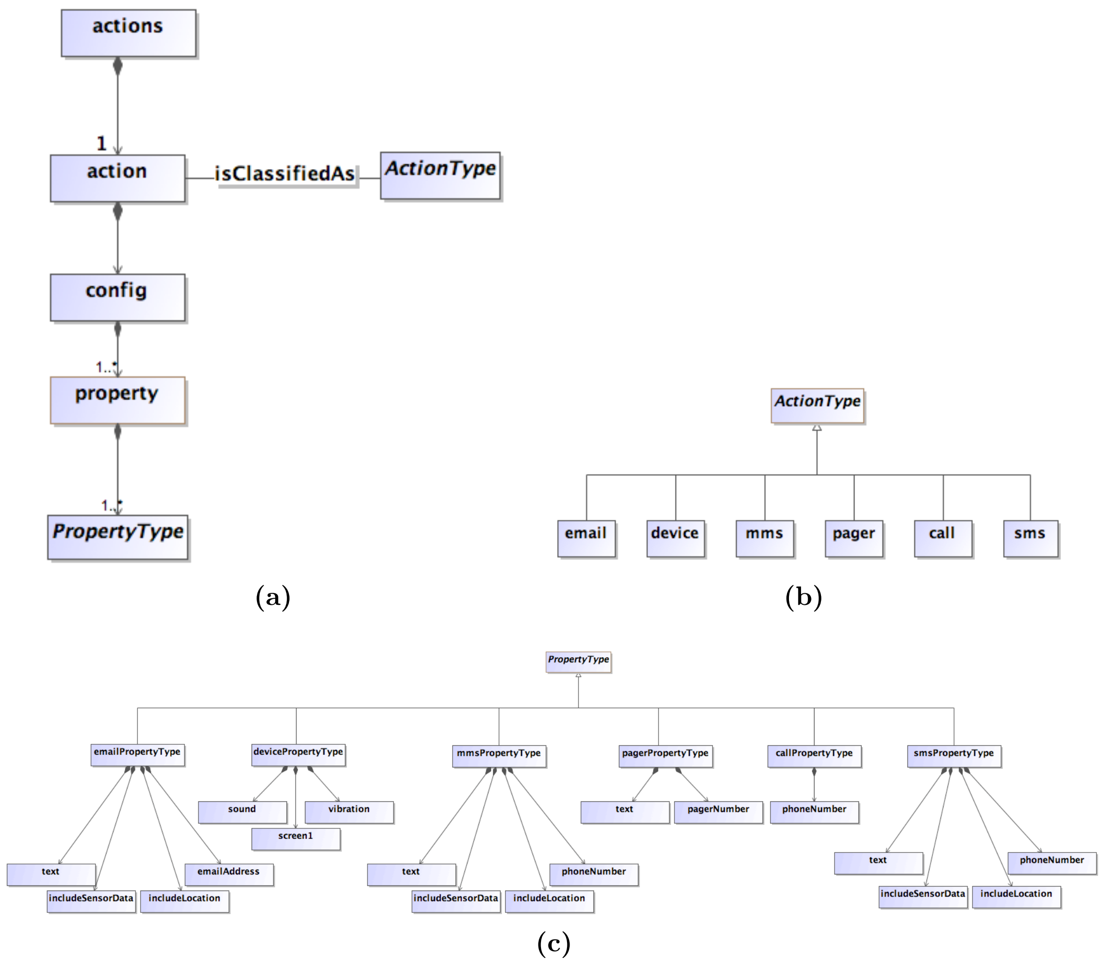
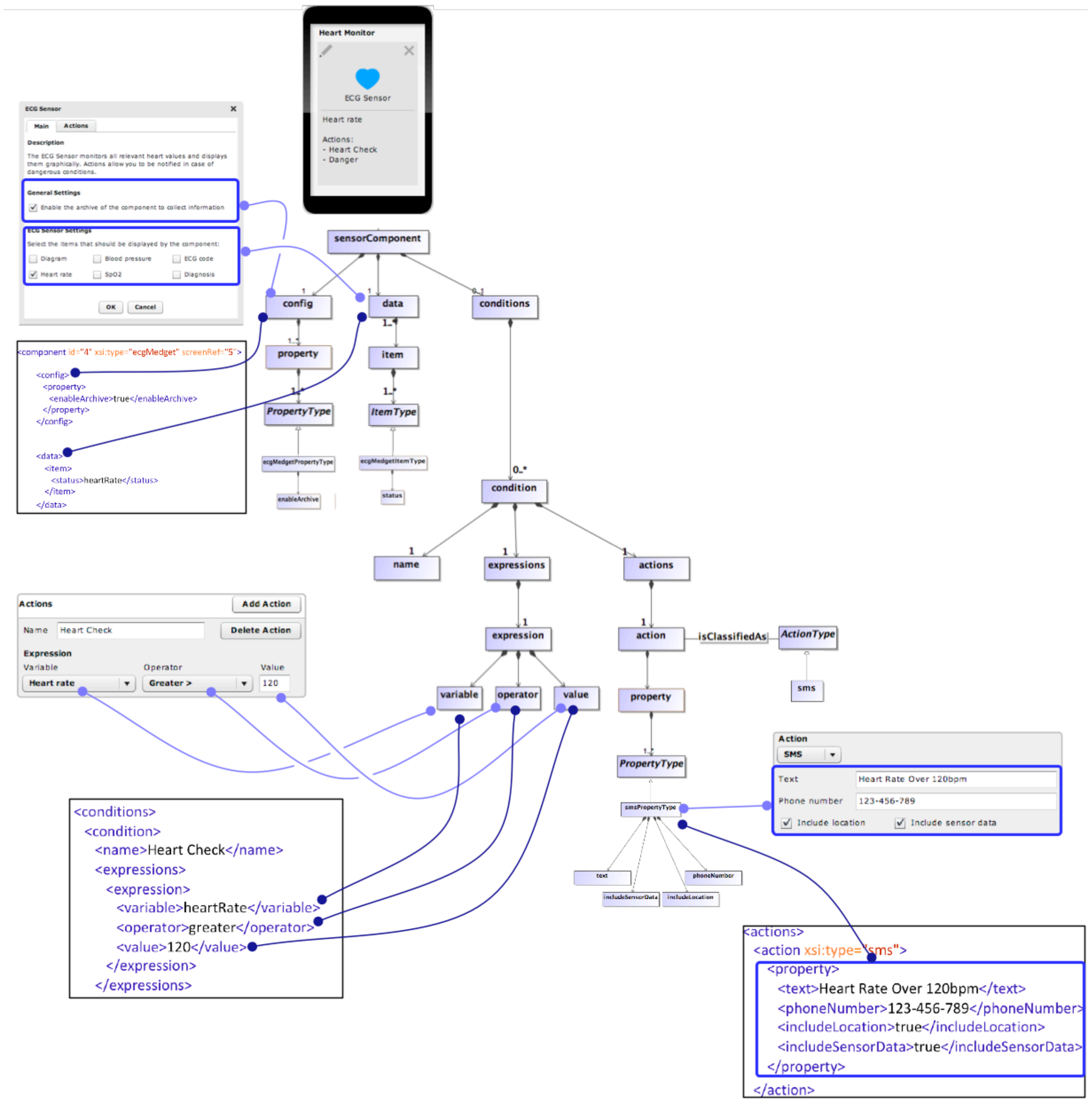


Figure 5.10: (a) The metamodel for *actions*. (b) The possible *action* types. (c) The possible property types for the different *actions*.

5.6.4 Mapping Model Data to Code

Just like *special components*, *sensor components* contain *config* and *data* elements which contain information about the component. How these information are mapped to code were already discussed in section 5.5.4. For this section, we will describe how *condition* and *action* elements are mapped to code.

Modeling Expressions with *special components*. Given the expression in the example shown in the model in figure 5.8a, we have the *Danger* expression which is declared as:



ECG Sensor

Main | Actions

Description

The ECG Sensor monitors all relevant heart values and displays them graphically. Actions allow you to be notified in case of dangerous conditions.

General Settings

✓ Enable the archive of the component to collect information

ECG Sensor Settings

Select the items that should be displayed by the component:

☐ Diagram	☐ Blood pressure	☐ ECG code
✓ Heart rate	☐ SpO2	☐ Diagnosis

OK | Cancel

```
<component id="4" xsi:type="ecgMedget" screenRef="5">

  <config>
    <property>
      <enableArchive>true</enableArchive>
    </property>
  </config>


  <data>
    <item>
      <status>heartRate</status>
    </item>
  </data>
```

Actions | Add Action

Name: Heart Check | Delete Action

Expression

Variable	Operator	Value
Heart rate	Greater >	120

```
<conditions>
  <condition>
    <name>Heart Check</name>
    <expressions>
      <expression>
        <variable>heartRate</variable>
        <operator>greater</operator>
        <value>120</value>
      </expression>
    </expressions>
```

Action

SMS

Text	Heart Rate Over 120bpm
Phone number	123-456-789

✓ Include location ✓ Include sensor data

```
<actions>
  <action xsi:type="sms">
    <property>
      <text>Heart Rate Over 120bpm</text>
      <phoneNumber>123-456-789</phoneNumber>
      <includeLocation>true</includeLocation>
      <includeSensorData>true</includeSensorData>
    </property>
  </action>
```

Figure 5.11: An example showing the relationship between concrete syntax and metamodel for the *ECGSensor component* with the action send *SMS*.

" if *heartRate* is *greater* than *150*, then do the action *call* with *phone number 911* ".

The *Mobia Processor* extracts these values from the model and store them in a container. In cases where there are multiple expressions, an array of containers stores the values in the different expressions. Table 5.1 shows the container variables and the values stored.

Container Variable	**Value**
variable	*heartRate*
value	*150*
operator	*greater* (processed to be ">" sign)
actionType	*call*
phoneNumber	*911*

Table 5.1: The container variables and the model element values stored in them.

Take note that it does not matter if some of the tokens are of different data types. They are still stored as strings (i.e. series of characters) since the template does not care if it is a string or integer. If necessary, type checking should be done either by the *Mobia Processor*, or in the *Mobia Modeler* itself when the user is trying to input information during configuration. These values are then mapped to the code template shown in listing 5.9.

Listing 5.9: A snippet of the code template for a *sensor component* where the conditions are evaluated.

```
#foreach($condition in $CONDITIONSVARMAP)
node = conditions.new ExpressionNode(
        "$condition.get("variable")",
        "$condition.get("operator")",
        "$condition.get("value")",
#if( $condition.get("actionType") == "email" )
        new AndroidActions(
                this,
                "$condition.get("emailAddress")",
                "$condition.get("name")",
                "$condition.get("text")" )
#elseif($condition.get("actionType") == "call")
        new AndroidActions(
                this,
                "$condition.get("phoneNumber")")
#elseif($condition.get("actionType") == "sms")
        new AndroidActions(
                this,
                "$condition.get("phoneNumber")",
                "$condition.get("text")" )
#end
        );
conditions.add(node);
#end
```

Modeling Actions with *special components*. Since the possible actions (e.g. call, sms, etc.) have uniform code and just differ in terms of the data (e.g. phone number) they contain, we create a class called *AndroidActions* which contains APIs for invoking such actions. A snippet of the class is shown in listing 5.10. As we have seen in listing 5.9, method calls with the corresponding parameters are made to the *AndroidActions* class.

Listing 5.10: A class containing all implementations for invoking different types of actions supported by the *Mobia Framework*.

```
public class AndroidActions{
    public static void callNumber(Activity callingActivity, String telephoneNumber) {

        try{
                String command = "tel:" + telephoneNumber;
                Intent intent = new Intent(Intent.ACTION_CALL, Uri.parse(command));
                callingActivity.startActivity(intent);
        }
        catch( Exception e ){...}
        }
    }

    public static void sendSMS(final Activity callingActivity, String telephoneNumber,
        String message ){...}
    public static void sendEmail(Activity callingActivity, String emailAddress, String
        subject, String message ){...}
    .....
```

5.7 Summary

In this chapter, an in-depth view of the models in the *Mobia Framework* was presented. This included the different purposes of the types of model components (basic, structure, sensor and special), the concrete syntax (i.e. how the models are presented inside the *Mobia Modeler* and its equivalent XML form), the corresponding metamodel, and the semantics in order to show a proposed mapping between the model elements and the code templates. Some basic information about the *Android Framework* (e.g. concept of Activities and Intents) were also presented in order to show why specific design decisions for code generation were made.

Chapter 6

Summary and Future Work

In this chapter, we will try to revisit the goals of this research and see how they are achieved through a summary of the previous chapters. Some ideas for further study will also be presented in the future work section.

Contents

6.1 Summary and Conclusion

In chapter 1, the motivation and benefits of allowing non-programmers (i.e. end-user) to create their own mobile applications were presented. This includes the *utilization of the end-user's domain knowledge and ideas as inputs* to the actual creation of the mobile application, and the possible *reduction of development time and costs* of producing the applications since there is no need to hire programmers as long as tools were readily available. This paradigm in which non-programmers are empowered to develop or modify their own applications is called end-user development (EUD). As mentioned by Liebermann et al. [LPWK06], the challenge on EUD is on the *design of tools and frameworks* that would allow end-users to easily develop their own applications that will support them in their goals and needs. The main goal of this research is to discover ways in order to alleviate this problem and *propose a framework that would allow EUD particularly for mobile applications*. We initially looked into the creation of applications in the domain of *mHealth* as proof of concept.

Three questions were presented in chapter 1 in order to guide us throughout the duration of this study:

- What do end-users want to have in tools that allow EUD for this specific application domain (MHealth)?
- What design and functionality should tools for EUD of mobile applications provide?
- What is a good design for an EUD framework?

The general approach we have employed in order to answer these questions is a combination of *model-driven software development* approach particularly *domain-specific modeling*, and *user-centered iterative design*.

What do end-users want to have in tools that allow EUD for this specific application domain (MHealth)?

In order to discover what end users want from EUD tools, different user-centered activities were carried out as discussed in detail in chapter 2. A summary of the different UCD activities is shown in figure 6.1.

During the initial stages of this research wherein the tool prototypes were not yet available, *surveys and interviews* were conducted for the purpose of collecting information with regards to the desired features for the tools (section 2.2.1), evaluate initial designs (section 2.2.3), and validate the purpose or usefulness of such tools when completed (section 2.3.1). Since the initial domain we were looking into was in the domain of *mHealth*, experts in the medical domain (section 2.2.2) and researchers that focus on medical-related research (section 2.3.1) were some of the people involved in activities that *require the collection of domain-specific knowledge*. For collecting general information that does not require

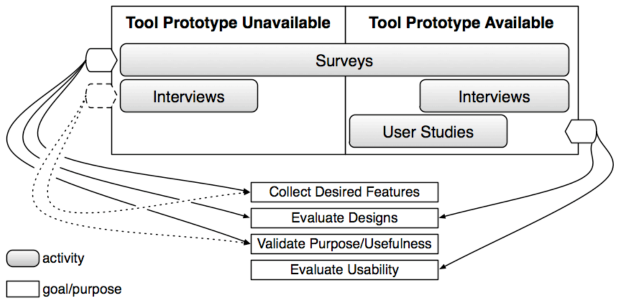


Figure 6.1: The different UCD activities performed throughout the duration of this research. Taken from chapter 2.

domain expertise such as the design and interaction of the tools involved (section 2.2.3), participants from different fields of expertise were involved in the activities.

During the time wherein the tool prototypes were already available, feedback in the form of surveys (section 2.2.4) and interviews (section 2.3.2) with experts in the medical field were also conducted to see if the existing prototype meets the needs for this domain. User studies were also conducted (sections 2.4, 3.2.4 and 3.2.5) in order to evaluate the usability of the tools. The typical activities carried out during the user study were:

- Participants were asked to explore the tools while the people conducting the studies observe the general interaction of the participants with the tools.
- The participants were then asked to perform several tasks (e.g. screen design task if applicable, add control logic to the application) wherein task times were measured. The typical hypothesis made was that, the tools that offered a more usable interface allowed the participants to carry out the tasks faster.
- The participants were then asked to give their feedback with regards to the tool by answering a questionnaire.

Based on the different UCD activities, results on what users want to have in tools for EUD can be summarized as follows:

- It is important that technically complex activities such as programming and mathematical computations are hidden from the user (sections 2.2.1 and 2.3.1).
- As much as possible, the tools should be graphical and interactive in nature (section 2.2.1), and should also feature interactions that they are already familiar with (section 2.2.3).
- Terminologies used in the tool (i.e. labels on the fields) should be similar to the terms that the users use in their respective domains (sections 2.2.4 and 2.3.2) such as in the health domain for example.

What design and functionality should tools for EUD of mobile applications provide?

The EUD tool we have iteratively designed, developed and evaluated throughout the duration of this research is called the *Mobia Modeler*. The combination of the different activities that influence the design and development of the different *Mobia Modeler* prototypes are shown in figure 6.2.

In order to discover which tool design and functionalities should be present in EUD tools, a closer look into existing tools and frameworks that allow mobile application development was carried out (sections 3.1.1, 3.1.2 and 3.1.3). This was combined with information collected through surveys (section 2.2.1) and interviews (section 2.3.1) with potential users. Also, how mobile constructs (i.e. user interface, input/output, control flow) in some of these surveyed tools and frameworks (section 3.1.4), including general design ideas from different software applications (section 3.1.5) were looked into. The types of information used for input and output specifically for the domain of *mHealth* were collected through surveys (section 2.2.2) from people in the medical field and surveys (section 2.2.3) from other potential users who are not necessarily experts in the medical domain. User studies were then carried out in order to evaluate the usability of the prototypes (sections 3.2.4 and 3.2.5).

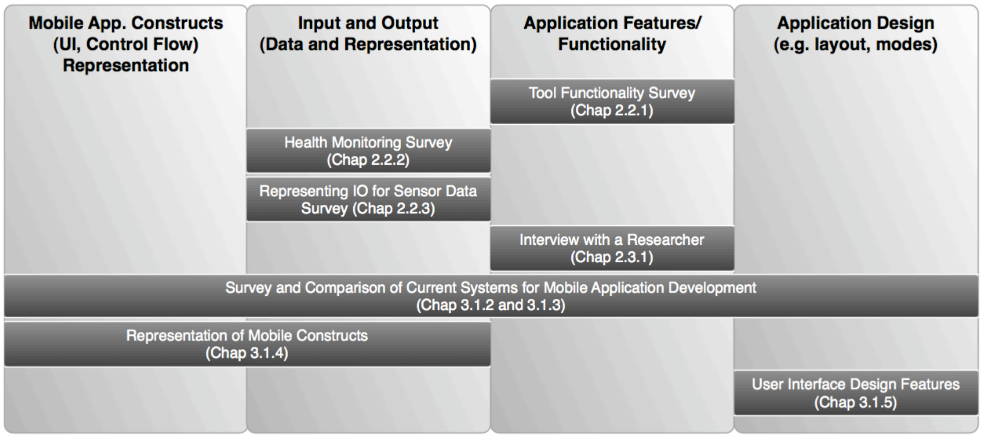

Mobile App. Constructs (UI, Control Flow) Representation
Input and Output (Data and Representation)
Application Features/ Functionality
Application Design (e.g. layout, modes)
Tool Functionality Survey (Chap 2.2.1)
Health Monitoring Survey (Chap 2.2.2)
Representing IO for Sensor Data Survey (Chap 2.2.3)
Interview with a Researcher (Chap 2.3.1)
Survey and Comparison of Current Systems for Mobile Application Development (Chap 3.1.2 and 3.1.3)
Representation of Mobile Constructs (Chap 3.1.4)
User Interface Design Features (Chap 3.1.5)

Figure 6.2: Summary of activities that influence the design and development of the *Mobia Modeler* prototypes. Taken from chapter 3.

A summary of the different *Mobia Modeler* prototypes that were created is shown in figure 6.3. The initial prototypes developed in order to explore the possible base frameworks to use for development, and also to combine the different designs surveyed were the *Mobia Piccolo* and *Mobia NBSuite*(section 3.2.3). The designs of the initial prototypes were then reorganized and developed into two distinct versions which were called *Mobia Integrated-View* and *Mobia Multi-View*. The goal for the creation of these two new prototypes was to evaluate which design promotes ease-of-learning and ease-of-use. The result of the user study showed that the *Mobia Integrated-View* which followed a *modeless design allowed* *the users to carry out their tasks faster and was the preferred version* by most of the participants. From this study, different types of users were observed according to their interaction with the prototypes and were presented in section 3.2.4: *Observing the User Experience.*

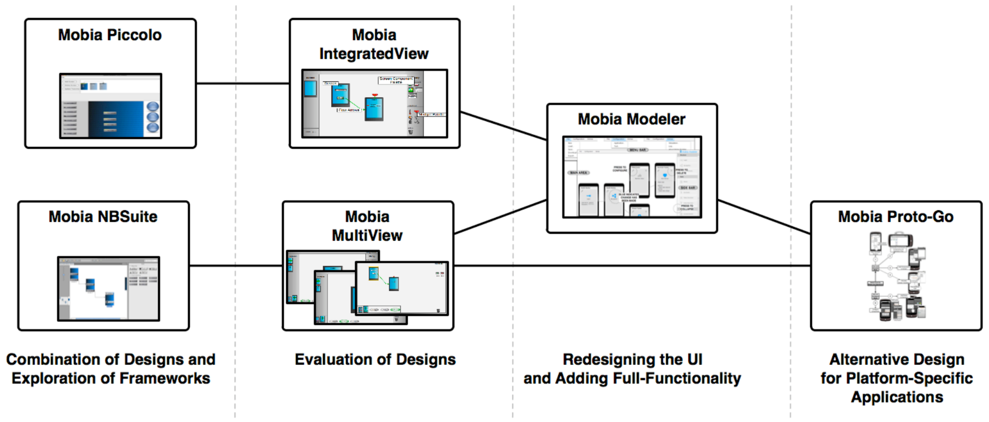

Mobia Piccolo
Mobia IntegratedView
Mobia Modeler
Mobia NBSuite
Mobia MultiView
Mobia Proto-Go
Combination of Designs and Exploration of Frameworks
Evaluation of Designs
Redesigning the UI and Adding Full-Functionality
Alternative Design for Platform-Specific Applications

Figure 6.3: A timeline of the different *Mobia Modeler* prototypes. Taken from chapter 3.

The integrated modeless design of the *Mobia Integrated-View* was then adapted to the *Mobia Modeler.* This version features a *configurable-component based design* which means that, instead of building mobile applications by combining individual user interface elements in the model, *components are combined and configured instead.* This tries to address the issues of modeling the user interface, application flow and modeling inputs and outputs that were mentioned in detail in section 3.2.5. The proposed approach tries to address the issues through the following ways:

- **Modeling the User Interface through Configuration.** Instead of allowing the user to create user interfaces by combining individual UI elements, a proposed way would be *to provide the user with configurable components that have already some predefined meaning and which can easily be configured to meet the user's needs.* In this way, the user can *concentrate on the solution to the problem domain and not be bothered about technical details* (UI elements to use, layout, etc.). Since the applications created are *domain-specific*, it is possible to define specific configurable components that solves problems in that domain.

  This solution also addresses *platform independence.* Since *different mobile devices have different features and may represent different user interface elements* (e.g. a textfield in one platform may be represented as another user interface component in another platform), it would make sense *to provide the user with a component that represents a solution to the problem instead.*

- **Modeling the Application Flow through Configuration.** The *Mobia Modeler structure component* is the proposed solution to creating application flow easily.

Structure components have a predefined meaning that allows an application to branch from one screen to the next depending on certain conditions that are defined through configuration. Control flow can easily be added to the model by dropping a *structure component* in the modeler's design area, in which the *Mobia Modeler* will automatically create a new screen and add the transition from the source to the target screen. All the user has to do is to configure the application logic for that certain *structure component*. Transformation and validation of inputs from the models for code generation is also made easier since the user is limited to using predefined conditions specific to the type of structure component there is.

- **Modeling Inputs and Outputs through Configuration.** In order to model complex logic that involves information taken from external devices (inputs), and the actions (output) that will be executed that satisfies certain conditions, the *Mobia Modeler Sensor component* is a proposed solution. The *sensor component* represent specific sensor devices which can easily be configured to model specific conditions and outputs.

The limitation of the current design of the *Mobia Modeler* is that, it only gives a high level overview of the components in the model and therefore limits the user from creating customized user interfaces. This feature was sacrificed for the sake of simplicity and platform independence. In order to overcome this limitation, a supplement tool that is fully interoperable with the *Mobia Framework* family of tools was introduced. The tool is called the *Mobia Proto-Go* which is a *platform-specific tool* that features user interface customization and runs on the target mobile device (section 3.2.6). The model created by the *Mobia Proto-Go* can be imported to the *Mobia Processor* which is responsible for automatic code generation. The model can also be imported to the *Mobia Modeler* to show the high-level (i.e. user interface is not shown) overview of the model. Another proposed way to overcome the limitation of the *Mobia Modeler* will be discussed in section 6.3.1 for future work.

What is a good design for an EUD framework?

In chapter 4, the details of our proposed framework which is called *Mobia Framework* was presented. This includes the details about the functional requirements, design and implementation of the framework. The underlying model and proposed transformation of the model to the Android framework was discussed in chapter 5.

Figure 6.4 shows an overview of the *Mobia Framework* and its parts. As we can see in the figure, the underlying model and the *Mobia Processor* are the parts that are used consistently no matter which front-end modeling tool is used (i.e. *Mobia Modeler* or *Mobia Proto-Go*). It is important that there is a **clear separation between the front-end and the underlying processor** since as we have observed during the different phases of this research, different users have different needs and preferences. Some users may prefer a

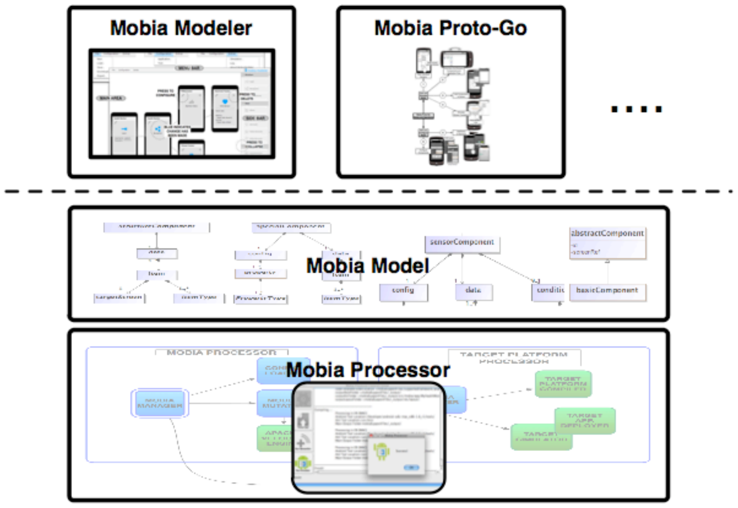


Figure 6.4: Clear separation between the *Mobia Processor* and front end tools with the model binding them together.

high level overview of the models and may not want to be bothered with UI design details (section 3.2.5). However, some may prefer to fully customize their own UI (section 2.2.4 and 3.2.6). Users should also not be limited to using one platform (i.e. Personal Computer, Mobile Device) in creating their applications.

There should be flexibility on whatever *design of the front-end tool* there is, what *technologies are used to create such tools*, and which *platform* these tools are running on. This is possible by having **a consistent model in between the two parts (i.e. modeler and processor) that serves as a bridge between them**.

6.2 Main Publications

Table 6.1 shows the publications related to this research, including the links to the chapters in which they are incorporated into.

6.3 Future Work

6.3.1 Bridging the gap between Non-Technical and Semi-Technical Users

In the development of the *Mobia Modeler* prototypes, the design approach for allowing the user to create the user-interface (UI) of the application being modeled are in two different ends of the abstraction spectrum (figure 6.5).

Publication	Description	Chapter
Mobia Modeler: Easing the Creation Process of Mobile Applications for Non-Technical Users *Florence Balagtas-Fernandez, Max Tafelmayer, Heinrich Hussmann* In Proceedings of the 15th International Conference on Intelligent User Interfaces (IUI 2010). Hong Kong China, Feb. 2010, ISBN 978-1-60558-515-4, pp. 269-272.	Configurable-component-based design of the modeling environment	3, 4, 5
Evaluation of User-Interfaces for Mobile Application Development Environments *Florence Balagtas-Fernandez, Heinrich Hussmann* In Proceedings of the 13th HCI International 2009 (HCII 2009). Town and Country Resort and Convention Center, San Diego, CA, USA, July 19-24 2009, ISBN 978-3-642-02573-0, pp. 204-213.	A look into some development environments Comparison of two designs of the *Mobia Modeler* (integrated vs. multiple-mode design)	3
Applying Domain-Specific Modeling to Mobile Health Monitoring Applications *Florence Balagtas-Fernandez, Heinrich Hussmann* In Proceedings of the 6th International Conference on Information Technology : New Generations (ITNG 2009). Las Vegas, Nevada, USA, April 27-29, 2009, ISBN 978-0-7695-3596-8, pp. 1682-1683.	Motivation and use-case scenarios in the target domain	1, 2
Modeling Information From Wearable Sensors *Florence Balagtas-Fernandez, Heinrich Hussmann* In Proceedings of the 4th International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI 2009). Sanibel Island, Florida, USA, February 8, 2009. CEUR Proceedings, Vol. 439, ISSN 1613-0073.	Evaluation on how to represent sensors in the modeling environment	2
Model Driven Development of Mobile Applications *Florence Balagtas-Fernandez, Heinrich Hussmann* In Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008) (Doctoral Symposium). L'Aquila Italy, September 2008, ISBN 978-1-4244-2187-9, pp. 509-512.	The general idea behind this research which is to apply model-driven development for mobile application creation	

Table 6.1: Selected list of publications related to this research.

The *Mobia Integrated-View* and *Mobia Multi-View* for instance allowed the users to design the user interfaces of the mobile application by combining and assembling individual UI elements on a screen which is similar to the functionality of UI toolkits (e.g. DroidDraw [Dro], Interface Builder [Appc]) and IDEs (e.g. Netbeans with Mobility Pack [Net]). However, in section 3.2.5, one of the issues mentioned is that of non-technical users being unaware of which UI elements to use for their application model. This issue was addressed with the configurable component-based design of the *Mobia Modeler* in which components are combined instead of individual UI elements.

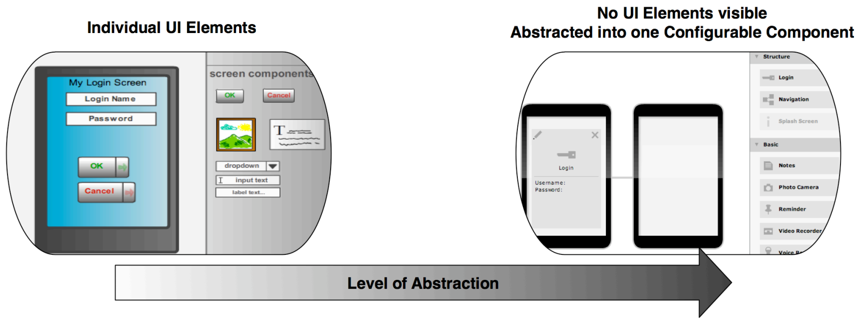


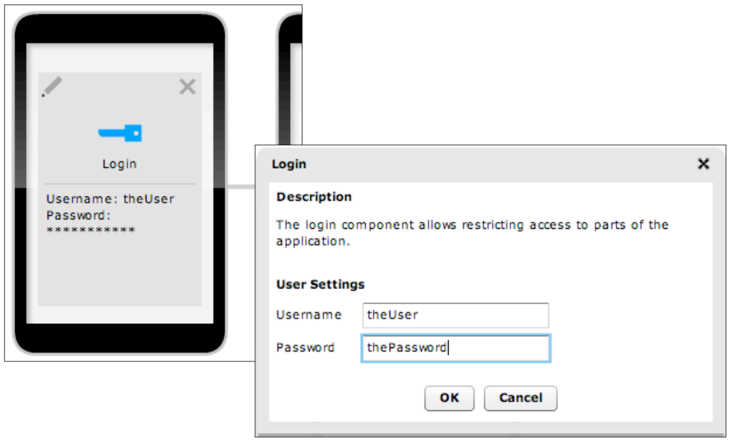
Figure 6.5: Different levels of abstraction for the *Mobia Modeler* prototypes.

However, as the users become more acquainted with the tool, they need to progress from being novices (i.e. non-technical users) to being semi-experts (i.e. semi-technical users) [vHVF09] and may need some way to be able to do more and not be limited by the tool. In terms of user interface design, this means that there should be a way for them to take control on which UI elements are present, including the control of the layout and adjustment of the look-and-feel of the user interface of the mobile application they are creating.

In the next subsections, some design ideas are proposed through an extension of the current design of the *Mobia Modeler* in order to accommodate different user expertise. The ideas are presented in a step-by-step manner that reflects the users' growth in knowledge as they use the tool.

Configurable Components for the Novice User

The design approach of the *Mobia Modeler* allows the the non-technical user to choose between a wide array of components and configure it based on the application being modeled (figure 6.6). This presents a very high level of abstraction in which the user does not need to identify which specific UI elements are needed in the application. The user just needs to focus on the data and context of use in order to configure a specific component .



Username: theUser
Password: ***********

Login

Description

The login component allows restricting access to parts of the application.

User Settings

Username: theUser

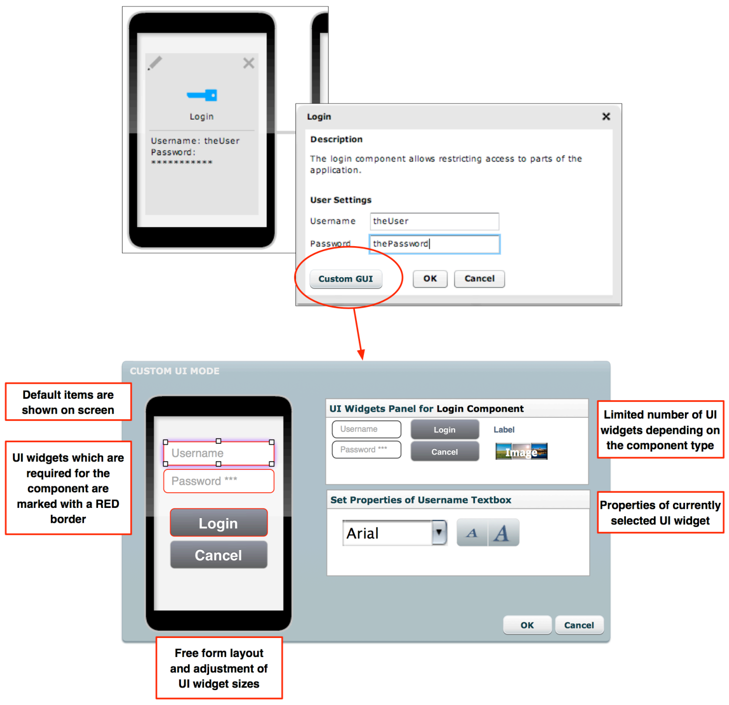
Password: thePassword

Figure 6.6: Configurable component-based design to accommodate the novice (i.e. non-technical) user.

Customizable Interfaces for the Semi-Technical User

As the user becomes more acquainted with the tool (i.e. the *Mobia Modeler*), or the user may have already some background in terms of UI elements for designing user interfaces, there must be a way to accommodate this increase in technical knowledge.

An alternative feature would be to provide a *custom UI mode* in the *Mobia Modeler*. In this mode, the user has the ability to customize the user interface of the mobile application being modeled. Figure 6.7 shows this design and possible features.


Login
Username: theUser
Password: ***********
CUSTOM UI MODE
Username
Password ***
Login
Cancel
UI Widgets Panel for Login Component
Label
Image
Set Properties of Username Textbox
Arial
A A
OK
Cancel
Default items are shown on screen
UI widgets which are required for the component are marked with a RED border
Limited number of UI widgets depending on the component type
Properties of currently selected UI widget
Free form layout and adjustment of UI widget sizes

Login

Description

The login component allows restricting access to parts of the application.

User Settings

Username theUser

Password thePassword

Custom GUI OK Cancel

Figure 6.7: An additional *custom UI mode* to accommodate the needs of the semi-technical user.

The *custom UI mode* is invoked through a button (e.g. Custom GUI button) in the configuration dialog as shown in figure 6.7. Once this button is pressed, a design palette is shown which contains: a screen, a UI widget palette, and a properties pane.

The following are proposed features of the *custom UI mode* in order to simplify the tasks of semi-technical users:

- In the beginning of the *custom UI mode*, all default UI elements that a component contains are already added to the screen. Those elements that are required (i.e. should always exist) in a component have a red border and cannot be deleted from the screen. The properties of required elements however can be modified through the properties pane.
  In the given example (figure 6.7), the *login* component has the required UI elements which will contain the username, password, and a button that would allow the user to log in. The *cancel* button is not a required component since some mobile platforms simply allow the user to use the *back button* on the device to go back to the previous state.
- In order to simplify the task of the user, only UI elements that are relevant to a certain type of component are made available in the UI widget palette. Furthermore, the UI elements are still abstracted depending on the purpose of the component.
  For example, in figure 6.7, since the component being configured is the *login* component, instead of providing generic UI elements such as Textfields/Textboxes and

buttons, dedicated *username box* and *password box* are provided for the input fields, and *login* and *cancel* for the buttons. Other optional elements provided are labels and images.
- A simplified properties pane should be made available. Only the most commonly set properties for a certain UI widget are shown.

Giving Full Control to the Expert User

In the case of the expert user who presumably has an idea what UI elements are and how to use them, full access to the variety of available UI elements in the chosen target platform/s can be made available in the modeler. This functionality can be accessed inside the *custom UI mode* where a button to the *advance user* interface is provided (figure 6.8).

During the configuration wizard, the user is asked which target platforms/devices would be used for the mobile application they are modeling. Depending on the number of target platforms there is, the expert user should be able to see what the UI would look like in these platforms.

The proposed design is similar to the *custom UI mode* discussed in the previous section as shown in figure 6.8. The difference between the expert mode and the previous mode are the following:

- The UI elements that are made available to the user in the UI widget palette are those that are from the target platforms (e.g. Android platform).
- The UI elements are separated in tabs in the UI widget panel. Selecting a tab would change the *screen* which is equivalent to the target platform as shown in the figure.
- In order to have consistency in terms of the UI elements added to the screen between the different platforms, a UI widget which is added to one platform is automatically added to the others provided so that it is also available in the other platforms. Only the layout and look-and-feel for the other platforms are not affected.

As we can see here, the implication for such design is that, a separate section in the underlying model (e.g. the XML form of the model) should provide an area wherein specific UI information for each target platform are stored. The underlying model would be similar to the generated model by the *Mobia Proto-Go* wherein the layout information is stored in the screen section of the model (section 3.2.6). The *Mobia Processor* will then process this data and assign it to the specific target platform.

6.3.2 Towards an Easily Extensible and Configurable Framework

Currently, extending the capabilities of the different parts of the *Mobia Framework* (*Mobia Modeler* and *Mobia Processor*) entails manually modifying the source code in order to accommodate additional components and target platforms.

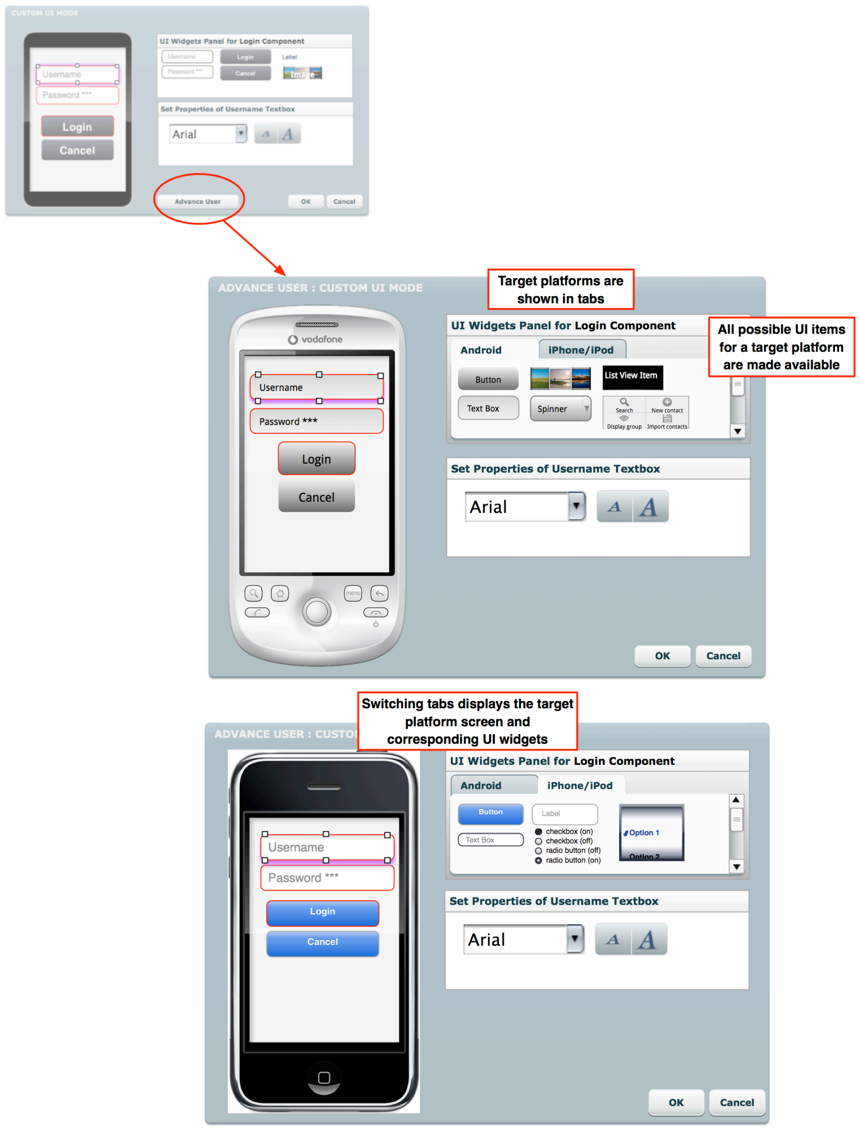

CUSTOM UI MODE
UI Widgets Panel for Login Component
Set Properties of Username Textbox
Advance User
OK
Cancel
Target platforms are shown in tabs
ADVANCE USER : CUSTOM UI MODE
All possible UI items for a target platform are made available
Android
iPhone/iPod
Username
Password ***
Login
Cancel
Button
Text Box
Spinner
List View Item
Arial
Switching tabs displays the target platform screen and corresponding UI widgets
Label
checkbox (on)
checkbox (off)
radio button (off)
radio button (on)
Option 1
Option 2

Figure 6.8: Platform-specific UI elements are made available to the expert user.

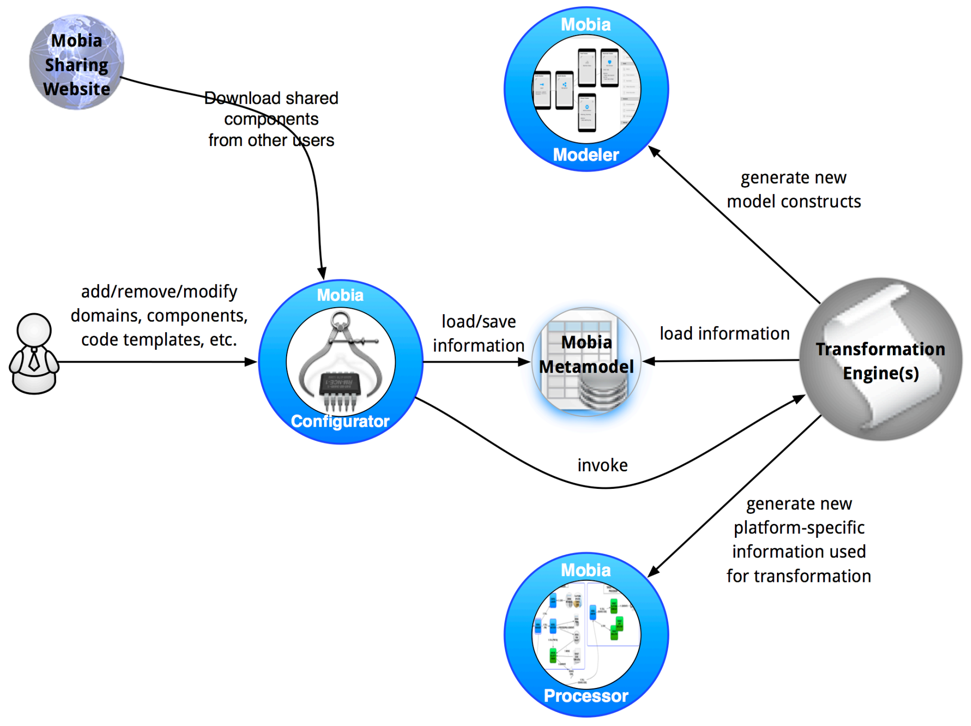

Mobia Sharing Website
Download shared components from other users
Mobia Modeler
generate new model constructs
add/remove/modify domains, components, code templates, etc.
Mobia Configurator
load/save information
Mobia Metamodel
load information
Transformation Engine(s)
invoke
generate new platform-specific information used for transformation
Mobia Processor

Figure 6.9: Overview of *Mobia Framework* tools with the proposed *Mobia Configurator*.

One proposed solution is to create and integrate an additional tool that binds the different parts of the framework together. We call this the *Mobia Configurator*. This would be similar to the MetaEdit+ Workbench [Metb] which allows the creation and customization of the MetaEdit+ Modeler.

The current design of the *Mobia Processor* is that it relies on the *Mobia Metamodel* which contains the description of the available components in the model. In the case of the *Mobia Configurator*, this same *Mobia Metamodel* can be used to configure both the functionality of the *Mobia Modeler* (e.g. add new domains and components), and the *Mobia Processor* (e.g. add new target platforms, code templates). The idea of having an underlying model act as a *central hub* was proposed by Pleuss et al. [PVH07]. In our case, tool support in the form of the *Mobia Configurator* can be used to help in modifying this central model in terms of the customization of the different tools (e.g. modeling environment, transformation tools) involved.

Figure 6.9 shows an overview of the different tools in the *Mobia Framework* including the possible processes and interactions with the *Mobia Configurator*. The *Mobia Configurator* loads the currently supported settings stored inside the underlying *Mobia Metamodel* and presents it to the user. The user can add, delete or modify new domains, components and other artifacts needed for the code transformation such as code templates, configuration information, etc. Changes made by the user are stored in the *Mobia Metamodel* which is then processed by transformation engines/tools in order to generate the new components for the *Mobia Modeler* and add transformation support to the *Mobia Processor*. Aside from the user manually adding new features for the *Mobia Framework*, a dedicated website can also be provided to allow *Mobia Framework* users to share their own components and download components other users have created. This is similar to the idea of using plugins created by other developers for web browsers in order to add new functionality to the browser, or adding new API support to IDEs in order to develop for a different platform which is not currently supported by the IDE.

Challenges. The challenges we need to consider in the design of the *Mobia Configurator* would first be on the design of the tool itself. We need to look at how to design the presentation of the interface such that adding, deleting and modifying parts of the framework (e.g. domains, components, target platform information) would not be complicated. Also, the underlying processing of the *Mobia Metamodel* and transformations to the different parts of the *Mobia Framework* should be made seamless. This entails an in-depth analysis of the current architectural design of the *Mobia Modeler* and *Mobia Processor* in order to find a way to make this possible.

6.4 Closing Remarks

In the present era, development of software applications is slowly shifting from professionally trained developers to non-programmers (i.e non-technical users) through end-user development (EUD). The responsibility of the professional software developer is now being extended to the design and development of EUD tools that would allow non-technical people transform their ideas into reality.

However, the challenge for people both in the software engineering and the HCI community is on finding out which designs and functionalities should be integrated into EUD tools in order to encourage non-technical people to utilize it to produce their own application solutions. Aside from user-related problems, another challenge is on the design of the underlying framework such that solutions can easily be extended.

This thesis proposes one solution which involves an iterative user-centered design and development of the front-end tools, and a model-driven development approach for the underlying framework. Models are used as a bridge to separate the different parts of the framework for more flexibility and extensibility.

We hope that our own experiences as documented in this thesis will guide future researchers and developers in their own quest of developing tools that will empower end-users to create their own software applications for whatever platform or purpose it may be.

Appendices

Appendix A

Mobia Framework Processor

A.1 Implementation Technologies

Technology	Link	Details
Java Platform Standard Edition 6	http://java.sun.com/javase/	Main programming platform.
Apache Velocity Engine 1.6.2	http://velocity.apache.org/	Used for generating the final source code files by combining templates and information from the Mobia-created models.
JDOM	http://www.jdom.org/	Used for parsing the input files.
Android 1.6 release 1	http://developer.android.com/sdk/android-1.6.html	Tools used to generate, compile and deploy an Android project.
Apache ANT 1.7	http://ant.apache.org/	Used to automate the build process of the generated project.

A.2 Input/Output Files and Folders

Folders and Files					Purpose
Level 1	**Level 2**	**Level 3**	**Level 4**	**Level 5**	
mobiaSupportFiles/	MobiaConfig.mainconfig				The main configuration file containing information about the folder locations for each of the supported frameworks and its related information, including the folder where the Mobia Metamodel is stored (PIM folder).
	MobiaConfig.supportTools				Contains script commands (e.g. ant, android)
	_output/				Contains the generated files (source folders and code, manifest files, etc.)
	pim/	mobiaPimSchema.xsd			XML Schema for the Mobia Metamodel
		actionTypes/	*.xsd		
		components/	*.xsd		
		domains/	*.xsd		
		tool/	*.xsd		
		screen/	*.xsd		
	supportedFramework/	**android/**	**androidTools/**	ToolsLocation.config.MAC ToolsLocation.config.WIN	Contains information about the location of the platform-specific tools (e.g. compilers)
			codetemplate/	*.vm	Contains code templates.
			config/	OutputFolderList.config	Contains information about the output location.
		genPurposeTools/	ToolsLocation.config.MAC ToolsLocation.config.WIN		Contains information about the location of the general

A.3 Configuration Files

The *Configuration Loader* loads two types of configuration files:

- The *MobiaConfig.mainconfig* which is the main configuration file containing information about the folder locations for each of the supported frameworks and its related information, including the folder where the *Mobia Metamodel* is stored (PIM folder).

```
#MOBIA MAIN CONFIG FILE
#NOTES:
#       # indicates comments
#       $ indicates variables (variables should be declared in the first few lines before you use them
#       (+) appends
#       Don’t␣forget␣to␣add␣"/"␣at␣the␣end␣of␣each␣folder

#␣The␣contents␣of␣this␣file␣should␣have␣an␣equivalent␣in␣the
#␣␣␣␣␣␣␣␣mobia.configloader.MobiaConfigConstants

#PIM␣SCHEMA␣LOCATION␣AND␣FILE
MOBIA_PIM_XSD_FILE␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣=␣mobilemSchema.xsd

#SUPPORTED␣FRAMEWORKS
ANDROID_FRAMEWORK_FOLDER␣=␣android/

#FOLDER␣LOCATIONS
MOBIA_PIM_FOLDER␣=␣mobiaSupportFiles/pim/
MOBIA_FRAMEWORK_OUTPUT_APPLICATION_FOLDER␣=␣mobiaSupportFiles/_output/
MOBIA_SUPPORTED_FRAMEWORK_FOLDER␣=␣mobiaSupportFiles/supportedframework/

#␣Change␣the␣values␣three␣for␣different␣supported␣framework
MOBIA_PSM_FOLDER␣=␣$MOBIA_SUPPORTED_FRAMEWORK_FOLDER␣+␣$ANDROID_FRAMEWORK_FOLDER␣+␣psm/
MOBIA_CODE_TEMPLATE_FOLDER␣=␣$MOBIA_SUPPORTED_FRAMEWORK_FOLDER␣+␣$ANDROID_FRAMEWORK_FOLDER␣+␣codetemplate/
MOBIA_FRAMEWORK_CONFIG_FOLDER␣=␣$MOBIA_SUPPORTED_FRAMEWORK_FOLDER␣+␣$ANDROID_FRAMEWORK_FOLDER␣+␣config/
MOBIA_FRAMEWORK_OUTPUT_CONFIG_FILENAME␣=␣$MOBIA_FRAMEWORK_CONFIG_FOLDER␣+␣OutputFolderList.config

#TOOL␣SPECIFIC␣INFO
#ANDROID_COMMAND_LOCATION␣=␣$MOBIA_SUPPORTED_FRAMEWORK_FOLDER␣+␣$ANDROID_FRAMEWORK_FOLDER␣+␣androidTools/tools/
#ANT_TOOL_LOCATION␣=␣$MOBIA_SUPPORTED_FRAMEWORK_FOLDER␣+␣genPurposeTools/ant-1.7.1/bin/
#␣The␣ToolsLocation.config␣files␣will␣contain␣the␣location␣of␣the␣executable␣files␣used
#␣for␣compilation
ANDROID_COMMAND_LOCATION_WIN␣=␣$MOBIA_SUPPORTED_FRAMEWORK_FOLDER␣+␣$ANDROID_FRAMEWORK_FOLDER␣+␣androidTools/ToolsLocation.config.WIN
ANT_TOOL_LOCATION_WIN␣=␣$MOBIA_SUPPORTED_FRAMEWORK_FOLDER␣+␣genPurposeTools/ToolsLocation.config.WIN
ANDROID_COMMAND_LOCATION_MAC␣=␣$MOBIA_SUPPORTED_FRAMEWORK_FOLDER␣+␣$ANDROID_FRAMEWORK_FOLDER␣+␣androidTools/ToolsLocation.config.MAC
ANT_TOOL_LOCATION_MAC␣=␣$MOBIA_SUPPORTED_FRAMEWORK_FOLDER␣+␣genPurposeTools/ToolsLocation.config.MAC

#KEYSTORE
ANDROID_KEYSTORE=␣$MOBIA_SUPPORTED_FRAMEWORK_FOLDER␣+␣$ANDROID_FRAMEWORK_FOLDER␣+␣mobia.keystore

␣␣␣␣␣␣␣␣
```

- The *MobiaConfig.supportTools* contains information about the folder locations for external tools used in the processor.

```
#MOBIA COMMANDS FILE/SCRIPT COMMANDS
#NOTES:
#       # indicates comments
#       $ indicates variables (variables should be declared in the first few lines before you use them
#       (+) appends
#       Don't␣forget␣to␣add␣"/"␣at␣the␣end␣of␣each␣folder
#␣␣␣␣␣␣␣No␣spaces␣please.␣Use␣the␣variable␣$SPACE␣instead␣to␣indicate␣space


#␣ANT
DEBUG␣=␣debug
ANT_COMMAND_WIN␣=␣ant.bat
ANT_COMMAND_MAC␣=␣ant

#␣ANDROID
ANDROID_COMMAND_WIN␣=␣android.bat
ANDROID_COMMAND_MAC␣=␣android
ADB_COMMAND_WIN␣=␣adb
ADB_COMMAND_MAC␣=␣adb

␣␣␣␣␣␣␣␣
```

- The *OutputFolderList.config* which contains information specific to a target platform in which code will be generated into. This is dependent on the structure of the target platform.

```
#MOBIA MAIN CONFIG FILE
#NOTES:
#       # indicates comments
#       $ indicates variables (variables should be declared in the first few lines before you use them
#       (+) appends
#       Don't␣forget␣to␣add␣"/"␣at␣the␣end␣of␣each␣folder
#␣This␣file␣contains␣the␣structure␣of␣the␣output␣folders␣for␣an␣application

#ANDROID_ROOT_FOLDER␣=␣mobiaGenerated/
#ANDROID_SOURCE␣=␣$ANDROID_ROOT_FOLDER␣+␣src/
#ANDROID_RES_LAYOUT␣=␣$ANDROID_ROOT_FOLDER␣+␣res/layout/
ANDROID_SOURCE␣=␣src/
ANDROID_RES_LAYOUT␣=␣res/layout/
ANDROID_RES_DRAWABLE␣=␣res/drawable/
ANDROID_DRAWABLE␣=␣drawable/

#AUTO_GEN␣=␣$ROOT_FOLDER␣+␣gen/
#ASSETS␣=␣$ROOT_FOLDER␣+␣assets/
#RES_DRAWABLE␣=␣$ROOT_FOLDER␣+␣res/drawable/
#RES_VALUES␣=␣$ROOT_FOLDER␣+␣res/values/
␣␣␣␣␣␣␣␣
```

A.4 Packages and Classes

Package	*Class*	Purpose
mobia.main	***MobiaManager***	The Mobia Manager class and the starting point for running the Mobia Processor.
mobia.configloader	***MobiaConfigConstants***	Used by the configuration reader in order to assign values to the data in the configuration files
	MobiaConfigInfoMap	Used to store all the configuration information
	MobiaConfigLoader	The Configuration Loader class responsible for processing the configuration files with the help of the MobiaConfigurationReader class.
	MobiaConfigReaderUtility	Reads mobia specific configuration files. NOTE: Change the values inside the MobiaConfig.MainConfig for different target platforms.
mobia.modelmutator	***MobiaPimSchemaConstants***	This contains constants used by the XSD Schema Parser for the Mobia Metamodel. TODO DEVELOPER: Add keywords here if additional keywords are used in the XSD file
	MobiaPimSchemaParser	Parses the PIM schema file in order to know which model components can be expected from the PIM file.
mobia.modelmutator.pim	***MobiaPimObjectInstanceLoader***	Contains the actual information from the Mobia Modeler PIM model instance. The contents depend on the contents found in the MobiaPimObjectTemplate
	MobiaPimObjectTemplate	Contains a Template for the pim file. This will be used by the MobiaPimObjectInstance later in order to store information from the model instance. Its structure is based on the MobiaMetamodel XSD file (schema)
mobia.modelmutator.psm	***MobiaPsmInstanceLoader***	Parent class of all PSM instances.
mobia.modelmutator.psm.android	***AndroidActivityContainer***	Contains information for each Activity (i.e. one screen instance) in Android
	AndroidApplicationContainer	Contains information about one project/application. WHat is basically present in the Manifest file.
	MobiaAndroidPsmInstanceLoader	Loads the data from the MobiaPimOject to an Android PSM instance.
mobia.codegenerator	***MobiaApacheVelocityEngine***	Class that uses the Apache Velocity Engine to merge information from the PSM container to the templates and generate the source codes.
mobia.arbiter	***MobiaArbiter***	Responsible for compilation of the generated source code and deployment of the generated application.
mobia.utility	***MobiaFileUtility***	Contains all helper methods for files that can be used by other classes
	MobiaOSUtility	Contains all helper methods for OS-specific operations.

Appendix B

Mobia Metamodel

The Mobia Metamodel is expressed using XML Schema Document (XSD) constructs. An XML Schema Editor such as the one from Oxygen XML Editor[1] was used to ensure that the schema is valid. In order to easily visualize the elements and relationships in the metamodel, a UML diagram is created (figure B.1).

General Model Structure. The general model structure of an application modeled using the *Mobia Modeler* consists of four main sections namely: *meta*, *tool*, *screens* and *components*. The *meta and tool* sections contain meta information (e.g. domain-specific) and tool-specific information (i.e. *Mobia Modeler* attributes) respectively. Information about the application being modeled is contained in the *screens and components* section.

The *meta section* contains additional information about the application being modeled which are not related to specific functionalities of the application itself (e.g. application name, domain name, target users etc.). The information in the *meta* section is supplied by the model creator during the configuration wizard. This can also be modified by choosing *Configuration→Application* in the menu.

The *tool section* of the model contains information with regards to the appearance of the *Mobia Modeler*. This information is incorporated into the model so that every time the modeler loads a previously created model, the appearance of the modeler will be adapted based on the information in this section. Similar to the *meta* section, the information in the *tool* section is supplied by the model creator during the configuration wizard. This can also be modified by choosing *Configuration* → *Tool* in the menu. As of the moment, this section contains information such as font size, locale (i.e. language such as English or German) and sidebar orientation (e.g. left, right).

[1] http://www.oxygenxml.com/

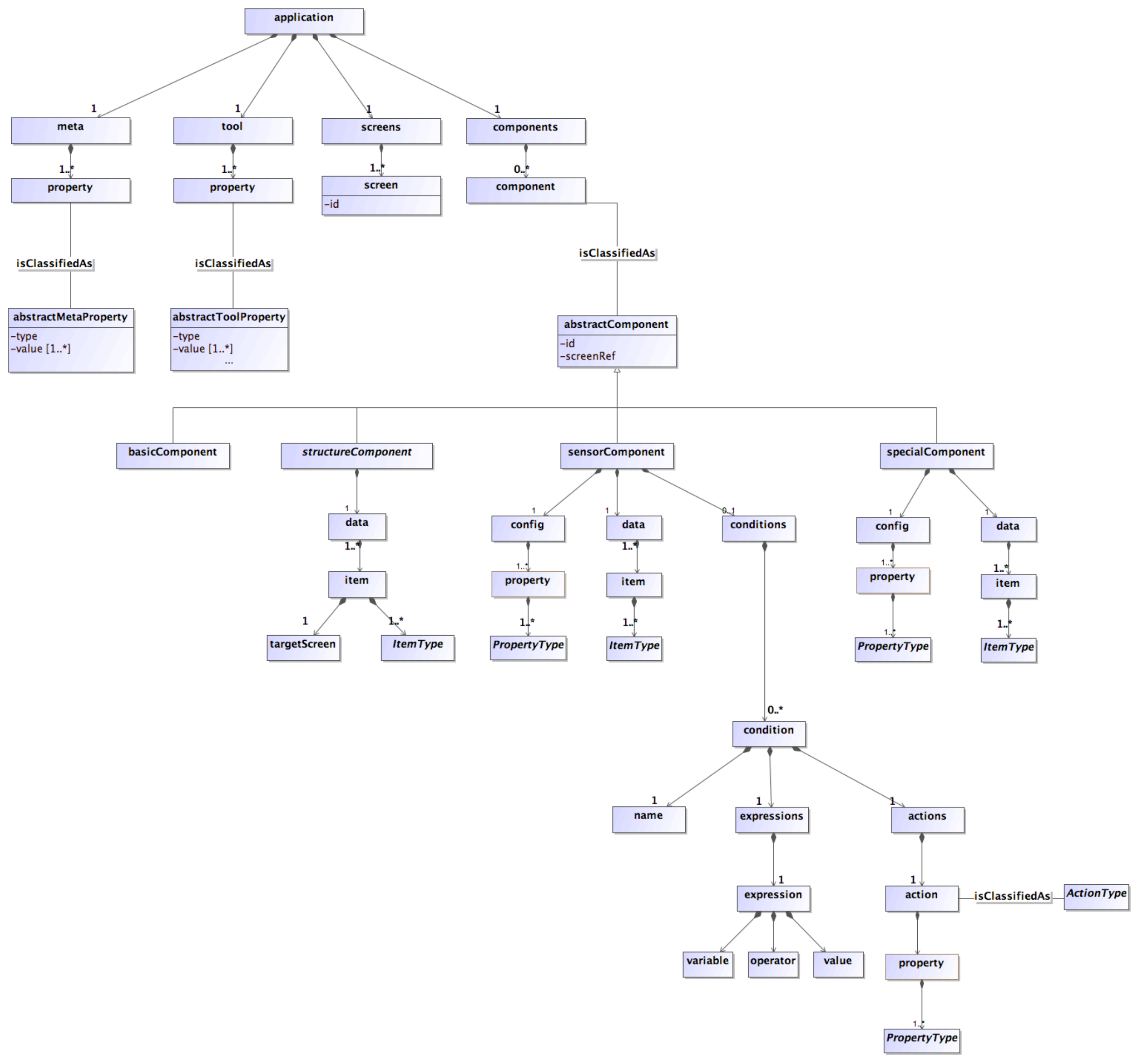


Figure B.1: A compact view of the Mobia Metamodel.

The *screens section* acts as a container for the *screen* instances in the modeled application. A *screen* instance only has one attribute which is the screen *id*. This is used later on to associate a certain *component* to a certain *screen*. This association is expressed by having a *component* store a *screen's id* as a reference.

The *components section* acts as a container for the different *component* instances in the modeled application. A *component* can be any of the following types: *basic, structure, sensor and special.*

Bibliography

[201a] Shift-work research: Where do we stand, where should we go? *Sleep and Biological Rhythms*, 8(2):95–105.

[201b] The state of mobile apps. http://blog.nielsen.com/nielsenwire/online_mobile/the-state-of-mobile-apps/. accessed 31 august 2010.

[Ali] Alive technologies products. http://www.alivetec.com. accessed 21 december 2009.

[Anda] Android development tools plugin for eclipse. http://developer.android.com/guide/developing/eclipse-adt.html.

[Andb] Android market. http://www.android.com/market/. accessed 28 december 2009.

[Andc] Android platform. http://code.google.com/android/documentation.html. accessed 04 january 2010.

[Appa] App inventor for android. http://www.appinventor.org/.

[Appb] Apple app store. http://www.apple.com/iphone/apps-for-iphone/. accessed 28 december 2009.

[Appc] Apple developer connection, iphone dev center. http://developer.apple.com/iphone/. accessed 04 january 2010.

[ASW05] Nicholas A. Allen, Clifford A. Shaffer, and Layne T. Watson. Building modeling tools that support verification, validation, and testing for the domain expert. In *Proceedings of the 37th conference on Winter simulation*, Orlando, Florida, 2005. Winter Simulation Conference. 1162782 419-426.

[BC08] Fabio Buttussi and Luca Chittaro. Mopet: A context-aware and user-adaptive wearable system for fitness training. *Artif. Intell. Med.*, 42(2):153–163, 2008.

[Ben82] Patricia Benner. From novice to expert. *The American Journal of Nursing*, 82(3):402–407, 1982.

[BFH09a] Florence Balagtas-Fernandez and Heinrich Hussmann. Applying domain-specific modeling to mobile health monitoring applications. pages 1682–1683, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[BFH09b] Florence Balagtas-Fernandez and Heinrich Hussmann. Evaluation of user-interfaces for mobile application development environments. In *Human-Computer Interaction. New Trends*, volume 5610/2009, pages 204–213. Springer Berlin/Heidelberg, 2009.

[BFH09c] Florence Balagtas-Fernandez and Heinrich Hussmann. Modeling information from wearable sensors. In *MDDAUI '09- Model Driven Development of Advanced User Interfaces 2009*, volume 439, Sanibel Island, USA, 2009. CEUR Proceedings.

[BFTH10] Florence Balagtas-Fernandez, Max Tafelmayer, and Heinrich Hussmann. Mobia modeler: Easing the creation process of mobile applications for non-technical users. In *Proceedings of the 15th International Conference on Intelligent User Interfaces (IUI 2010), Hong Kong, China, Feb. 2010*. ACM New York, NY, USA, February 2010.

[BHSW07] Tom Broens, Aart Van Halteren, Marten Van Sinderen, and Katarzyna Wac. Towards an application framework for context aware m-health applications. *Int. J. Internet Protoc. Technol.*, 2(2):109–116, 2007. 1357871.

[Blo02] Stefan Blomkvist. Persona - an overview. Extract from the paper: The User as a personality. Using Personas as a tool for design. Position paper for the course workshop "Theoretical perspectives in Human-Computer Interaction" at IPLab, KTH, 2002.

[BMC$^+$06] G. Bartolomeo, N. Blefari Melazzi, G. Cortese, A. Friday, G. Prezerakos, and R. Walker S. Salsano. Sms: Simplifying mobile services - for users and service providers. In *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services*. IEEE Computer Society, 2006. inproceedings.

[BMSBM07] G. Bartolomeo, F. Martire, S. Salsano, and N. Blefari-Melazzi. Simple mobile services, a service creation architecture for mobile services. In *Wireless World Research Forum, Meeting 18*, 2007.

[BRW07] Gregor Broll, Enrico Rukzio, and Björn Wedi. Authoring support for mobile interaction with the real world. *Late Breaking Results & Posters in conjunction with Pervasive 2007*, 2007.

[BZJ04] Muhammad Ali Babar, Liming Zhu, and Ross Jeffery. A framework for classifying and comparing software architecture evaluation methods. pages 309–309, 2004.

[CL09] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. On non-functional requirements in software engineering. Conceptual Modeling: Foundations and Applications, 2009.

[Con09] Vital Wave Consulting. Mhealth for development: The opportunity of mobile technology for healthcare in the developing world. http://www.unfoundation.org/global-issues/technology/mhealth-report.html. Technical report, 2009.

[Cza05] K. Czarnecki. Overview of generative software development. *Lecture Notes in Computer Science*, 3566:326, 2005.

[DB07] Jürgen Dunkel and Ralf Bruns. Model-driven architecture for mobile applications. *Business Information Systems*, pages 464–477, 2007.

[DD80] Stuart Dreyfus and Hubert Dreyfus. *A five-stage model of the mental activities involved in directed skill acquisition.* Storming Media, 1980.

[DFAB93] Alan Dix, Janel Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction.* Prentice Hall International, 1993. book.

[DMLG09] Amir Dotan, Neil Maiden, Valentina Lichtner, and Lola Germanovich. Designing with only four people in mind? – a case study of using personas to redesign a work-integrated learning support system. In Tom Gross, Jan Gulliksen, Paula Kotzé, Lars Oestreicher, Philippe Palanque, Raquel Oliveira Prates, and Marco Winckler, editors, *Human-Computer Interaction – INTERACT 2009*, volume 5727, pages 497–509. Springer, 2009.

[Dro] Droiddraw. http://www.droiddraw.org/. accessed 04 january 2010.

[FH03] Andy Field and Graham Hole. *How to Design and Report Experiments.* 2003.

[Fra03] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing.* Wiley Publishing, Inc., 2003.

[FSH$^+$01] L. Fernando Friedrich, John Stankovic, Marty Humphrey, Michael Marley, and John Haskins. A survey of configurable, component-based operating systems for embedded applications. *IEEE Micro*, 21(3):54–68, 2001.

[Fug00] Alfonso Fuggetta. Software process: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, Limerick, Ireland, 2000. ACM. 336521 25-34.

[Gam] Gamesalad creator. http://gamesalad.com/. accessed 04 january 2010.

[Gli07] Martin Glinz. On non-functional requirements. In *15th IEEE International Conference on Requirements Engineering*, pages 21–26, 2007.

[Goo01] Kim Goodwin. Perfecting your personas. *Cooper Interaction Design Newsletter. http://www.cooper.com/journal/2001/08/perfecting_your_personas.html*, August 2001.

[Gre04] Jack Greenfield. Software factories: assembling applications with patterns, models, frameworks and tools (november 2004). http://msdn.microsoft.com/en-us/library/ms954811(classic).aspx. accessed 30 january 2010., 2004.

[GS03] Jack Greenfield and Keith Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, New York, NY, USA, 2003. ACM.

[GSCK03] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software factories: assembling applications with patterns, models, frameworks and tools.* Wiley Publishing, Inc., 2003.

[HCM04] Richard Hull, Ben Clayton, and Tom Melamed. Rapid authoring of mediascapes. In *UbiComp 2004: Ubiquitous Computing*, pages 125–142. Springer Berlin / Heidelberg, 2004. UbiComp 2004: Ubiquitous Computing.

[HS08] Paul Holleis and Albrecht Schmidt. Makeit: Integrate user interaction times in the design process of mobile applications. In *Pervasive '08: Proceedings of the 6th International Conference on Pervasive Computing*, pages 56–74, Berlin, Heidelberg, 2008. Springer-Verlag.

[IJZ04] R.S.H. Istepanian, E. Jovanov, and Y.T. Zhang. Guest editorial introduction to the special section on m-health: Beyond seamless mobility and global wireless health-care connectivity. *Information Technology in Biomedicine, IEEE Transactions on*, 8(4):405–414, 2004. article.

[Kar96] John Karat. User centered design: quality or quackery? *Interactions*, 3(4):18–20, 1996. 234814.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain Specific Modeling, Enabling Full Code Generation.* IEEE Computer Society Publications and John Wiley and Sons Inc, 2008. book.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Pearson Education, Inc., Boston, USA, 2003. book.

[LG06] Peter Leijdekkers and Valerie Gay. Personal heart monitoring and rehabilitation system using smart phones. In *ICMB '06: Proceedings of the International Conference on Mobile Business*, page 29, Washington, DC, USA, 2006. IEEE Computer Society.

[LHL04] Yang Li, Jason I. Hong, and James A. Landay. Topiary: a tool for prototyping location-enhanced applications. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, Santa Fe, NM, USA, 2004. ACM.

[LPWK06] Henry Lieberman, Fabio Paternò, Volker Wulf, and Markus Klann. *End-User Development: An Emerging Paradigm*, volume 9, pages 1–8. Springer Netherlands, 2006.

[MB01] Ruth Malan and Dana Bredemeyer. Defining non-functional requirements. http://www.bredemeyer.com/pdf_files/nonfunctreq.pdf. accessed 07 july 2010., 2001.

[Meta] Metaedit+ domain specific modeling (dsm) environment. http://www.metacase.com/products.html. accessed 04 january 2010.

[Metb] Metaedit+ workbench. http://www.metacase.com/mwb/. accessed 04 january 2010.

[MIC] Model integrated computing website. http://www.isis.vanderbilt.edu/research/mic. accessed 30 january 2010.

[Mid] Midwife definition. http://www.medterms.com/script/main/art.asp?articlekey=4384. accessed 11 april 2010.

[ML08] Martin Malmsten and Henrik Lindström. User-centred design and agile development: Rebuilding the swedish national union catalogue. *Code4Lib Journal*, (5), 2008.

[Moba] Mobhealth - a mobile health framework (diplomarbeit by bernhard engstler from lmu medieninformatik).

[Mobb] Mobhealth framework. http://sourceforge.net/projects/mobhealth.

[Mod] Modelbaker from widget press. http://www.widgetpress.com/modelbaker. accessed 04 january 2010.

[Mur99] Tom Murray. Authoring intelligent tutoring systems: An analysis of the state of the art. *International Journal of Artificial Intelligence in Education*, 10:98–129, 1999.

[Mye86] Brad Myers. Visual programming, programming by example, and program visualization: a taxonomy. *SIGCHI Bull.*, 17(4):59–66, 1986. 22349.

[Net] Netbeans with mobility pack. http://netbeans.org/features/javame/index.html. accessed 04 january 2010.

[Nik] Nike + ipod sports kit. http://www.apple.com/ipod/nike. accessed 21 december 2009.

[Nor02] Donald Norman. *The Design of Everyday Things*. Doubleday Business, 2002.

[OFM06a] Nuria Oliver and Fernando Flores-Mangas. Healthgear: A real-time wearable system for monitoring and analyzing physiological signals. In *BSN '06: Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks*, pages 61–64, Washington, DC, USA, 2006. IEEE Computer Society.

[OFM06b] Nuria Oliver and Fernando Flores-Mangas. Mptrain: a mobile, music and physiology-based personal trainer. In *MobileHCI '06: Proceedings of the 8th conference on Human-computer interaction with mobile devices and services*, pages 21–28, New York, NY, USA, 2006. ACM.

[OMG] Omg model driven architecture (mda). http://www.omg.org/mda.

[Örg09] Ugur Örgün. *Design and Evaluation of User-Interfaces for Mobile Applications Development*. Diploma thesis, media informatics, ludwig maximilians university, 2009.

[OT08] Phillip Olla and Joseph Tan. Designing a m-health framework for conceptualizing mobile health systems. pages 1–24. Idea Group Inc (IGI), 2008. Chapter 1 inbook.

[Ous98] John Ousterhout. Scripting: Higher-level programming for the 21st century. volume 31, pages 23–30, 1998. IEEE Computer.

[Ovi] Nokia ovi store. http://store.ovi.com. accessed 30 august 2010.

[Pal] Palm developer center, project ares. http://ares.palm.com/ares/about.html. accessed 04 january 2010.

[Pat99] Fabio Paterno. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, 1999.

[Pet95] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, 1995.

[PKCD02] Pengkai Pan, Carly Kastner, David Crow, and Glorianna Davenport. M-studio: an authoring application for context-aware multimedia, 2002. 641082 351-354.

[PVH07] Andreas Pleuss, Arnd Vitzthum, and Heinrich Hussmann. Integrating heterogeneous tools into model-centric development of interactive applications. *Model Driven Engineering Languages and Systems*, 4735/2007:241 – 255, 2007.

[QtC09] Qt creator white paper. http://qt.nokia.com/files/pdf/qt-creator-1.3-whitepaper. accessed 04 january 2010., 2009.

[QtM] Qt for mobile platforms. http://qt.nokia.com/products/qt-for-mobile-platforms. accessed 04 january 2010.

[RC02] Mary Beth Rosson and John M. Carroll. *Usability engineering: scenario-based development of human-computer interaction*. Morgan Kaufmann Publishers Inc., 2002.

[Roe08] Till Roenneberg. A worldwide experimental platform, proposal for an erc advanced grant, 2008.

[San] Sana website. http://www.sanamobile.org/about.html. accessed 12 april 2010.

[SK97] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *Computer*, 30:110–111, 04 1997.

[Som04] Ian Sommerville. *Software Engineering 7*. Pearson Education Limited, 7th edition, 2004.

[Sta] International Organization for Standardization. Iso tr 18529: Human-centered lifecycle process descriptions.

[Sta99] International Organization for Standardization. Iso 13407: Human-centred design processes for interactive systems, 1999.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[Taf09] Max Tafelmayer. *Mobia Modeler: An Adaptable Mobile Application Modeler for Non-Expert Users*. Diploma thesis, media informatics, ludwig maximilians university, 2009.

[Tea] WeP Software Team. Worldwide experimental platform (wep): Vision/scope document for release 1.0.

[Tid06] Jennifer Tidwell. *Designing Interfaces:Patterns for Effective Interaction Design*. O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2006. book.

[Tol04] Juha-Pekka Tolvanen. Metaedit+: domain-specific modeling for full code generation demonstrated [gpce], 2004. 1028686 39-40.

[VA05] Maria Virvou and Eythimios Alepis. Mobile educational features in authoring tools for personalised tutoring. *Computers & Education*, 44(1):53–68, 2005. doi: DOI: 10.1016/j.compedu.2003.12.020.

[vHVF09] Robert van Herk, Janneke Verhaegh, and Willem F.J. Fontijn. Espranto sdk: an adaptive programming environment for tangible applications. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 849–858, New York, NY, USA, 2009. ACM.

[WeP] The wep project. http://thewep.org.

Acknowledgments

DAAD Deutscher Akademischer Austausch Dienst German Academic Exchange Service
UNIVERSITY OF THE PHILIPPINES - MANILA NATIONAL TELEHEALTH CENTER
theWeP
Alina
Dominikus
Anita
Doris
Alex W
Raphael
Franziska
Rainer
Bettina
Sebastian

First of all I would like to thank **Prof. Dr. Heinrich Hußmann** for giving me the opportunity to do this research by trusting in my abilities and taking me in as his PhD student. I am truly grateful for all the guidance and wisdom he has imparted to me.

It would have been impossible for me to conduct this research without the help of the **Deutscher Akademisher Austauscht Dienst (DAAD).** I am truly grateful for the financial assistance and for making my stay in Germany as comfortable as possible. I would especially like to thank **Susanne Kammueller** and **Elke Burbach** from the DAAD, and **Davina Dy** from the German Embassy in Manila, who were always ready to help whenever I had any scholarship-related inquiries.

Thank you **Dr. Eduardo Mendoza** for also serving as a mentor to me. Because of you, I was able to get in contact with people who made it possible for me to do my research. You have helped so many aspiring scientists and researchers, and I am truly grateful to be one of them.

Thank you **Prof. Dr. Gabriele Taentzer** for serving as my second reviewer. Thank you for taking the time and interest in my work, and all for the valuable advice and input that made this thesis possible.

I would also like to thank **Prof. Dr. Till Roenneberg** and the rest of the **WeP team** for welcoming me into their group and for taking the time to answer my surveys and willingness to participate in my interviews. Your input has been a very valuable part of my research.

I would also like to thank **Dr. Alvin Marcelo** and the rest of the **National Telehealth Center team** for their valuable inputs and feedback with regards to the Mobia framework. Thank you for giving me a valuable part of your time and for sharing your expertise.

Thank you to **Yaxi Chen** for her friendship and support as we go hand-in-hand in our journey to achieving our dream of having a PhD!

I would also like to thank for **Prof. Dr. Andreas Butz** for being there and helping me with scholarship-matters.

Thank you to my office "room"-mates **Sara Streng, Alexander Deluca and Max Maurer** for offering me a **FuN** and **"warm"** environment. I truly feel very welcome and part of the group because of you!

Thank you to my colleagues at the **Media Informatics Group** for being so friendly and "cool". I couldn't ask for a better group to belong to.

Thank you to **Richard Atterrer** and **Andreas Pleuss** who have served as my "mini" mentors with regards to PhD life and model-driven stuff. I would also like to thank **Gregor Broll** who has patiently given his time to orient me with SMS matters and mobile-related queries.

To **Mama**, **Papa**, my brothers **Roberto** and **Jose**, and my sisters **Lourdes**, **Carmencita** and **Rowena**, thank you for all the support, words of encouragement and prayers. Although we are all apart from each other, I still feel safe because I know no matter what happens you will be there for me. I am truly blessed to have a great family and I thank God for that.

Thanks to all the **Pinoymunchkins** (for those who are stillin Munich and for those who are now scattered around the Globe), who has served as a family for us here in Munich.

I would also like to thank my students **Max Tafelmayer, Ugur Örgün and Jenny Forrai**, whom I have worked with and were truly very helpful in order to make this research possible. It was really an honor working with you!

Most of all, I would like to thank my beloved and supportive husband **JOY**, for making this journey easier and more fun for me. I look forward to having more journeys in this lifetime with you as I open up the next chapter of my life.

Curriculum Vitae

Personal Information

Name	:	Florence Balagtas-Fernandez
Date of Birth	:	April 21, 1980
Place of Birth	:	Butuan City, Philippines
Nationality	:	Filipino

Education

October 2007- February 2011	PhD Candidate Department of Computer Science, Media Informatics Group Ludwig-Maximilians-Universität (LMU), Munich, Germany *Dissertation title:* Easing the Creation Process of Mobile Applications for Non-Technical Users (Model-Driven Development of Mobile Applications)
June 2003 – April 2006	Master of Science in Computer Science University of the Philippines, Diliman, Quezon City, Philippines *Thesis Title*: Connecting Pervasive Frameworks through Mediation
June 1997 – April 2001	Bachelor of Science in Computer Science (Cum Laude Graduate) University of the Philippines, Diliman, Quezon City, Philippines

Work Experience

June 2003 – May 2007	Department of Computer Science, University of the Philippines (Diliman) *Instructor (2003-2006), Assistant Professor (2006-2007)*
Sept. 2002- May 2003	Canon Information Technologies, Philippines Inc. *Software Engineer*
July 2001 – June 2002	Epson Software Development Laboratories *Software Engineer*