Inverted GUI Development for IoT with Applications in E-Health

Johnsson, Björn A

2017

Document Version:
Publisher's PDF, also known as Version of record

Inverted GUI Development for IoT with Applications in E-Health

Björn A. Johnsson

Doctoral Dissertation, 2017

Department of Computer Science
Lund University

Dissertation 55, 2017
LU-CS-DISS:2017-1

ISBN 978-91-7753-238-5 (printed)
ISBN 978-91-7753-239-2 (electronic)
ISSN 1404-1219

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Bjorn_A.Johnsson@cs.lth.se
http://cs.lth.se/bjorn_a_johnsson

Typeset using LaTeX
Illustrated using Google Drawings
Printed in Sweden by Tryckeriet i E-huset, Lund, 2017

Abstract

In the context of Internet of Things (IoT), the research of this dissertation is concerned with the development of applications for end-user devices, i.e. devices through which the end-user directly interacts with systems. The complexity of such applications is partly due to network intricacies, and partly because GUI (Graphical User Interface) development is generally complicated and time consuming. We employ a middleware framework called PalCom to manage the former, and focus our research on the problems of the latter, by expanding the scope of PalCom to also enable GUI development. In particular, the research goal is a more efficient GUI development approach that does not require program code to be written.

To enable end-users with little or no programming experience to participate in the GUI development process, we eliminate the need for programming by introducing a new development approach. We view this approach as "inverted" in that the development focus is on presenting functionality from an application model as graphical components in a GUI, rather than on retroactively attaching functionality to manually added graphical components. The inverted GUI development approach is supported in two steps. First, we design a language for describing GUIs, and implement interpreters that communicate with remotely hosted application models and render GUI descriptions as fully functional GUIs. Second, we implement a graphical editor for developing GUIs in order to make the language more accessible.

The presented solution is evaluated by its application in a number of research projects in the domain of e-health. From the GUIs developed in those projects, we conclude that the GUI language is practically viable for building full-blown, professional grade GUIs. Furthermore, the presented graphical editor is evaluated by direct comparison to a market leading product in a controlled experiment. From this, we conclude that the editor is accessible to new users, and that it can be more efficient to use than the commercial alternative.

Sammanfattning

Belysning, skrivare och vitvaror. Personvågar, TV-apparater och hemlarm. Våra hem fylls med saker som kan koppla upp sig mot Internet. I det som kallas *Sakernas Internet* kan uppkopplade enheter av olika slag knytas till varandra för att bilda användbara sammansättningar, eller *system*. Men vem ska bygga dessa system? I framtiden ser det ut att kunna bli användarna som själva skapar system för att förenkla sin vardag. Vi presenterar en lösning som gör det möjligt att skapa "appar" för dessa system genom ett grafiskt program, helt utan att behöva programmera.

För det mesta vill man att system i Sakernas Internet ska smälta in i omgivning; ett system som ska anpassa musiken baserat på vem som är hemma ska fungera automatiskt utan att användaren ska behöva lägga sig i. Ibland behöver dock systemen kontrolleras direkt av användaren, t.ex. för att tända eller släcka lampor. I sådana fall kan användarens telefon användas som en "fjärrkontroll" för systemet. I detta arbete har vi jobbat med att förenkla skapandet av grafiska användargränssnitt för appar som gör detta samspel mellan människa och system möjligt. Lösningen bygger vidare på *mellanprogramvaran* PalCom, som sedan tidigare gör det möjligt för användare att skapa egna system för Sakernas Internet.

Problemet med grafiska användargränssnitt är att de är komplicerade och tidskrävande att skapa, och ofta har inslag av programmering. Detta förhindrar "vanliga människor" från att skapa användargränssnitt till sina system. Vår lösning hanterar båda problemen genom att ta bort behovet att skriva programkod. Vi introducerar istället ett "omvänt" tillvägagångssätt för att skapa användargränssnitt, där fokus ligger på att representera tillgänglig funktionalitet som grafiska komponenter, snarare än tvärtom. Till exempel kan funktionaliteten [*Tänd Lampa, Kök*] visas som en knapp i användargränssnittet.

Det nya tillvägagångssättet stöds i grunden av ett nytt programspråk som beskriver användargränssnitt på ett generellt sätt. På så vis kan samma teknik användas för att skapa gränssnitt för olika *plattformar*, t.ex. telefoner från olika tillverkare. Vi har skapat ett antal program som kan läsa beskrivningar och visa dem som användargränssnitt. För att öka teknikens tillgänglighet skapade vi också en s.k. *grafisk editor* där användaren grafiskt skapar sina användargränssnitt, helt utan att skriva programkod. Exempelvis kan man dra [*Tänd Lampa, Kök*] från en ruta till en annan för att skapa knappen som tänder lampan.

Lösningen vi presenterar har utvärderats i ett antal forskningsprojekt inom e-hälsa, där digitala hjälpmedel används för att förbättra sjukvården. Den har t.ex. använts för att skapa användargränssnittet för den app som används inom den

avancerade hemsjukvården (ASIH) i Lund. Genom detta vet vi att lösningen är *skalbar*, d.v.s. praktiskt användbart även för större gränssnitt. Ett experiment med studenter från Lunds Tekniska Högskola (LTH) har också utförts för att utvärdera den grafiska editorn. Resultaten visar att programmet är enkelt att komma igång med, och kan vara effektivt i jämförelse med en kommersiell produkt.

Acknowledgements

As a PhD student, there have been times when I have doubted myself: “Can I finish this in time? Do I have what it takes? Am I even doing the right things?!”. When such gloomy thoughts arose I turned to my supervisor, and he never failed to bring my confidence back up. For this, and everything else he has done for me since the start of my PhD studies in July 2011, I sincerely thank Professor Boris Magnusson. I value the effortlessness of how we collaborate, and look forward to continuing working with him. I also thank my co-supervisor Professor Görel Hedin for her support during my studies and her invaluable scrutiny of the content of this dissertation.

To Gunnar Weibull, I would like to express my gratitude for the effort he put into developing the graphical editor that we present as part of this work. It is thanks to him that I was able to focus on other important aspects of the research. I also thank Mia Månsson as the first true user of the solution presented in this dissertation, and for providing valuable insights on how to improve it. She and Weibull enthusiastically applied the solution in a number of related research projects, for which I am grateful. Furthermore, I thank my colleagues at the Department of Computer Science, in particular Professor Martin Höst and Mattias Nordahl for their contributions to the research and papers we have worked on together.

From the itACiH project, I would like to thank my colleagues, in particular Rickard Törnblad and Thomas Persson. Their input to, and support for, my research has been part of the success of its application in the project. I also thank the staff at ASIH in Lund for using the application that was built with the solution we present in this dissertation. Testing in a real-world environment and working closely with the end-users has been crucial to the success of the project and this research, and I appreciate the patience of the staff with us as researchers while we were working on improving our technologies.

I thank my older brother Dan Johnsson for taking the time to proofread this dissertation. Furthermore, many years ago in my early teens, he was the one who got me interested in, and taught me about, computers. What he taught me back then resulted in the career I have today, and for this I am forever grateful to him. I also thank my loving parents, for raising me as a responsible and independent individual capable of finishing his PhD studies.

Last but certainly not least I thank my dear wife, Emma Johnsson, for her inexhaustible patience during the past six years. In my studies, I have experienced

excruciating workload peaks that required my undivided attention day and night; without the encouragement and understanding of my spouse in such times, this work would not have been possible. As a final remark, I would like to express my admiration for the endurance she has shown in taking care of our two young children during the last few month when I was busy writing this dissertation.

Björn A. Johnsson
Lund, April 2017

This work was funded by Vinnova (grants 2011-02796 and 2013-04876) in the project *IT Support for Advanced Care in the Home* (itACiH). The results from the research have benefited from being applied by our research group in projects financed by Forte (grant 2013-2101) as *Lund University Child Centered Care* (LUC3), and by Vinnova (grant 2015-00382) as *Innovative Technology for the Future's Emergency Medical Care*.

Inverted GUI Development for IoT with Applications in E-Health

Contents

Chapter 1

Introduction

In the context of Internet of Things, the work presented in this dissertation aims to make GUI development more efficient, and accessible to more types of users. The research has been published at peer-reviewed international conferences, and the contributions have been applied in a number of research projects on e-health.

1.1 Research Context

Cancer is one of the leading causes of death worldwide, accounting for 8.2 million deaths in 2012 [76]. The number of patients being diagnosed with cancer is accelerating in Sweden [16]. To manage this situation of increasing volumes, advanced home care has been identified as a possible solution. The work of this dissertation has been conducted within the project itACiH: IT Support for Advanced Care in the Home. The original vision of the project was to increase the quality-of-life for cancer patients by allowing them to be treated in their homes; it branched out to provide similar care for patients with other medical condition, e.g. renal failure. To realize the vision, a distributed system for supporting advanced home-based care was developed as part of the project.

A central aspect in the itACiH system is the home of the patient. Depending on the diagnosis of the patient, medical equipment of different types needs to be installed in their homes. For example, equipment for monitoring weight and blood pressure was needed in the case of renal failure patients. These medical devices are connected to the system, thus making them remotely available in real-time. As such, we view the itACiH system as an Internet of Things (IoT) system that consists of a network of geographically distributed sub-systems, each one hosting a web of devices. The system is built on a service-oriented middleware framework called PalCom. PalCom makes the development of IoT systems easier by providing high-level programming abstractions to low-level network constructs. In doing so, systems such as itACiH can be developed in terms of application level services that communicate across devices, possibly between different networks and even different networks types, e.g. WiFi to Bluetooth.

In working on the itACiH project, we identified a number of *end-user devices*, i.e. devices from which the end-user needs to actively interact with the system in some way, typically through a GUI (Graphical User Interface). One example of this is the tablet computers used by the mobile teams of nurses that visit the patients in their homes; they use an application to make notes about patients, take photos, etc. Another example is the home-based hub that interfaces with the medical equipment. On this device, the patient can manually enter measurements and communicate with the doctor. Developing the GUIs for these types of interactions is generally complicated and time consuming. In the context of service-oriented IoT systems, the research presented in this dissertation aims to simplify the GUI development process, thus making it more efficient and accessible to more users. In practice, the contributions of our research have been applied in itACiH to build the application used by the mobile teams of nurses. Furthermore, other members of our research group have applied the contributions in related projects on e-health.

1.2 Problem Statement

The research of this dissertation is concerned with the development of applications for end-user devices in IoT systems. Devices can be any type of hardware, from simple devices such as temperature sensors, to complex ones such as tablet computers. Furthermore, devices can be distributed across multiple networks, and communicate by sending data packages (messages). An end-user device is any device through which the end-user can directly interact with the system in some way. We see the applications of such devices as being composed of three primary segments:

Graphical user interface Includes the functionality needed to enable the end-user to graphically interact with the application.

Application logic Includes the core functionality of the application. Depending on the application domain, this can include any type of functionality, from patient data management to handling sensor readings.

Network interface Includes the functionality needed to enable the device to communicate with other devices in the system.

The complexity of these segments are of two different types: *essential* and *accidental* [19]. The complexity of the application logic is essential, i.e. it is directly related to the complexity of the problem to be solved. As such, any development technology will have little impact on the development effort needed to develop the application logic segment. Development effort, in this case, may refer to e.g. the amount of time needed for development or the number of lines of program code. Complexity that is caused by the applied development technologies is referred to as accidental. As such, accidental complexity can be reduced by

improving or replacing those technologies. We identify the complexities of the GUI and the network interface as accidental, and therefore seek to minimize the effort needed to develop each respective segment. For the network interface, the effort can be reduced by employing a middleware. A middleware simplifies development by providing high-level abstractions to low-level network tasks, e.g. establishing and managing connections. For example, we have found that by going from an ad hoc network interface solution to using the middleware framework PalCom, development time could be reduced by a factor of 10 [40]. In the context of service-oriented IoT systems, the research presented in this dissertation aims to deliver a similar impact on the development effort needed for the GUI segment.

GUI development is generally complicated and time consuming. Two reasons for this are complex development tools and the difficulty of modularization [53]. While many development tools, e.g. Android Studio, provide intuitive graphical GUI editors, program code typically has to be written in order to bridge the gap between the GUI and the application logic. This adds to the complexity of the development task. Furthermore, requiring programming knowledge limits the number of possible GUI developers. Regarding modularization, it is best practice to package application logic as modules (model) that are independent from the GUI (view). If the separation of concerns between GUI and application logic is not strictly enforced, the two segments can get intertwined. This increases the long term development effort of both.

Based on this discussion, we define the following research questions:

RQ 1.1 Is it possible to produce a more efficient approach – in terms of developer productivity – for developing GUIs in the context of service-oriented Internet of Things systems?

RQ 1.2 In the given context, can we increase the number of possible GUI developers by removing the barrier caused by the need to write program code when building GUIs?

1.3 Research Methodology

Johannesson and Perjons [38] define the goal of *empirical research* as describing, explaining and predicting the world. This must be done faithfully and with great attention to detail, and always without regard to personal interests and biases. In contrast to empirical research, Johannesson and Perjons define *design research* as going beyond describing, explaining and predicting: it aims to change and improve the world. To achieve this goal, novel artifacts are conceived and created. Furthermore, the authors state that in contrast to plain design, design research also seeks knowledge about the artifacts, their use, and their environment. The overall research methodology applied for the research in this dissertation is *design science*, a special strand of design research. Design science aims to “create innovation in

the form of ideas, models, methods and systems that support people in developing, using and maintaining IT solutions" [38]. In our work the focus lies on simplifying the process of creating GUIs, ultimately to the point where even non-programmers can engage in the activity. As such we have implemented and evaluated a number of artifacts to learn of their impact on the user. Johannesson and Perjons describe the six main activities of design science, which we outline and related to our research below.

Explicate Problem This activity is about investigating and analyzing a practical problem, where the problem should be of some general interest. The activity precisely formulates and motivates the problem by highlighting its significance in some practical setting. We have in this chapter presented our problem and why it is relevant, and formulated specific research questions. From studying the literature, we present related work in Chapter 3.

Define Requirements The explicated problem is further processed in this activity by proposing *artifacts* that could solve the problem. The artifacts are defined in terms of requirements that can be traced back to the demands from the problem statement. For our research, we started from an initial set of requirements based on our research questions and the context-of-use for our proposed artifacts (Chapter 4). Additional requirements were subsequently collected iteratively based on experience and feedback from using the artifacts in their intended practice.

Design and Develop Artifact In this activity, the explicated problem is addressed by creating one or several artifacts that satisfy the defined requirements. In developing our artifacts, related solutions in previous work have been an important resource of inspiration. Furthermore, possible solutions were discussed with other members of the research group in order to get early feedback. Conventional computer science and engineering principles were used for the implementation work. The main artifacts that resulted from this activity are discussed in Chapters 5 and 6.

Demonstrate Artifact This activity serves as a soft evaluation (proof-of-concept) to determine whether the created artifacts are practically viable. For this purpose, the artifacts are used in illustrative or real-life cases. In the latter, the activity also serves to communicate the solution to the intended users. In our research, the artifacts have been continuously used in a number of research projects (Chapter 7). Besides the practical benefits for the projects, this also served to demonstrate and assert the state of our research.

Evaluate Artifact In this activity, the artifacts are evaluated to determine to which degree they solve the explicated problem, based on the defined requirements. For our research, this activity overlaps with the previous activity, in that the research results have been evaluated by using the artifacts and products

produced with the artifacts in real-world scenarios. We also performed a controlled experiment (Chapter 8) to formally evaluate the efficiency of one of the artifacts.

Communicate Artifact Knowledge The final activity involves communicating knowledge about the artifacts, as gathered during the other activities. The purpose is to allow an audience, e.g. researchers, to assess, replicate and possibly extend the findings of the research. Much of the work presented in this dissertation has been published and presented at various international conferences and workshops.

As Johannesson and Perjons remark, these activities may appear to form a linear method with strict ordering, where each activity is completely finished before moving on to the next. This is not the case, however, as design science projects are always iterative; this is also true for our research. Except for the problem statement, which has not been changed dramatically over time, the other activities have been performed several times in varying order. Furthermore, the research projects in which our contributions have been applied were developed using *participatory design* [32]. The goal was to increase product quality by working iteratively and having the end-users participate in the design process.

1.4 Contributions

The contributions in this dissertation include a new approach for GUI development, the specification of a GUI description language, and the implementation of several tools to enable development and deployment of GUIs for IoT systems. In particular, the contributions are as follows.

The inverted GUI development approach A new approach to GUI development that focuses on presenting functionality as graphical components in a GUI rather than on attaching functionality to manually added components. We see this as an inverted way of working with GUI development. The approach is introduced in Chapter 4.

Language support The design and specification of an XML-based language for describing the GUIs of IoT systems in a platform-independent manner. The language allows for GUIs to be created without the need for writing program code by supporting the inverted approach to GUI development. The language is presented in Chapter 5 and Appendix A.

Tool support The design and implementation of a number of tools needed to support the inverted GUI development approach. The tools are described in full as part of Chapter 6, and in short below.

Graphical GUI editor A graphical editor where the user can interactively develop GUIs represented in the presented language. The tool reads target system metadata, which in practice allows the user to start by selecting the desired functionality, and thereafter get suggestions for graphical components that can represent that functionality. The tool outputs GUI descriptions that can be interpreted on any platform.

Language interpreters Interpreters for two different platforms were developed; the most mature one is for Android. These tools take GUIs described in the presented language as input, interprets these descriptions and renders the resulting GUI for the user to interact with. Both interpreters build on a common core that handles communication as defined in the GUI description.

Mobile manager/launcher A kind of virtual machine that allows functionality modules (services) to be dynamically loaded during runtime, and that can serve as a connectivity hub in mobile scenarios. The tool was critical in making the Android interpreter tool practically viable, and was needed for many of the applications that have been developed as part of this work.

The idea for the inverted GUI development approach was developed as a joint effort between Björn A. Johnsson and Boris Magnusson. Johnsson developed the language that supports the approach, partially based on requirements extracted from the end-user collaboration of the itACiH project. The graphical GUI editor was implemented by Gunnar Weibull as part of his master's thesis work [83]. The work was supervised by Johnsson, who contributed the novel workflow of the editor, as based on the inverted GUI development approach. Johnsson implemented the common core used by all language interpreters, and the interpreter renders for the two supported platforms. As part of his work on the editor, Weibull made aesthetical contributions to the Android renderer, e.g. to make graphical components more colorful. A variant of the mobile manager/launcher for desktop computers was implemented prior to this dissertation as part of previous research; the contribution of Johnsson was in adapting it for non-stationary use, i.e. on mobile devices such as smartphones and tablets.

1.5 Publications

The results from this dissertation have been published as papers for peer-reviewed international conferences and workshops. An overview of the design and implementation of the system in which our contributions have been evaluated (Chapter 7) was published as

> Björn A. Johnsson and Boris Magnusson. "Supporting Collaborative Healthcare using PalCom – The itACiH System". In: *2016 IEEE*

> *International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. Sydney, Australia: IEEE, Mar. 2016, pp. 1–6. DOI: 10.1109/PERCOMW.2016.7457141

The paper was co-authored by Björn A. Johnsson, and the presented system was partially developed by Johnsson. A high-level overview of the main themes and results of this dissertation was published as

> Björn A. Johnsson and Gunnar Weibull. "End-User Composition of Graphical User Interfaces for PalCom Systems". In: *Procedia Computer Science* 94, 2016. The 11th International Conference on Future Networks and Communications (FNC 2016), Montreal, Quebec, Canada, pp. 224–231. ISSN: 1877-0509. DOI: 10.1016/j.procs.2016.08.035

Johnsson was lead author, with technical contributions as outlined in Section 1.4. The paper won "Best Paper Award" at The 11th International Conference on Future Networks and Communications (FNC 2016). By invitation, an extended version of the paper has been submitted for a special issue journal as

> Björn A. Johnsson. "Towards End-User Composition of Graphical User Interfaces for Internet of Things". In: *"Future Generation Computer Systems"*, 2016. Submitted

For this paper, Johnsson was the sole author. Notification of acceptance is still pending. The controlled experiment presented in Chapter 8 was published in part as

> Björn A. Johnsson, Martin Höst, and Boris Magnusson. "Evaluating a GUI Development Tool for Internet of Things and Android". In: *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings*. Ed. by Pekka Abrahamsson et al. Cham: Springer International Publishing, 2016, pp. 181–197. ISBN: 978-3-319-49094-6. DOI: 10.1007/978-3-319-49094-6_12

Johnsson was lead author for the paper. Furthermore, the experiment was planned, executed and analyzed primarily by Johnsson, with input and support by co-authors Martin Höst and Boris Magnusson.

The following papers have contributions that are related to this dissertation. The abstract for a conference demonstration of PalCom was published as

> Boris Magnusson and Björn A. Johnsson. "Some Like It Hot: Automating an Electric Kettle Using PalCom". In: *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*. UbiComp '13 Adjunct. Zurich, Switzerland: ACM, 2013, pp. 63–66. ISBN: 978-1-4503-2215-7. DOI: 10.1145/2494091.2494110

It was co-authored by Johnsson. The demo system was implemented by Johnsson, and made use of contributions from this dissertation. A technical solution needed for the system in which our contributions have been evaluated was published as

> Thomas Sandholm, Boris Magnusson, and Björn A. Johnsson. "An On-Demand WebRTC and IoT Device Tunneling Service for Hospitals". In: *2014 International Conference on Future Internet of Things and Cloud*. Barcelona, Spain, Aug. 2014, pp. 53–60. DOI: 10.1109/FiCloud.2014.19

Johnsson co-authored the paper, made minor contributions to the solution, and assisted in the evaluation of the work. Another technical solution that makes use of our contributions has been accepted to be published as

> Björn A. Johnsson, Mattias Nordahl, and Boris Magnusson. "Evaluating a Dynamic Keep-Alive Messaging Strategy for Mobile Pervasive Systems". In: *Procedia Computer Science*, 2017. The 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017), Madeira, Portugal. In press

The paper was co-authored by Johnsson, who also developed and evaluated the technical contributions presented in the paper.

1.6 Dissertation Outline

The rest of the dissertation is outlined as follows.

Chapter 2: Background further describes to the context in which this research has been conducted.

Chapter 3: Related Work presents previous solutions to problems similar to the stated problem of this dissertation. We relate our contributions to this work.

Chapter 4: An Inverted Solution introduces the inverted approach to GUI development. It also gives an overview of the produced solution from a perspective of requirements and aspirations.

Chapter 5: Language Support presents and exemplifies the language that has been created as part of our research, both on a conceptual level and in greater detail. The abstract syntax of the language is presented in Appendix A.

Chapter 6: Tool Support presents the tools that were created to support the inverted GUI development approach.

Chapter 7: Applications in E-Health discusses the major research projects in which our contributions have been applied, for the purpose of evaluation. A thorough walkthrough of the most mature system is given.

Chapter 8: A Controlled Experiment presents the planning, execution, and analysis of a controlled experiment for evaluating the efficiency of one of our contributions – the graphical editor.

Chapter 9: Future Work discusses possible future directions for our research.

Chapter 10: Conclusions briefly summarizes the dissertation, and draws conclusions from the presented work.

Chapter 2

Background

Ubiquitous computing, pervasive computing and Internet of Things are three terms that, while different from a historical perspective, are nevertheless used interchangeably to refer to a world where interconnected devices fill our surroundings. To technically support this, a middleware can be considered – we present PalCom. PalCom has been used in a number of research projects, recently in itACiH. The itACiH project both motivates and provides the means to evaluate the work presented in this dissertation.

2.1 Ubiquitous Computing and Beyond

Mark Weiser is often referred to as the father of *ubiquitous computing*. In a visionary paper that Weiser wrote while working at Xerox's Palo Alto Research Center, ubiquitous computing is described as "a new way of thinking about computers, one that takes into account the human world and allows the computers themselves to vanish into the background" [87]. The concept shifts focus from ordinary desktop and laptop computers, and instead envisions a world were all devices in the environment – big and small – are connected to each other. With such large amounts of computing devices, it is not feasible for devices to demand attention from the user, or even for the user be aware of them: they have to vanish into the background. Weiser made a comparison to the amount of motors in a modern car. There are motors for starting the engine, turning the windshield wipers, etc. The user of a car does not have to be aware of, or pay special attention, when using these motors. In the same way, ubiquitous computing envisions a world where the user should not even notice when she is using a computer; they will be integrated in all aspects of life, virtually invisible to the user [86]. The technologies that support this are referred to as *calm*, in that they are proactive and avoid disturbing the user [88].

Weiser outlined three key challenges for realizing ubiquitous computing [85]:

– The need for wireless bandwidth will be substantial. Even assuming modest bandwidth requirements per devices, the sheer volume of devices will amount to a collective demand that is challenging to meet.

– Support for mobile infrastructure must be provided once it becomes norm, rather than exception, that computers migrate from network to network. Weiser mentioned that protocols such as TCP/IP need to be reworked for mobile scenarios.

– With system interactions happening on many different types of devices, user interfaces that can *migrate* from one screen to another must be supported; the users must not be limited by the capabilities of individual devices.

These challenges were at the time significant and, as time would tell, far ahead of their time in terms of any sort of resolution.

Since the time Weiser first described ubiquitous computing, many projects on the subject emerged at universities and in industry. By 2001, ubiquitous computing had also become known by the name *pervasive computing* [71]. The idea was to re-evaluate the concepts of ubiquitous computing based on roughly a decade of hardware progress; elements that were exotic in 1991 were by then becoming commercially viable, e.g. handheld computers and wireless networks. The refined vision of pervasive computing can be summarized in three precepts [12]:

– A device is not a mere container of applications that must be managed by the user. Instead, a device is a window into the computing environment.

– Applications are designed to assist the user in performing a specific task, not merely to exploit the hardware capabilities of devices.

– The computing environment is used to enhance the surroundings of the user. It should not be limited to storing data and running software.

In pervasive computing, devices are hence seen from a perspective of possibilities rather than limitations. A device is a window to external functionality, and should not be limited by device hardware. The last precept restates the original vision of ubiquitous computing well, putting focus on the user experience in the computing environment, and de-emphasizing the focus on traditional software.

Pervasive computing is closely related to, and builds upon, results of the related research fields of *distributed computing* and *mobile computing* [68, 71]. Distributed computing arose in order to bridge the gap between personal computers in local area networks. In distributed computing, system components are distributed across multiple networked computers – *nodes* – that communicate by sending network messages. By using this approach to system design, goals beyond the capabilities of any single node can be accomplished. The concepts of this field, e.g. remote communication, fault tolerance and security, are essential to pervasive computing; they are well covered by the literature, e.g. [24]. A related field is mobile computing, which was conceived when researchers had to confront the challenges of building distributed systems that include mobile devices. These challenges include [70]: mobile devices have limited computational resources, e.g.

processor speed and memory size; mobile connectivity is highly unpredictable both in terms of performance and reliability; the dependency on a finite energy source (battery) limits both hardware and software solutions on mobile devices.

In 2006, Rogers [65] suggested that it was time to move on from the vision of calm computing. She argued that while considerable research efforts had been carried out in the pursuit to realize Weiser's vision, even the most advanced and impressive endeavours had failed to provide a world of calm computing. Calm computing relies heavily on the concept of *context-aware computing*, i.e. identifying the routines of users with the purpose of using this information to assist them in some way; Rogers argued that such proactive and "smart" systems were still far off, and instead suggested the pursuit of more practical goals. In particular, she suggested a mindset where people themselves were to be smart and proactive in their everyday practices, and that technologies should enable the users to participate in the creation of the user experience.

These and similar ideas have materialized as the *Internet of Things*, or IoT. The term was first coined by Ashton [7] in 2009 in a presentation on supply chain management. Since then, the definition has been widened to be non-domain-specific [33], including areas of application such as healthcare, transportation, and home automation. IoT is now a well-covered area of research, with a number of books published on the subject, e.g. [31, 78]. With the advent of the Internet, humans beings from across the world could easily connect with each other. Miorandi et al. [50] define IoT as shifting the focus to interconnecting physical devices, or *things*. Such things can communicate both with each other and the users in order to provide a given set of services to the users. Miorandi et al. specify the following three fundamental pillars of IoT:

1. Things must be identifiable.

2. Things must be able to communicate.

3. Things must be able to interact, among themselves and with the users.

With these fundamentals and the right set of methods and tools, the user can take a proactive role in IoT, building systems that reflect their needs and desires. This stands in stark contrast to traditional ubiquitous computing, where the computing environment is supposed to figure out the needs of the user.

Decades have past since Weiser's original vision of ubiquitous computing. Today, technologies and hardware are catching up with the vision, and the number of devices in our surroundings is rapidly increasing. According to a press release by Gartner – a world leading information technology research and advisory company – 6.4 billion devices will be connected worldwide in 2016 [49]. For 2020, the forecast says 20.8 billion devices. While ubiquitous computing has not fully materialize, e.g. in terms of calm technologies, the discipline has certainly been heavily influential on modern descendants such as Internet of Thing.

We have presented three disciplines that enable devices to be connected to each other: ubiquitous computing, pervasive computing, and Internet of Things. The disciplines have been presented from an evolutionary perspective, focusing on how the original vision of Weiser has evolved over time. In many cases, however, the literature on these topics makes little or no distinction between the three disciplines. For the intents of this dissertation, the terms will be used interchangeably; the sometimes subtle differences between the disciplines have no impact on this work. The primary interest is the core concept of networks of connected devices that need to be coordinated in order to serve the user. In particular, this research focuses on how to support user-to-device interactions.

2.2 Middleware

Much like both distributed and mobile computing, pervasive computing relies heavily on *middleware* to mediate between the network kernel and the applications running on pervasive devices [68]. Middleware is any software with the purpose of supporting the development of distributed applications by managing the complexity and heterogeneity of the infrastructure of such distributed applications [21]. In doing so, a simpler and more easily manageable development environment can be provided. Another useful definition states that a middleware "facilitates the communication and coordination of application components that are potentially distributed across several networked hosts" and "provides application developers with high-level programming abstractions" [67].

We elaborate on the definition of a middleware by providing a concrete example of how middleware can support development by managing complexity. Network communication is typically implemented by using network sockets. Sockets are the entry points to applications that may be running on separate networks, and provide a two-way communication stream between the applications. On the level of the application, a protocol for encoding and decoding communication must be implemented. By using a middleware, the complexity of sockets, protocols, and other related activities would be *abstracted* so that the user could achieve the same result by e.g. performing method invocations through a high-level API. The middleware effectively hides the details of the communication protocol, and handles the establishment and maintenance of sockets, thus significantly reducing the development effort.

Schmidt [72] divides middleware into layers. The top layer interfaces with applications, and the bottom layer interacts with the operating systems and the actual hardware of devices. The four layers – from top to bottom – are:

Domain-specific services Middleware in this layer are highly specialized and target one specific domain, e.g. healthcare or online banking. High-level services for that domain are defined by the middleware. The Boeing Bold

Stroke architecture [73] is an example of a middleware that operates in this layer.

Common services In this layer, the middleware of the lower layers are enhanced by defining domain-independent services. These can be reused, e.g. to manage globally available resources, which allows developers to concentrate on application logic rather than on the low-level intricacies of recurring distribution tasks. The Interoperable Replication Logic (IRL) architecture [11] is one example of this type of middleware.

Distribution These middleware provide high-level programming abstractions, e.g. reusable APIs, that build upon the capabilities encapsulated in the lowest layer. As such, distributed applications can be developed similarly to how conventional applications are developed, e.g. by invoking operations on the objects of remote devices. Examples of middleware for this layer include CORBA [36] and Java RMI (Remote Method Invocation) [58].

Host infrastructure Middleware in this layer encapsulate native low-level functionality of operating systems and device hardware. This helps prevent many programming activities that are tedious, error-prone, and non-portable such as socket programming and concurrency programming. MetaSockets [66] is an example of a middleware in this layer.

Other similar efforts to classify middleware have been performed. In a survey by Sadjadi and McKinley [67], different middleware approaches were classified as quality-of-service-oriented, real-time, or stream-oriented. In a survey of middleware for Internet of Things by Bandyopadhyay et al. [13], a classification system based on middleware features was introduced. The features studied were device management, interoperation, platform portability, context awareness, and security and privacy issues.

We believe that with the advances in networking technologies and the increasing number of connected devices, the possibility of a fully materialized Internet of Things (IoT) appears more promising than ever. However, with such large amounts of devices – with varying hardware platforms, operating systems, and communication media – the ability to simplify development circumstances will be key to the adoption and development of IoT. As of yet, no de facto standard IoT middleware has emerged [64], despite considerable research efforts in the area. Some of the major technical challenges for such a middleware include interoperability, scalability, provision of abstractions, support for spontaneous interactions, dynamic infrastructures, multiplicity, and security and privacy [22].

The use of a middleware can be instrumental in allowing developers and users alike to create systems of devices that enhance the user experience in a pervasive computing environment. By abstracting and hiding system complexity, middleware can make application development simpler and less error-prone. We have mentioned a few available middleware, with varying degrees of practical

viability and usage. For the work presented in this dissertation we build on a middleware framework called PalCom, which we introduce next.

2.3 The PalCom Framework

We have chosen to build the solution we present in this dissertation on the foundation of PalCom, which consists of a set of network protocols and a middleware framework. The metadata in the PalCom protocols lends itself well to the purposes of the presented solution, although other frameworks could also have been considered. Furthermore, the current reference implementation of PalCom is maintained by our research group at the Department of Computer Science, Lund University.

2.3.1 Introduction

Funded by the European Commission [23], PalCom started as a project that ran between 2004 and 2007. PalCom is an acronym for *palpable computing*, a concept that was introduced in the project [4]. Through this concept, the project sought to make computing more understandable (palpable) to humans, in contrast to traditional computing paradigms which – while effective for device-to-device interactions – can be incomprehensible to humans. The vision for palpable computing can be summarized by six pairs of *palpable qualities*, as listed in Table 2.1. For each pair, the property on the left corresponds to a quality that is traditionally desirable in e.g. ubiquitous computing. The property on the right is introduced by PalCom and constitutes a contradictory quality needed to increase palpability. As an example, while *scalability* is an important system property, in palpable computing it must not come at the expense of *understandability* since this would impede humans from interacting with the system.

PalCom has been realized as a set of network protocols and a middleware framework that offers development resources such as APIs and tools. In his doctoral dissertation, Svensson Fors [79] gives a detailed presentation of PalCom. The middleware enables devices to be combined across heterogeneous networks, and the services they offer to collaborate even if they were not specifically designed

Table 2.1: Definition of *palpable quality* pairs [62]. A general cornerstone of e.g. ubiquitous computing (left) is opposed by a new focus (right) added in palpable computing.

Invisibility	$\leftrightarrow$	Visibility
Scalability	$\leftrightarrow$	Understandability
Construction	$\leftrightarrow$	De-construction
Heterogeneity	$\leftrightarrow$	Coherence
Change	$\leftrightarrow$	Stability
Sense-making	$\leftrightarrow$	User control

to do so. Hence, new functionality can be created by coordinating already existing services in new formations; once created, services in PalCom can be re-purposed in any number of solutions. PalCom supports this by offering mechanisms for discovery and routing between different network technologies, a standardized way to exchange descriptions of services, and a combination mechanism based on configurations and coordination scripts for assembling services into systems.

The original objectives of the PalCom project were ambitious and covered a wide area of research, spanning across several disciplines: computer science, interaction design, industrial design, ethnography, and sociology. As such, there were many diverging and even contradicting ideas that failed to fully materialize. From a technical perspective, PalCom has since the conclusion of the project been further developed by researchers at the Department of Computer Science, Lund University. In particular, the entire code base of the framework has been re-implemented, the protocols have been enhanced to meet new criteria, and additional tools have been developed. Currently, PalCom has a reference implementation in Java which runs on most computers. Furthermore, a partial implementation in C makes it possible to run PalCom on smaller devices, although with limited functionality [55].

PalCom has been successfully deployed in a number of research projects, for systems of varying scope and domain. It has been used in healthcare systems, to track and analyze data from patients with various diseases [1, 35]; with proprietary third-party technologies, such as *1-Wire* and *Tellstick*, to assemble systems of wired and wireless actuators and sensors [48]; and for flexible video conferencing and monitoring scenarios [26, 69]. Recent research projects which incorporate aspects of PalCom include applications for real-time control of mobile robots in the *ENGROSS* project [77] and advanced healthcare in the home as part of the itACiH project (Section 2.4).

2.3.2 Terminology

The main concepts in PalCom are illustrated by the sample system in Figure 2.1. A PalCom *device* represents any physical or simulated device in the context of the PalCom *universe*, i.e. the conceptual collective that all PalCom devices belong to. Devices provide functionality as a number of PalCom *services* that can be coordinated by PalCom *assemblies*. Assemblies can either act as the "silent drivers" of systems or provide functionality in the form of *synthesized services* by aggregating the functionality of its coordinated services. In PalCom, devices connect to the universe through a number of *media abstraction objects* (MAOs). These represent some kind of communication media, e.g. WiFi or Bluetooth. Devices can communicate with other devices that are connected to a corresponding media abstraction object. Furthermore, a device can *route* traffic between its media abstraction objects, thus enabling devices on one network to find and communicate with devices on a second network.

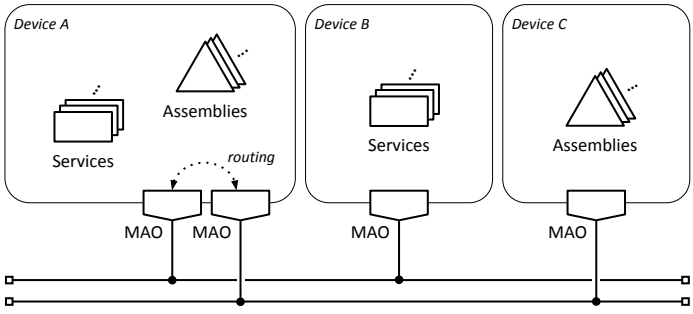


Figure 2.1: The main concepts in PalCom. Devices host functionality as services that can be combined with assemblies. Devices communicate via a number of media abstraction objects (MAOs), between which traffic can be routed.

2.3.3 Component Structure

Devices, services, commands and parameters are collectively referred to as PalCom *components*. There is a structure for these components, where devices are on the top level and parameters are at the bottom level: devices host services, services enable interaction through commands, and commands are supplemented with parameters to carry data.

A PalCom device can represent any kind of device. A device is typically a piece of hardware, like a GPS tracker or a digital camera. It can also be a virtual device simulated in software. Ideally, the PalCom device runs on the physical hardware itself, accessing the functionality of the device directly. However, this is not always possible, e.g. due to limited computing resources or locked down hardware (trade secrets). In such situations, it can be useful to run a simulated PalCom device. This is usually done be connecting the physical device to a computer on which the simulation is run. A *bridge service* that communicates with the hardware through its drivers can then serve as a connector between the PalCom environment and the system of the physical device. Hence, it is no more complicated to include existing hardware and protocols in PalCom systems than it is to include them in ad hoc systems.

A PalCom service is hosted on a PalCom device and typically represents some kind of computation or action that can be performed by the device. The service can have a direct correlation to something in the physical world, such as displaying a text on the screen of a device or moving one of its actuators. It may also be purely software-oriented, e.g. with the purpose of updating an internal counter. A *service description* defines a service in terms of the interface through which it is accessed. By having services provide their description on request, they are said to be *self-describing*. As such, PalCom services do not rely on standardization at the domain level for interoperability. Hence, one strength of PalCom is that independently

developed services that were not specifically designed to cooperate can still be combined since the service descriptions can be used for system coordination in assemblies.

The service description specifies a list of the *commands* that the service provides. A PalCom command is a message that can be sent to, or received from, a service; it is through this that the user manipulates a given service. Commands are identified with a (service level) unique ID, specify a direction that indicates whether the service should send or receive the command, and optionally list none or more *parameters*. A PalCom parameter is a construct that holds the data being passed to or from a service; a command with no parameters can be sent or received, but does not contain any actual data. Parameters are identified by a (command level) unique ID, and may contain different kinds of data, e.g. plain text or images.

2.3.4 Discovery

As illustrated in Figure 2.1, a device can have multiple media abstraction objects (MAO). Each MAO represents some media over which the device can communicate, e.g. WiFi or Bluetooth. When a service on one device wants to communicate with a service on another device, it delegates the command to be sent to the *communication layer* of the sender device. The communication layer selects a MAO on which the receiving device is reachable, and sends the command. Note how PalCom in this way becomes *network transparent*: service developers do not have to worry about how the command is going to reach the target device, only if it actually did or not. Hence, the network complexity is practically hidden from the perspective of the developers. This is possible since devices are addressed by a unique ID on the level of PalCom, rather than by MAO-specific IDs such as IP addresses. This enables devices to seamlessly migrate between networks.

To maintain this flexibility, PalCom uses a proprietary *discovery protocol* to enable devices to be aware of each other. In short, this is achieved with a *heartbeat* mechanism. At given frequencies, devices (optionally) send broadcast messages over all available MAOs to tell other devices that they are still "alive". The heartbeat also urges other devices to, in turn, identify themselves as alive. If no heartbeat has been received for a given device and an adjustable amount of time, the device is considered unreachable on that MAO. For example, if a device migrates from one network to another, it would eventually become unreachable for all devices on the old network, since the heartbeats from the device would stop arriving. However, for devices connected to both the old and the new network the device would immediately re-appear. Since the device is always addressed by its PalCom ID, the switch between networks would go unnoticed. Service descriptions and other metadata is propagated by the discovery protocol.

On devices with more than one MAO it can be meaningful to enable *routing*. Routing enables messages sent from devices on one MAO to be redistributed to affected devices on all other MAOs of the receiving device. Hence, routing can

expand the reach of devices beyond their own communication capabilities by using those of other devices. In Figure 2.1, Device A has routing enabled as illustrated by the dotted line between the two MAOs of the device. Because of this, Device B will be able to communicate with Device C and vice versa, even thought there is no direct line of communication between the two. For example, a Bluetooth gadget device might be limited to Bluetooth (MAO) for communication. To enable non-Bluetooth devices to interact with the gadget, a Bluetooth MAO could be enabled on a smartphone which has several other MAOs and routing enabled. Other devices could then reach the gadget through the smartphone.

2.3.5 Configuration and Coordination

In PalCom, the *functionality* of services is strictly separated from *configuration* and *coordination*. Typically, services do not establish connections to other services; they provide functionality as described in the service description, and do not need to know how or from where commands are arriving.

Functionality *Services should provide functionality, and only functionality.*

To create connections between services, a PalCom assembly is used. The *configuration* of services is the first of the two major parts of an assembly. In Figure 2.2, the assembly (A) is configured to create connections to two services: S_1 and S_2.

Configuration *Assemblies specify to which services connections should be established.*

The other major part of an assembly is the *coordination* of services. Coordination is specified in the form of an event-based scripting language. A script is made up of a set of *event-action list* pairs. There are a several events to choose from, e.g. the event of a command arriving from a service. For each event, one or more actions are specified. The are a number of available actions, e.g. sending a command to a service or setting the value of a variable. In the script excerpt of A in Figure 2.2,

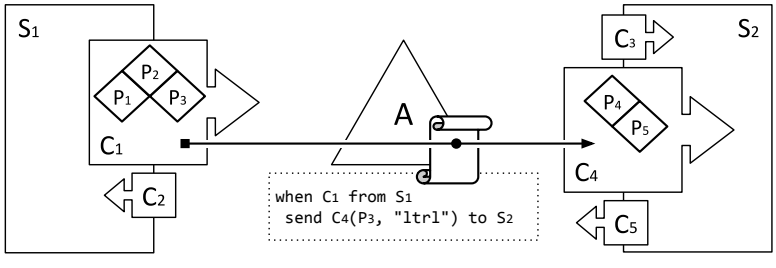


Figure 2.2: The configuration of assembly A includes services S_1 and S_2. The assembly script (dotted box) describes the coordination of commands (C_i) and parameters (P_i).

the reception of the *output* command of S_1 – which we denote $C_1{}^{\leftarrow\!\ominus}$ – is listed as an event. When the command is received, the second *input* command of S_2 will be sent, i.e. $C_4{}^{\oplus}$. Note how only one of the parameters of $C_1{}^{\leftarrow\!\ominus}$ is used; the first parameter of $C_4{}^{\oplus}$ – which we denote $C_4{::}P_4$ – gets the value of $C_1{::}P_3$ and $C_4{::}P_5$ gets the value of a string literal (*"ltrl"*).

Coordination *Assemblies script the interaction behavior of services through a set of event-action list pairs*

In most cases, assemblies are completely autonomous and silently drive systems. Another possible use of assemblies is to aggregate functionality into new, combined functionality and provide that in the form of a synthesized service. Synthesized services are specified as part of the assembly, and can be included in scripts – its own or that of other assemblies – like any other service.

2.3.6 Security

An important aspect in PalCom is system security, which can be provided on two different levels. On the device level, a standard media abstraction object that *tunnels* PalCom communication as TCP traffic between two devices can be used. These tunnels can be secured with certificates and SSL (Secure Sockets Layer) encryption, thus providing point-to-point security between devices. The server side of a tunnel specifies which certificates to accept, e.g. certificates signed by a particular CA (Certificate Authority) or those on a white-list managed by the server side. On the level of services, individual connections can be encrypted with certificates and DTLS (Datagram Transport Layer Security) [3]. This provides fine-grained control over which system resources are encrypted, and can be useful in situations where TCP-based solutions are inappropriate. Individual services that require encrypted connections specify which certificates to accept.

2.3.7 Utilities

To enable and simplify its operation, a number of helpful utility resources are provided in PalCom. We briefly present the major ones.

PalcomBrowser. The PalCom browser (PalcomBrowser) is used for browsing the PalCom universe. It is itself a PalCom device with selectable media abstraction objects; the tool list all devices reachable through these media abstraction objects. Devices can be examined, and their services can be tested in real-time by sending and viewing commands. This is enabled by the metadata in the protocol that all services implement, which includes the specification of the commands the service can send and receive, and a natural language description. By interpreting this metadata, PalcomBrowser facilitates a mode of direct interaction with services through a crude GUI. Moreover, the browser also enables the creation of assemblies

in a graphical editor. In this editing mode, PalCom components are dragged and dropped in a tree structure to model both configuration and coordination.

TheThing. TheThing is an all-purpose PalCom device that acts as a virtual machine for launching and managing system components. TheThing is used to host assemblies, an activity that is conventionally inappropriate on specialized devices. For this purpose, TheThing is typically run on a physical device that can reach all services configured by the hosted assemblies. The tool can also host services, which is useful during development to test services, for domain-independent services e.g. text formatting and unit conversions, or for bridge services that connect PalCom with third-party technologies. Furthermore, TheThing can be used to define cross service/assembly parameters, i.e. values that should be defined once for multiple uses. Different PalCom components can access these parameters from the same source. Lastly, a typical use case for TheThing is to act as a server, a centralized hub for large systems. The PalCom device of TheThing would in such cases enable routing between a number of media abstraction objects, thus allowing devices on different networks to communicate.

Resource library. The PalCom library provides a number of resources to enable the development of PalCom systems, e.g. the APIs needed for developing new services or media abstraction objects. The most common usage of the library is for developers to implement services for new functionality. In such cases, the abstract Java class `AbstractSimpleService` is inherited and extended in order to create new specialized or general-purpose services.

2.4 The itACiH Project

The contributions presented in this work have been evaluated in the context of the project itACiH: IT Support for Advanced Care in the Home. From applying our solution in the project, new real-world requirements were continuously uncovered, which helped in evaluating and improving the solution.

2.4.1 Introduction

It has long since been known that the number of patients that get diagnosed with some sort of cancer is increasing. A report from 2010 [15] states that the number of diagnoses in Sweden had been steadily increasing for the past 20 years, and that the number was projected to increase further. In order to handle this development, new organizations, processes, and systems with the purpose of streamlining the treatment of these patients are needed. Hospital-based home care (HBHC) is one such endeavour. Not surprisingly, many patients appreciate the possibility of being treated in their homes rather than at the hospital. This is fortunate, since an increased rate of treatment in the home can also decrease overall treatment

costs, an important factor to consider for maintaining current levels in quality-of-care on a larger scale. A requisite for the patients agreeing to this type of treatment is that they have confidence in the security aspects of their treatment. To make the patients feel safe at home, innovation is required in many areas, e.g. communication between patients and the medical staff, remotely surveying the states of patients, and remotely adjusting medical equipment.

In the itACiH project (2012–2015), a new platform for supporting HBHC was developed. The project started with a focus on home-based care for terminally ill cancer patients, with a long-term goal of providing support for additional treatment forms and medical conditions other than cancer. The project was a cooperation between academia and industry, and was founded by Vinnova [81] as part of their program on challenge driven innovation. In the project, academic research was combined with cross-discipline competence from e.g. medical, telecom, and security companies. An agile project plan was applied and the work was iterative, which allowed end-users such as nurses to influence how the project developed.

2.4.2 Challenges

In Sweden, the availability of HBHC is dependent on which part of the country the patient lives in. This is because it is made possible by specialized facilities, where patients are enrolled at a hospital ward, but treated at home. For palliative care of cancer patients, the treatment focus is mostly on relieving pain and maximizing quality-of-life. This kind of treatment involves medical equipment in the home, and weekly visits from the medical staff. For the treatment facilities included in the itACiH project, there was no prior system or infrastructure to support HBHC. In practice, the medical staff was carrying out their daily work with pen and paper. We outline the challenges that were identified in the project with regards to this mode of operation and the presented home-based treatment form.

Quality of Digital Records. During visits in the homes of patients, the medical staff needs to collect data in various forms and from several sources: textual notes from discussions with the patients, medical forms and checklists filled out by staff or patient, current or historical equipment readings, etc. Collecting this data on paper is problematic: after returning to the hospital ward, the staff has to manually re-enter the data into the patient record system. In practice, this means that all documents have to be scanned and committed to the records as images. Because of this, the quality of the digital records are negatively affected: scanned documents cannot be searched or used for statistical evaluations, nor can they be graphically rendered to observe trends. Having both the equipment and the staff enter data directly in digital form would open up a lot of possibilities in terms of both efficiency and quality.

Real-time Data Reports. Collecting patient data on paper has other negative effects. For equipment in the homes of patients, the staff has to collect sample

measurements manually, e.g. by reading the display on the physical device. This means additional visits if up-to-date measurements are important. Aside from the additional travel time, this is a problem because the data cannot be analyzed before the sample is brought back to the ward. Since the staff usually visits multiple patients during the same trip, they may not return to the ward until later in the day. If tests based on the collected data indicate the need for a change in the treatment, these changes are also delayed. The same problem applies for all data collected on paper. Having both equipment and staff report measurements remotely would alleviate the problem of treatment delays, providing real-time reports and increasing the accuracy of the readings.

Equipment Remote Control. Adjustment and general control of the equipment in the home of the patient is another challenge. The equipment that is used was designed for hospitals, and is typically operated by buttons on the device itself. In a hospital setting, this is unproblematic since access to the devices is readily available to the staff. Put them in the home of a patient, however, and every adjustment and observation requires an additional visit. Again, this is troublesome for the staff due to the additional travel time; for patients, the repeated disturbance can cause lower quality-of-life. In the case of cancer patients, pain relief is often provided using infusion pumps delivering anesthesia. The dosage has to be fine-tuned, which involves several cycles of adjust-observe-repeat. If each such cycle requires a visit, the logistics alone will cause lower quality-of-care. Making the home-based equipment available for remote monitoring and control would mean less interference for the patients, and more accurate and timely equipment adjustments.

Security and Sense of Safety. It is important that the patients feel safe in their homes, by having a high level of confidence in the safety of the treatment form. Otherwise, the benefits in terms of quality-of-life that come from being treated at home could be outweighed by rational or irrational fears. The patients need to know that someone who understands their situation and condition is keeping an eye on them. For patients being treated centrally at the ward, a button can be pressed to summon a nurse at any time. This sense of safety must be supported by the infrastructure, e.g. through video conferencing and monitoring solutions. If needed, staff can be dispatched to the home of the patient. However, if a situation can be managed remotely, travel time is saved, and the patient gets more timely attention. Of course, a false sense of safety has little practical value. Hence, infrastructure security to protect patient data and critical system operations is a key challenge.

2.4.3 System Architecture

One of the initial focuses of the itACiH project was to develop a working prototype for a system that would support HBHC for cancer patients. Due to the distributed nature of such a system, and the need for dynamic equipment installations in the

homes of patients, it was decided to build the system using the PalCom framework. In designing the original architecture of the itACiH system, a few key nodes (locations, personas) were identified. Figure 2.3 illustrates these, and we elaborate on the nodes as they were initially envisioned in the project.

Server The server is the primary junction point of the system. All other nodes connect to the server in some way, and either read or write (or both) data. The server stores all types of data about patients: symptom assessments, equipment readings, etc. To ensure patient confidentiality, patient data is de-coupled from patient information through the use of pseudo anonymous patient identifiers.

Home Support for communication with equipment based in the home of the patient is supported through a communication hub to which equipment connects. Measured values can thus be reported to the rest of the system. To be practical, some equipment needs to be mobile; wireless solutions are required. Non-static equipment installations and ubiquitous interaction patterns require innovations to support the home hub. Technologies for video communication solutions are employed to increase patient safety.

Mobile team The mobile teams of nurses that visit the patients in their homes are equipped with tablets, through which they can communicate information in real-time. Features include viewing patient information, or looking up which tasks are scheduled for a visit and marking those tasks as completed when appropriate. Furthermore, the staff should also be able to fill out electronic counterparts of forms and enter information about the condition the patient, making it instantly available throughout the system.

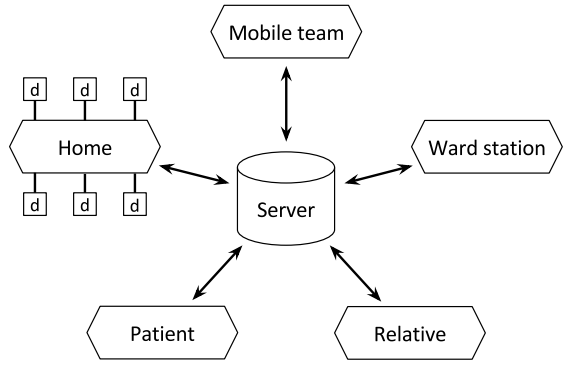


Figure 2.3: Initially proposed infrastructure for the itACiH system. All nodes access data from a centralized server. The homes of patients host a dynamic installation of devices (d).

Ward station The ward station is used by the staff for interacting with home-based equipment and for viewing patient data. This station integrates information from all the various sources in the system, e.g. equipment readings (home) and symptom assessments (mobile team). For staff members that are unable to access the ward station, and that are not equipped with a tablet, there is another option: a web application offers similar functionality to the ward station, and can be accessed using an ordinary web browser.

Patient For patients, there are a number of interesting scenarios to support, e.g. self-administered condition assessments, or registering that a prescribed drug has been consumed. The patients use a scaled down version of the tablets used by the mobile team, where certain features have been removed or altered to protect the system from having the patient make critical input mistakes.

Relative Similar to patients, relatives or friends involved in the treatment of a patient may be interested in monitoring the condition of the patient. Depending on both the needs of the patient and the relatives, this is achieved either using a further scaled down version of the mobile team tablet, or using a web application similar to the one that complements the ward station.

The evaluation of the work presented in this dissertation has been carried out in the context of the mobile team, although others have applied the solution in e.g. the home node. We elaborate on the subject in Chapter 7.

2.4.4 Development

The itACiH project was developed using participatory design [32]. Prospective users such as physicians and nurses were part of the development process, and actively participated in the design of the system. Development was iterative, i.e. the system was built and delivered in increments to the end-user, where it was used and hence evaluated by professionals in a real-life, practical environment. To safeguard against failures due to the prototype nature of early releases, the system was used alongside the normal, mostly paper-based routine during a transitional period. Tools supporting the nurses in the field were used and evaluated in the field. Home-based equipment and solutions, however, were initially set up in a special room at the ward. The purpose was to ensure patient security while still being able to evaluate the technical solutions. By using this "simulated home", should anything go wrong e.g. when remote managing equipment, the staff would be right outside the door instead of a possibly long car drive away.

The iterative design loop was turned around on a weekly basis, and the system was updated to a new and improved version every month. The development team was situated in close proximity to the medical staff, at times in the same building. This enabled good communication between the developers and the end-users. Comments could hence relatively quickly become improvements to the

system, and suggested functionality could easily be collected and feed into the design process. Furthermore, the medical staff was involved in prioritizing new functionality, which put focus on what was actually needed in the field.

Medical systems need to fulfill certain requirements in order to be cleared for active clinical use; the certification process in the itACiH project followed the iterative model. By building the system on PalCom, a component-based architecture was established. The system was structured as a collection of separate sub-systems. This supported the iterative review process by allowing for partial reviews after changes and additions to individual sub-systems. The quick certification process was key to allowing incremental releases.

2.4.5 Project Post-Mortem

The first stage (two years) of the project ended in November of 2013. The success of itACiH was apparent to the funding agency, and the project received another two years of funding. The second stage ended in December of 2015.

During the first stage of the project, much of the infrastructure and basic functionality was developed. At the end of Stage 1, the itACiH system was actively being used on a daily basis by the nurses and doctors of the unit for advanced home care (ASIH) in Lund. Much of the envisioned functionality for the server, ward station and mobile team had been implemented; the staff used two ward stations and 7 Android tablets to care for roughly 50 patients. The second stage of the project focused on refining and adding new functionality for supporting the original treatment form. Furthermore, the system was enhanced by adding initial support for additional medical conditions, e.g. renal failure. With this, the home of the patient was introduced to the infrastructure. Stage 2 had an unexpected emphasis on operational support, especially on staff scheduling. This feature proved universally useful across treatment forms and staff roles; the ward station was discontinued and replaced by the web application due to its superior accessibility. The project did not explicitly explore the challenges for including relatives and the patients themselves in the treatment process.

Since the conclusion of the project, further development and commercialization of the itACiH system has been carried out in a spin-off company with the same name [37]. As of the end of 2016, all ASIH units in Skåne County were using the operational support of itACiH – a total of over 500 active users. Other parts of the system are still in active use at unchanged levels since the end of the project, and the plan is to expand in those areas during 2017 and beyond.

2.5 Discussion

In order to support and make the Internet of Things available to the users, it must be supported by rigorous middleware. Such middleware must allow for devices

to be identifiable and able to communicate. Furthermore, devices must be able to interact among themselves and with the user. As of yet, no de facto middleware has emerged. However, considering these three IoT principles, we see PalCom as a viable option. In PalCom, devices are identifiable by unique, network transparent IDs which allows them to migrate between networks. The PalCom protocols enable device discovery and communication. In IoT, users are encouraged to undertake a proactive role by building the systems they need; the original vision for PalCom was to make computing more understandable for humans. To that end, protocol metadata and self-describing services allow users to understand and assemble existing services into useful systems. Hence, PalCom supports device-to-device interactions by keeping the human in the loop. Previously however, only crude auto-generated GUIs supported human-to-device interactions. From applying PalCom practically, we have found that the need for both types of interactions is substantial. In the system from the itACiH project, there are plenty of nodes where devices need to interact. For most of these nodes, there is also a need for human interactions in the form of GUIs. Generally, the design and implementation of GUIs is a complicated and time consuming task; with the iterative work process of e.g. itACiH, where releases are frequent, this problem is amplified. In summary, the recurring need for GUIs in the itACiH system and the crude support for human-to-device interactions in PalCom reveals a general need for a solution to efficiently create GUIs for PalCom systems.

Chapter 3

Related Work

We have identified a number of related technologies that aim to simplify application development in general and GUI development in particular. A User Interface Description Language can be used to describe GUIs for multiple platforms. Conventional graphical editors simplify GUI design with intuitive drag-and-drop mechanics. Visual programming replaces the complexity of traditional programming languages with visual alternatives, and one technology automatically generates GUIs based on domain-specific objects. We relate our work to these technologies.

3.1 User Interface Description Languages

When developing applications for multiple platforms, it can be particularly beneficial to develop the core application logic of the application separately from the GUI. In such cases, a *User Interface Description Language* (UIDL) can be used to develop the application GUI. A UIDL can be defined as a "high-level computer language for describing characteristics of interest of a UI with respect to the rest of an interactive application" [75]. As such, a UIDL provides the means of specifying the GUI independently from the target platform and the programming language that would have served to implement it otherwise. Hence, GUIs described with a UIDL are said to be *platform-independent*, i.e. they can be presented on any (supported) computing platform.

There is a wide range of UIDLs for different purposes that are more or less readily available. In a literature survey on the topic of UIDLs, Guerrero-Garcia et al. [34] classified UIDLs according to criteria such as tool support, supported platforms, availability, etc. None of the surveyed languages have been widely adopted across multiple organizations, i.e. there is no de facto standard UIDL. Instead, we perceive these languages as ad hoc solutions to related problems from different domains. Furthermore, many of the languages surveyed by Guerrero-Garcia et al. lack tool support or only support a rendering tool that interprets GUI descriptions. In many cases, as a consequence of this, GUI descriptions have to be created by manually writing program code according to the specification of the selected UIDL. For the solution presented in this dissertation, graphical tool

support was considered a priority in order to not exclude non-experts from the process of creating GUIs.

From a technical perspective, UIDLs vary widely in how they define the visual aspects of a GUI and how they define interaction behavior [34, 75]. In some languages, there is no clear distinction between the former and the latter, which we refer to as *presentation* and *functionality* respectively. In UIML [2], GUI presentation is described as a set of interface elements, and functionality is specified as mappings from those elements to external entities, i.e. logic in an application model. This is similar to the language in our solution, which is unsurprising since UIML was used as a source of inspiration for early versions of the language. Another language that shares similarities with the language we present is XIML [59]. In XIML, the presentation (*interface components*) is complemented with *attributes* and *relations*. Attributes are features or properties of components, and relations link one or several components together in order to specify functionality. A different approach is taken in for example AUIML [8], where the focus is not on the presentation itself but rather on specifying the purpose of user interactions. The designer may choose to specify precisely what to display, or allow the renderer to interpret abstract interaction representations based on the runtime context.

The solution we present in this dissertation is based on a new UIDL. This language shares similarities with many other UIDLs, both in purpose and execution. While there are technical intricacies and subtleties that differentiate our language from others, the main difference lies in our domain integration with Internet of Things via PalCom.

3.2 Conventional Graphical Editors

An *integrated development environment* (IDE) is an application that integrates features needed for developing, compiling and debugging new applications, all in the same environment. The purpose of collecting these features is to simplify the software development process, thereby shortening the development times for applications in general. In order to specifically simplify the development of the GUIs for new applications, many IDEs incorporate a *graphical editor* for creating and editing GUIs. Examples of such editors include Windows Form Designer in Visual Studio [39], WindowBuilder [89] in Eclipse, and Interface Builder in Xcode [54].

A typical graphical editor presents a “canvas” to the user on which she can compose a GUI. The user starts by selecting a graphical component from a palette adjacent to the canvas, e.g. a button or a text box. The component is then positioned in relation to other components in the GUI by dragging it from the palette and dropping it onto the canvas. By selecting (highlighting) a component that has been placed on the canvas, a list of its properties is displayed in the editor; the

user can customize the component (size, font, color, etc.) simply by entering new values into the editor. This method of working simplifies the process of creating GUIs by requiring little or no programming skills, and by providing continuous visual feedback on the final result. After partially or completely finishing the graphical design of the GUI, however, the user must define the behavior of the GUI. This is typically done by writing program code that links functionality to the manually added graphical components, so called "glue code". In practice, behavior is specified by implementing *callback methods*, i.e. methods that are invoked by graphical components when certain events are triggered. A button might for example call `onClicked()` every time it is clicked in the GUI by the end-user. In contrast to the first phase of GUI development, this second phase requires programming skills from the user. Depending on the complexity of the application, and whether application logic has been developed separately, the programming effort might be small or large in scale.

Some IDEs are platform-dependent, i.e. created applications only run on one platform. One such example is Android Studio [74]. Due to the increasing popularity of the mobile market and the many mobile platforms available, cross platform IDEs are becoming more common. Qt Creator [63] is an example of a development environment that can be used for creating applications that run on several platforms, including iOS, Android and Windows Phone. Our solution adopts the same platform-independent approach. Furthermore, many of the traditional aspects of graphical editors have influenced the solution we present in this work. As such, our solution is similar to other graphical editors in e.g. the use of drag-and-drop mechanics and the concept of canvas editing. Unlike conventional editors, however, our solution does not require program code to be written when developing GUIs.

3.3 Visual Programming

To get over the barrier caused by the complexity of programming in traditional languages such as C++ or Java, *visual programming* [47] is one possible solution. As a broad definition, visual programming refers to "any system that allows the user to specify a program in a two-(or more)-dimensional fashion" [52]. In practice, this usually translates to a graphical development environment where traditional and novel programming concepts are represented graphically, and can be created without having to write program code. One example of such an environment is Scratch [61]. Scratch seeks to enable people of all ages, backgrounds and interest to develop their own programs, even people that previously had not imagined themselves as programmers. The approach of Scratch is to represent programming statements graphically as colorful pieces of a jigsaw puzzle; these can be fitted to define logic. The PalCom tool PalcomBrowser offers a primitive form of visual programming that is used when creating the assemblies that configure and coordinate PalCom services into fully functional systems.

Some visual programming solutions focus entirely on the specification of application logic, e.g. Bloqqi [27], a data-flow language for building control systems. In others, however, visual programming is merged with the constructs of conventional graphical editors. This allows GUIs to be created graphically, without requiring program code to be written. One such tool is App Inventor [30], which offers visual drag-and-drop building blocks in the place of a text-based programming language. Using App Inventor, even inexperienced users can manage to create fully functional Android apps. Like traditional graphical editors, App Inventor offers a palette of graphical components which can be placed on a *viewer* (canvas). Unlike traditional graphical editors, however, functionality is specified with an approach similar to that of Scratch, i.e. by fitting jigsaw pieces. Recently, Google announced App Maker [51], a new "low-code" application development tool [6]. While details are scarce as of the time of this writing, App Maker is presented as a visual programming tool that requires little or no program code to be written when creating GUIs. Similar to App Inventor, App Maker appears to follow the conventional editor workflow of selecting and placing graphical components on a canvas.

Another solution based on the idea of visual programming is JavaBeans [25]. With JavaBeans, Java components – both graphical and otherwise – can be packaged so that they can be used in a graphical programming environment. One example of an environment with support for JavaBeans is NetBeans [17]. Similar to other graphical editors, GUIs are designed by selecting and placing *beans* (components) on a canvas. Functionality is then specified by creating connections between beans, a process that does not involve programming. Instead, connections are specified by selecting a source event and a target operation, and by providing connection parameters. Since application logic can be developed by programmers and packaged as beans, the result is that GUIs can be created by talented designers that do not need to be programmers themselves.

The above is a selection of solutions that are related to our own in that they aim to remove the barrier of having to write program code from the GUI creation process. Unlike ours, however, these solutions still follow the traditional workflow, albeit without having to write program code: manually add graphical components, and attach functionality to those components retroactively.

3.4 Automatic GUI Generation

The *Naked Objects* pattern introduces an object-oriented approach to GUI development. In his doctoral dissertation, Pawson [57] presents the pattern in detail. With Naked Objects, the user is presented a 1:1 graphical representation of the domain objects in a system, and can manipulate those objects by performing direct method invocations through the presented graphical components. The idea is to simplify the widely adopted model-view-controller (MVC) pattern, which

Pawson describes as being implemented as a four-layer architecture. In MVC, the business concepts of a system may appear across all four of these layers in various forms, which adds to the complexity of the system. With Naked Objects, business concepts are instead concentrated to the *domain object layer* and have one direct mapping in the *presentation layer*. In practice, the interfaces of domain objects are interpreted in order to automatically generate the GUI. For this to work properly the object-oriented concept of *behavioral completeness* must be supported by all domain objects. Behavioral completeness implies that objects must not only model the properties of their real-world counterpart, but also their behavior.

Naked Objects can be seen as primarily servicing the programmer. By applying the pattern, developers do not have to explicitly devote resources towards GUI development since those are automatically generated. Instead they can focus on the application logic of the domain objects. The Naked Objects pattern can be defined by the following three principles [60]:

1. The domain objects should encapsulate all application logic.
2. The GUI must represent all aspect of the domain objects.
3. The GUI must be created automatically from domain objects.

For the solution presented in this dissertation, we agree that application logic should be managed separately from the GUI. Furthermore, our solution is also based on generating GUIs based on domain objects (PalCom services). However, we do not apply a completely automated approach and do not enforce a strict 1:1 relation between domain object and representation. In those two aspects, Naked Objects are closer to the approach for GUI generation applied in the PalCom tool PalcomBrowser.

The pattern of Naked Objects is currently being used in commercial solutions such as Apache Isis [5] and BlueJ [14]. Research efforts toward applying Naked Objects for the development of Android applications include JustBusiness [29], a framework for atomically generating the GUIs of Android applications.

3.5 Discussion

The solution we present in this dissertation shares many similarities with the types of technologies that have been presented here. We base our solution on a UIDL which differs from other languages mainly in its domain integration with Internet of Things via PalCom. The graphical editor we present for this language borrows concepts from conventional graphical editors, e.g. the use of drag-and-drop mechanics. For our solution, however, we wanted to increase the user base by not requiring explicit programming, which is the case when specifying GUI functionality in conventional graphical editors. Visual programming is proposed by some as a solution to this problem. However, for such a technology to be

considered for our purposes, it would have to produce platform-independent GUIs. Furthermore, we believe that the final step in the conventional GUI development approach, i.e. retroactively attaching functionality to manually added graphical component, is too open ended for the average casual user, even when visual programming is applied. Our solution instead introduces a novel approach to GUI functionality specification that is based on interpreting metadata in PalCom to selectively generate GUI components. This approach, unlike completely automatic techniques such as Naked Objects, keeps the human in the loop which allows for more specific GUI designs.

Chapter 4

An Inverted Solution

The recurring need for GUIs in the itACiH system reveals a general need for a cost effective method for creating the GUIs of end-user devices. From a practical perspective, in systems such as these where most nodes require a GUI of some sort, keeping development costs down becomes a priority. To enhance GUI development productivity, and to avoid excluding non-programmers from the process, we introduce an "inverted" approach to GUI development.

4.1 Requirements

For the system that was built during the itACiH project, the PalCom framework was chosen as supportive middleware due to its extensive support for such ubiquitous systems. The lack of support for creating GUIs in the PalCom framework and the recurring need for GUIs in the itACiH systems motivates the creation of a uniform solution for developing GUIs specifically for PalCom systems. This new solution needs to relate to the already established *user roles* of PalCom. Table 4.1 shows the original roles that PalCom users could undertake prior to this work. The primary distinction between the roles is that they require different levels of technical expertise from the user. *Assemblers* create assemblies using PalcomBrowser – a standard PalCom tool. No other software, and hence no other technical skills, are required. The user experience it completely graphical, demanding no coding knowledge from the user. This non-programming quality is intrinsic of the role of the assembler. *Developers* implement new PalCom services using for example a third-party IDE and APIs in the PalCom library. For this role, users are expected

Table 4.1: Original user roles in PalCom. Developers create services by writing program code; assemblers create (assemble) systems of such services in a graphical tool.

Role	Creates	Uses
Developer	Service	PalCom library, 3rd-party IDE
Assembler	Assembly	PalcomBrowser

to have the technical expertise required to install required software (compilers, libraries, etc.) and to program in a given programming language – currently Java. This is a considerable step-up in the requirement of technical skills compared to that of assemblers.

In order to address the problem statement from Chapter 1, we introduce a solution that provides support for creating GUIs targeted at PalCom systems. The solution is introduced at the level of the assembler role. This choice of skill level has practical implications; we formulate the following requirements:

Requirement 4.1 The solution must be platform-independent. PalCom systems can be deployed on multiple platforms, a feature that must be retained for the GUIs of such systems. Users at the assembler level, however, cannot be expected to have the necessary knowledge to adapt the GUIs for multiple platforms. Instead, the new solution must allow the user to create platform-independent GUIs that can be deployed on any supported platform.

Requirement 4.2 The solution must allow for *codeless* functionality specification. In PalCom, functionality is provided through services that are produced by developers. When assembling PalCom systems, assemblers can incorporate these services into their system. This allows them to define the functionality of entire systems without having to write any program code. The same concept must also hold true when creating GUIs for such systems.

For our solution, we propose a hybrid between a platform-independent UIDL and a graphical editor. By harnessing the platform-independent nature of a UIDL and combining that with the user friendliness of a graphical editor, a solution that satisfies both Requirement 4.1 and Requirement 4.2 can be realized. Specifically, we introduce a graphical editor for a PalCom-specific UIDL targeted at users on the level of the assembler role, i.e. the same type of users that create assemblies.

4.2 The Inverted GUI Development Approach

To avoid violating Requirement 4.2 we introduce a novel approach to how functionality is specified for GUIs. In essence, the conventional GUI development approach is turned $180°$. We refer to this as the *inverted approach to GUI development*, and define it as follows:

> *The inverted approach focuses on presenting functionality as graphical components in a GUI, rather than on retroactively attaching functionality to manually added graphical components.*

We illustrate the inverted approach from a perspective of workflow and architecture.

4.2.1 Conventional Approach

The typical workflow of a conventional graphical editor can be summarized as follows:

1. Choose graphical component.
2. Place on canvas.
3. Add code to attach functionality.

The user chooses graphical components from a palette or menu. These are dragged onto the canvas and dropped to specify placement in relation to other graphical components. For the final step, behavior is specified by attaching the component to functionality. In conventional editors, this is done by implementing callback methods. In the best of cases, i.e. if the principle of separation of concerns is followed, the implementation may consist solely of glue code that connects the component to the application model. In the worst case scenario, extensive coding – possibly including application logic – may be required in order to define the behavior of the component. The conventional development workflow for the former case is illustrated in Figure 4.1. For graphical editors which include visual programming elements, the final step may not require explicit program code to be written, i.e. conventional programming. However, a manual and retroactive intervention that may be more or less extensive is still required.

In Figure 4.2, a typical system architecture when developing with the conventional approach is illustrated. The GUIs that are produced with conventional editors are deployed on devices other than the development device. However, both the GUI and the attached application model must run on the same device, and in the same runtime process. This implies that all functionality of the GUI must be locally available; any access of remote functionality must be embedded in the locally deployed application model. Since the application model shares the same runtime process as the GUI, it will be unavailable whenever the application of the GUI is not running. This effectively makes the model useless to other system components, e.g. an alternate GUI.

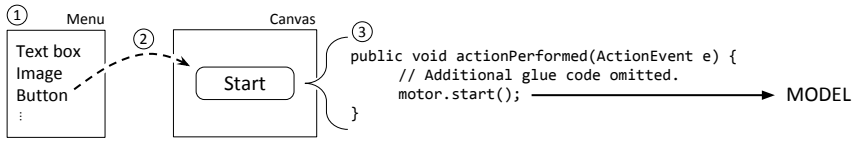


Figure 4.1: Workflow for conventional development approach. The user manually adds graphical components ("Start" button) to the GUI. Functionality from the model is then attached by writing more or less extensive glue code.

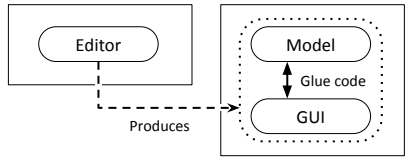


Figure 4.2: Architecture for conventional development approach. The editor produces a GUI that is run on a separate device. The GUI shares runtime process (dotted box) with the application model, which it accesses though the user-specified glue code.

4.2.2 Inverted Approach

By applying the inverted development approach, the workflow instead becomes:

1. Choose functionality component.
2. Place on canvas.
3. Select representation and customize link.

The user starts by dragging and dropping functionality components – rather than graphical components – onto the canvas. In our case, the metadata available in PalCom makes this possible. It also aids in the third stop by providing information for suggesting suitable graphical components that can represent the chosen functionality. Once representation has been selected, the user may customize the *link* that is created between the functionality (PalCom) component and the graphical component. Otherwise, default behavior is inferred. The inverted development workflow in the context of PalCom is illustrated in Figure 4.3.

Architecture-wise, the inverted approach differs from the conventional approach in a few key ways. The architecture is illustrated in Figure 4.4. Since the focus of the approach is on presenting functionality, a strict separation of concerns between GUI and model can be ensured. Because of this, the two elements can run on separate devices, thus making distributed functionality feasible. The native

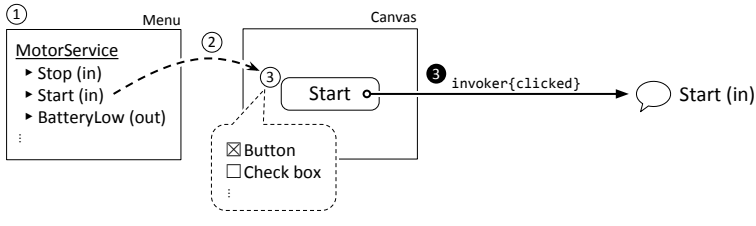


Figure 4.3: Workflow for inverted development approach. The user selects the desired functionality ("Start" command). From suggestions based on metadata, a graphical component ("Start" button) is selected and *linked* to the original source of functionality.

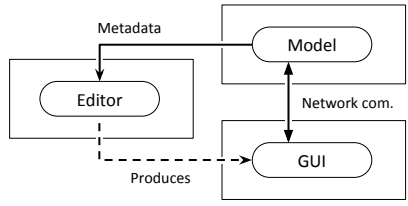


Figure 4.4: Architecture for inverted development approach. Through strict separation of concerns, the GUI and model are run independently on separate devices. Since the model is always-on, the editor can read it in real-time during the design process.

distribution features of PalCom makes this a powerful characteristic. By distributing the model, it becomes possible for one GUI to access multiple models, and for multiple GUIs to access the same model. Furthermore, since the model is always available (independently of the GUI), its functionality can be integrated into the editor as described above.

4.3 Aspirations

The inverted GUI development approach has been realized as a PalCom-centric UIDL (PML, Chapter 5) and a set of tools (Chapter 6), most notably the Graphical PML Editor (GPE). With the way in which GPE supports the construction of GUIs, the user starts from PalCom component such as services and their commands, and represent those graphically as e.g. buttons and text boxes. In other words, the user starts by identifying what she wants to do, after which she gets suggestions for graphical components that can represent that functionality.

By having GPE implement the inverted development approach, users can develop entire GUIs without having to write any program code. This is in line with our goal of introducing GUI development at the level of the PalCom assembler role, and Requirement 4.2. As such, GPE is used by the same type of users that previously used PalcomBrowser to create PalCom system by assembling services. Ultimately, it is the ambition that both these tools are to be operable by non-programmers. To that end, none of the tools require conventional programming to take place. However, even though no program code needs to be written manually, we cannot reject the possibility that programming skills are actually required. This is because neither tool has been formally evaluated in the demographic of non-programmers. It is however, as mentioned, the aspiration for this group to adopt the tools.

GPE is built on top of the platform-independent language PML. A body of PML code that properly *describes* a GUI is referred to as a PML description; GPE is used to produce such descriptions, which can be loaded and interpreted as GUIs on any supported platform. Hence, the presented solution satisfies Requirement 4.1.

The ambition with this requirement is for the user to be able to use one and the same solution when creating GUIs for different platforms. For assemblers, this is a necessity. For developers, it avoids the overhead costs of having to learn the tools of additional platforms. However, it is not necessarily the ambition that a description is to be portable between platforms without modification. In other words, interpreted GUIs need not look identical on all platform; there may be minor or major variations in how graphical component look and feel, how they are structured and arranged, etc.

By default, PML offers a suite of standard graphical components such as buttons, text and image boxes, and drop-down lists. It is the ambition that this selection is to be sufficient to cover most needs for basic GUI development and quick prototyping. It is however not feasible for such a suite to cover all possible future needs of the user in terms of graphical expressiveness. Furthermore, a balance must be struck between expressiveness and complexity in order to allow for a diverse array of GUIs to be created while at the same time not alienating the technically less capable non-programmer. To address this problem, we extend the PalCom library to enable users on the level of the developer role to create custom PML *parts*, i.e. graphical components. This construct is for advanced users, and is completely analogous with how services are created and loaded during runtime. It allows developers with specialized needs to create completely new or extend existing parts as needed. With custom parts, the ambition is for PML to be a viable option also for larger development projects.

Based on the discussions above, we extend the original user roles of PalCom from Table 4.1 as per Table 4.2.

Table 4.2: Extended user roles in PalCom. In addition to services, developers also create specialized graphical components (parts). These and a set of standard parts are used by assemblers to create GUIs (descriptions) for single services or assembled systems.

Role	Creates	Uses
Developer	Service Custom PML part	PalCom library, 3rd-party IDE
Assembler	Assembly PML description	PalcomBrowser Graphical PML Editor

Chapter 5

Language Support

PalCom User Interface Description Language, PUIML, or more commonly PML is a platform-independent, XML-based language that is used to describe GUIs and their interaction with the underlying application model in terms of PalCom services. PML supports the inverted approach to GUI development by specifying behavior exclusively in terms of links between components. Once created, PML descriptions can be interpreted on specific platforms to form fully functional GUIs.

5.1 Overview

A PML description – formally *PalCom User Interface Description* – is a body of PML code that *describes* all aspects of the GUI that will be created by interpreting it. Similar to PalCom assemblies, every PML description must specify which PalCom components, i.e. devices, services, commands and parameters, are part of its configuration. PalCom components are represented by a particular type of *PML component*. Through their class and properties, these identify and manage all communication for the PalCom component they represent. The graphical content of the GUI is also represented in terms of PML components. These have their own type, and belong to one of several classes that specify the kind of graphical component being represented, e.g. a text box or a button. PML provides a suite of classes for standard graphical components to select from, and the assortment can easily be expanded through an API in the PalCom library. The PML components represent graphical components in a generic, platform-independent manner. Visual details can be customized by adjusting the style properties of the PML component. The structure of GUIs is defined by *nesting* compatible components within each other, e.g. a button inside a window.

As discussed in Chapter 4, the inverted GUI development approach focuses on allowing the user to select functionality to be presented as graphical components in the GUI, rather than on choosing graphical components and attaching functionality to those e.g. with glue code. To support this novel approach, PML defines behavior by assigning *behavior properties* for components. These properties *link* one component of the description to another, giving the link a specific *role*. Figure 5.1

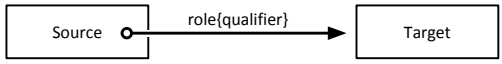


Figure 5.1: Generic *link* between two PML components. Links have a given role and can be further specialized by a qualifier. Specific examples of links are illustrated in Figure 5.2.

illustrates a generic link between two PML components. Whether two components can be linked, as well as what roles are acceptable for the link depends on the classes of the linked components. The four roles for links are "invoker", "reactor", "provider" and "viewer".

The role of invoker is used when one PML component (*source*) is to trigger, or *invoke*, an *action* at another component (*target*). The role is complemented with a value – the *qualifier* – that indicates what *event* must occur at the source component to *activate* the link. The action that is triggered at the target component depends on its class specification. For example, consider an invoker link between the PML components of a graphical button and a PalCom command (input), with the qualifier as the "clicked" event of the button. This link would cause the command to be sent to its corresponding PalCom service whenever the button is clicked by the end-user. The role of reactor is the reactive counterpart of the invoker role: an action (qualifier) at the source component is triggered by an event at the target component.

The role of provider is used when a property value of one PML component is to be assigned as the new value for a property of another component. The qualifier of this role indicates which property of the source is to be used for the assignment. The action that activates the link, and hence the assignment to be performed, depends on the class of the source component. Similarly, the property value that is set for the target component depends on its class specification. Consider as an example a provider link from the PML component of a text box, to the PML component of a PalCom parameter. The qualifier is the text property of the text box. This link would cause the value property of the parameter to get assigned the text in a text box, as entered by the end-user. In this example, this would happen every time the end-user edits the text in the GUI. The reactive counterpart of the provider role is the role of viewer: the value of a property (qualifier) at the source component is assigned the value of a property at the target component.

In many cases, an invoker link from Component A to Component B can be replaced by a reactor link from Component B to Component A. The same applies for provider links, which can be replaced by viewer links in the reversed direction. The main motivation for having two pairs of practically interchangeable roles is that PML was originally designed with an emphasis on graphical components: links should originate from graphical components and target PalCom components. There is, however, no limitation as to which types of PML components can act as source and target for links. For example, a component representing an output command of a service could be linked as invoker for a component representing

an input command of another service. In this case, the link would activate upon receiving the first command, thus sending the second command to its service. A second reason for the four roles is that in the current implementation, the link qualifier applies to the source component only. As such, the direction of a link affects which behavioral aspects can be specified. For future revisions of PML, we are considering making links undirected, i.e. with no specific source/target. By also introducing qualifiers on both sides of links, only two roles would be necessary in the considered implementation.

5.2 The Echo Example

As a simple but concrete example of how PML uses links to specify GUI behavior, Figure 5.2 illustrates both the graphics and the links of a GUI for echoing messages. The GUI is described using PML, and was interpreted on an Huawei Nexus 6P running Android 6 using the interpreter tool to be presented in Chapter 6. In the example, the GUI connects to a PalCom service that echoes all messages sent to it back to the sender – EchoService. The service has one input command, Voice⊕, and one output command, Echo⊖. Both commands carry a parameter containing

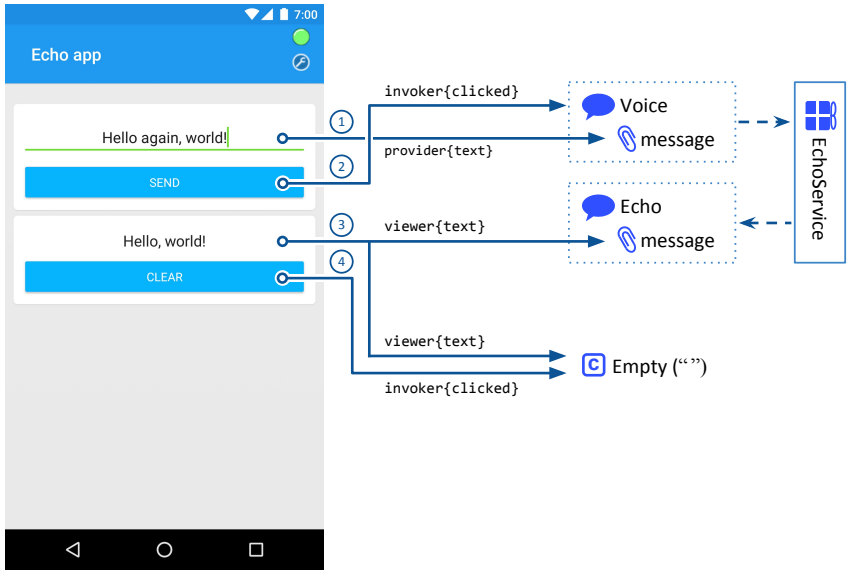


Figure 5.2: Screenshot of a PML description interpreted on a Huawei Nexus 6P. Links between graphical components, a command (speech bubble), parameters (paper clips) and a constant (C) model the behavior of the GUI, i.e. displaying echoed messages.

the echo message: Voice::message and Echo::message, respectively. In the GUI, the user can type in the uppermost text box ("Hello again, world!"). By pressing the button labeled "SEND", Voice$^{\oplus}$ with the content of the text box as its message is sent to the echo service. When received, EchoService responds by sending Echo$^{\leftarrow\ominus}$ containing the same message back to the GUI. Upon reception, the content of the message parameter is displayed to the user ("Hello, world!"). Pressing the button labeled "CLEAR" clears this message, resetting it to empty.

All PML links needed to model this behavior are illustrated in Figure 5.2. The text box provides its text value to Voice::message whenever the text changes (1). The send button is linked as invoker to Voice$^{\oplus}$ (2), i.e. when the user clicks the button the command (and its parameter) is sent to the echo service. The text label is linked as viewer for both Echo::message and an empty constant (3). Upon receiving Echo$^{\leftarrow\ominus}$, the text value of the label is updated with the value of the parameter. Since the clear button invokes the constant (4), whenever it is clicked the empty value is propagated to its viewers, hence clearing the echo message label.

5.3 Structure

The code in PML descriptions is formatted according to the standard XML specification. A structured and readable format such as XML was favored so that power users could make advanced edits to PML descriptions without the need for special tools. PML descriptions are structured by division into three component blocks – *universe*, *structure*, *logic* – and three property blocks – *style*, *discovery*, *behavior*.

Universe The *units* of a description are defined in the universe block. Units represent the PalCom components (devices, services, etc.) that can be used in the description. The universe block is supplemented by the discovery and behavior blocks. The **discovery** block contains properties that identify the PalCom components of units. The **behavior** block contains properties that specify the behavior of units.

Structure In the structure block, the *parts* of a description are defined. Parts represent the graphical components that make up the GUI that will be presented to the end-user. The block is supplemented by the style and behavior block. The **style** block contains properties that affect the visual characteristics of parts. Like for units, the behavior block contains properties that specify the behavior of parts.

Logic *Facts* are defined in the logic block of the description. Facts represent internal components (constants, variables) that can be used in the description. The logic block is supplemented by the behavior block.

```
<puiml>
  <universe><unit/><unit/>...</universe>
  <discovery><property/><property/>...</discovery>

  <structure><part/><part/>...</structure>
  <style><property/><property/>...</style>

  <logic><fact/><fact/>...</logic>

  <behavior><property/><property/>...</behavior>
</puiml>
```

Listing 5.1: Overview of a PML description, illustrating the six blocks for PML components (universe, structure, and logic) and properties (discovery, style, and behavior).

Listing 5.1 illustrates the structure of a PML description, including the six blocks with components and properties. Note that this is not proper PML code, and serves only to illustrate the structure.

In PML, all components are represented as XML elements with a corresponding tag name of “unit”, “part” or “fact”. These elements have two XML attributes: “id”, a string that uniquely identifies the component in the context of the description; and “class”, which specifies what type of component is being represented. Depending on the class of the component, its element may contain other nested components. These concepts are illustrated in Listing 5.2, where a service (class attribute, line 3) is specified as being hosted on “device1” (ID attribute, line 2).

Discovery, style and behavior properties are represented as XML elements with the tag name “property”. All property elements must have the attribute “name” set to specify which property is being set for the PML component in question. A value is assigned to the selected property by editing the inner text of the property element. In order to successfully link a property to an existing component, one of the attributes “unit-name”, “part-name” or “fact-name” must also be set, depending on what sort of component the property should apply to. The attribute value should match the value of the ID attribute of a previously defined PML component. On line 10 in Listing 5.3, the property ”p:id” of a unit “device2” (unit-name attribut) is set to “C:30e8f8a2”. As an alternative way of structuring PML descriptions, one can choose to specify the properties of a component directly in its definition, rather than in one of the *global* properties blocks. To do this, a *local* properties block is created inside the element of the component. In Listing 5.3, the property “p:id” of

```
1  <universe>
2    <unit id="device1" class="P:Device">
3      <unit id="service1" class="P:Service"/>
4    </unit>
5  </universe>
```

Listing 5.2: PML components are defined by the tag name, e.g. “unit”, and the class attribute, and can be nested for logical structuring, e.g. a service on a device.

```
1   <universe>
2     <unit id="device1" class="P:Device">
3       <discovery>
4         <property name="p:id">C:30e8f8a2</property>
5       </discovery>
6     </unit>
7     <unit id="device2" class="P:Device"/>
8   </universe>
9   <discovery>
10    <property unit-name="device2" name="p:id">C:30e8f8a2</property>
11  </discovery>
```

Listing 5.3: PML properties can be specified both locally, i.e. inside an component (line 4), and globally by referencing an existing component (unit-name, line 10).

"device1" is set on line 4. Note how contrary to for global properties, no unit/part/fact-name attribute has to be specified for local properties; the property element is nested inside the component element, implicitly specifying which component the property belongs to. Both ways of structuring descriptions yield the same end result. The difference between the two alternatives is hence mostly of interest for power users making direct description edits. Defining properties locally can be useful for smaller descriptions in order to have all information pertaining to a PML component in one place. For larger descriptions, defining properties globally will allow for a quicker overview of the description structure, while keeping the details provided by the properties in one collective place.

5.4 Details

A more fine-grained insight into PML is provided by expanding on the purpose and content of each of the six blocks of a PML description. The abstract syntax of PML is presented as part of Appendix A. To practically demonstrate PML, the code that makes up the echo example in Section 5.2 is presented for each respective description block. The description for the example was originally created using the graphical editor to be presented in Chapter 6; the presented code has been edited for readability purposes.

5.4.1 Universe Block

In the universe block, the units of the PML description are declared. Units represent PalCom components. Hence, there is only four classes that units can belong to; Table 5.1 outlines these classes. Units can contain other units – units can be *nested*. How units can be nested depends on their classes. The universe block can contain an unbound number of units of class "P:Device". Units of class "P:Device" can in turn only contain units of class "P:Service", since a device cannot contain e.g. a command without it belonging to a service first. Analogously, "P:Service" can

Table 5.1: Valid classes for PML units, i.e. PalCom components.

P:Device	PalCom device.
P:Service	PalCom service.
P:Command	PalCom command.
P:Param	PalCom parameter.

```
<universe>
  <unit class="P:Device" id="devHost">
    <unit class="P:Service" id="svcEcho">
      <unit class="P:Command" id="cmdVoice">
        <unit class="P:Param" id="prmVoiceMsg"/>
      </unit>
      <unit class="P:Command" id="cmdEcho">
        <unit class="P:Param" id="prmEchoMsg"/>
      </unit>
    </unit>
  </unit>
</universe>
```

Listing 5.4: Universe block of the echo example, Section 5.2.

only nest “P:Command”, which in turn only nests “P:Param”. It is important to note that how units are nested affects how they will be located in the PalCom universe. For example, when trying to locate a PalCom service, the discovery properties of the service-unit (“P:Service”) will be used. However, these alone are not enough to identify the service. The discovery properties of the parent device-unit (“P:Device”) must be used to specify the PalCom device on which the service is hosted. This means that the same service-unit could be used to describe different services just by being nested in a different device-units.

Listing 5.4 shows the universe block of the PML description for the echo example. Note that this code only specifies the relation between the PalCom components: one device hosting one service, which has two commands with one parameter each. Each unit is given a unique internal ID through the ID attribute. The identity of the PalCom components is specified in the discovery block by referencing these IDs.

5.4.2 Discovery Block

In the discovery block, the discovery properties of the PML description are declared. Discovery properties apply to the units declared in the universe block, and are mainly used to identify the corresponding PalCom components of units, i.e. how they can be located in the PalCom universe. The class of the unit determines which discovery properties are available to set. Some of these properties are required and must be specified, while others are optional and may be left out for an implicit default value.

```
1  <discovery>
2    <property unit-name="devHost" name="p:id">C:30e8f8a2</property>
3    <property unit-name="svcEcho" name="p:required">true</property>
4    <property unit-name="svcEcho" name="p:instance">1</property>
5    <property unit-name="svcEcho" name="p:cdid">X:EchoDevice</property>
6    <property unit-name="svcEcho" name="p:cn">BAJ1</property>
7    <property unit-name="svcEcho" name="p:udid">X:EchoDevice</property>
8    <property unit-name="svcEcho" name="p:un">BAJ1</property>
9    <property unit-name="cmdVoice" name="p:id">Voice</property>
10   <property unit-name="cmdVoice" name="p:direction">in</property>
11   <property unit-name="prmVoiceMsg" name="p:id">message</property>
12   <property unit-name="cmdEcho" name="p:id">Echo</property>
13   <property unit-name="cmdEcho" name="p:direction">out</property>
14   <property unit-name="prmEchoMsg" name="p:id">message</property>
15 </discovery>
```

Listing 5.5: Discovery block of the echo example, Section 5.2.

The discovery block of the PML description for the echo example is presented in Listing 5.5. Note how each property refers to a previously declared unit through its internal ID (unit-name attribute). Line 2 specifies the identifier used for discovering the device "devHost". Lines 3–8 specify properties for discovering the service "svcEcho"; lines 6–8 are used to support versioning of services. The commands Voice⊖ and Echo⊖ and their respective message parameters are identified on lines 9–11 and 12–14 respectively.

5.4.3 Structure Block

In the structure block, the parts of the PML description are declared. Parts represent the graphical components of the GUI that will be presented to the end-user, and can belong to one of several classes from the standard suite of PML; Table 5.2 outlines these. Additionally, parts can represent custom graphical components, as developed using the PalCom library. In such cases, the Java canonical class name of the custom part is used for the class attribute.

The structure block can and must hold one, and only one, part of class "G:Application". This class represents the interpreting application inside the description, and must be the first part defined in any description. Similar to units, parts can be nested, i.e. parts can contain other parts. The nesting of parts is used to logically structure the GUI; the way in which they may be nested depends on the classes of the parts. In principle, only parts of container classes (containers) can contain other parts. As an example, a button (part of class "G:Button") can logically reside within a window (part of container class "G:Window"), but a button can not logically reside within another button.

It is through the logical nesting of parts, in conjunction with the layout-related style properties provided for container parts, that the GUI gets its structure. Layout properties define how graphical components should be laid out within the container

Table 5.2: Valid classes for PML parts, i.e. graphical components.

G:Application	○	Logical component that represents the interpreting application internally.
G:Window	●	Top-level, self-contained, structural component. May contain other components.
G:Area	●	Structural component. Primary purpose is to contain and layout other components.
G:Tabbed	●	Structural component. Contained components are represented as tabs, and displayed after selection.
G:RadioGroup	○	Logical component that groups radio buttons.
G:RadioButton		Clickable, two-state component. One instance per group can be checked.
G:Label		Component for displaying simple text.
G:Button		Clickable component.
G:CheckBox		Clickable, two-state component. Multiple instances can be checked.
G:TextField		Component for text input.
G:Image		Component for displaying images.
G:TextArea		Component for displaying text (advanced).
G:NumberSlider		Component with horizontally slidable handle. Selects numeric value in given range.
G:DropDownList		Compact (expandable) component for item selection from a list.
G:SystemNotification		System wide notification. Visible even when the application is not.
G:YesNoDialog		Dialog box that may block the rest of the application until confirmed/dismissed.
G:QuickNote		Temporarily visible text message.
G:Sound		Plays a notification sound.

○ Limited container ● Full container

```
1  <structure>
2    <part class="G:Application" id="appEcho">
3      <part class="G:Window" id="winMain">
4        <part class="G:Area" id="areaTop">
5          <part class="G:TextField" id="txtMessage"/>
6          <part class="G:Button" id="btnVoice"/>
7        </part>
8        <part class="G:Area" id="areaBottom">
9          <part class="G:Label" id="lblMessage"/>
10         <part class="G:Button" id="btnClear"/>
11       </part>
12     </part>
13   </part>
14 </structure>
```

Listing 5.6: Structure block of the echo example, Section 5.2.

component, e.g. from left to right or from top to bottom. To properly structure the graphical components of a PML GUI, there are three things to consider:

1. The nesting of a part decides in which container its corresponding graphical component should be placed.

2. The order in which the parts are declared decides the order in which the corresponding graphical components will be laid out within the container.

3. The specified layout-related style properties of the container part decide the formation in which the graphical components should be laid out.

Of these, Items 1 and 2 are specified in the structure block whereas Item 3 is specified in the style block. Listing 5.6 shows the structure block of the PML description for the echo example. Note how the content matches the structure of the GUI in Figure 5.2: a window (line 3) divided into two separate sections, one containing a text box and a button (lines 4–7), and another containing a label and a button (lines 8–11).

5.4.4 Style Block

In the style block, the style properties of the PML description are declared. Style properties apply to the parts declared in the structure block. They are used to specify what the corresponding graphical component of a part should look like in the resulting GUI; the properties that are available to set depend on the class of the part. For example, container parts provide several layout-related properties, such as component arrangement and padding, while text input/output parts offer several font-related properties, such as font size and color. Some style properties are compulsory and must be given a value, while others are optional and may be omitted for an implicit default value.

The style block of the PML description that describes the echo example is presented in Listing 5.7. Note how the layout formation of sub-parts is specified

```
1  <style>
2    <property part-name="winMain" name="g:title">Echo app</property>
3    <property part-name="winMain" name="g:layout">linear</property>
4    <property part-name="winMain" name="g:layout-gap">0,25</property>
5    <property part-name="winMain" name="g:layout-orientation">vertical</
         property>
6    <property part-name="areaTop" name="g:layout-gap">0,25</property>
7    <property part-name="areaTop" name="g:layout-orientation">vertical</
         property>
8    <property part-name="areaTop" name="g:border">raised</property>
9    <property part-name="txtMessage" name="g:align-h">center</property>
10   <property part-name="btnVoice" name="g:size">-1,-1</property>
11   <property part-name="btnVoice" name="g:text">Send</property>
12   <property part-name="areaBottom" name="g:layout-gap">0,25</property>
13   <property part-name="areaBottom" name="g:layout-orientation">vertical</
         property>
14   <property part-name="areaBottom" name="g:border">raised</property>
15   <property part-name="lblMessage" name="g:font-size">18</property>
16   <property part-name="lblMessage" name="g:align-h">center</property>
17   <property part-name="btnClear" name="g:size">-1,-1</property>
18   <property part-name="btnClear" name="g:text">Clear</property>
19 </style>
```

Listing 5.7: Style block of the echo example, Section 5.2.

for container parts. On lines 3–5, for example, the main window of the GUI (“winMain”) is set to arrange sub-parts in a *linear* and *horizontal* fashion, i.e. one after another from top to bottom, with a given amount of padding between components (“$0, 25$”).

5.4.5 Logic Block

In the logic block, the internal components of the PML description are declared. These represent constants and variables, and can for example be used as default texts in GUIs or to model states. In PML, these components are referred to as facts. Facts can belong two one of two classes, as as outlined in Table 5.3

Listing 5.8 shows the logic block of the PML description for the echo example; a single constant is declared.

Table 5.3: Valid classes for PML facts, i.e. internal components.

G:Constant	Holds a constant value.
G:Variable	Holds a value that can be changed.

```
1  <logic>
2    <fact class="G:Constant" id="cstClear"/>
3  </logic>
```

Listing 5.8: Logic block of the echo example, Section 5.2.

5.4.6 Behavior Block

In the behavior block, the behavior properties of the PML description are declared. These are primarily used to specify the behavior of the GUI. As introduced in Section 5.1, this is done by creating links between pairs of PML components, and assigning a role to that link. A secondary purpose of the behavior block is to declare properties that are behavior-related but that are not links. One example of this is the property that specifies the delimiter that text areas use for separating lines in data to be displayed in the GUI. When used this way, behavior properties are similar in function to discovery and style properties. Behavior properties apply to the units, parts and facts declared in the universe, structure and logic block respectively. What behavior properties are available to set depends on the class of the component; some properties are required, while others are optional.

When used for specifying links, behavior properties are more complex than other property types. The roles of links are specified by using one of the property names “p:invoker”, “p:reactor”, “p:provider” and “p:viewer”. The qualifier is specified by assigning a value to an attribute that depends on the role of the link, e.g. “event” for invoker links. Furthermore, a component can be the source for multiple links with the same role and qualifier. In such cases the attribute “order” can be set, starting at 0, to specify the order in which the links will be activated. For example, if a button has invoker links to multiple PalCom commands for the “clicked” qualifier, the command for the link with the lowest ordering value will be sent first when the button is clicked.

The behavior block of the PML description that describes the echo example is presented in Listing 5.9. The specified links (lines 2–7) are illustrated in Figure 5.2, with the exception of the link on line 2 which has been omitted in the figure. In this example, all link sources are graphical components and hence specified by the “part-name” attribute. Link targets are specified as the inner text of the elements that represent the links. We elaborate on the each individual behavior property:

```
1  <behavior>
2    <property part-name="appEcho" name="p:invoker" event="loaded">winMain</
         property>
3    <property part-name="txtMessage" name="p:provider" get="text">prmVoiceMsg
         </property>
4    <property part-name="btnVoice" name="p:invoker" event="clicked">cmdVoice<
         /property>
5    <property part-name="lblMessage" name="p:viewer" set="text" order="0">
         prmEchoMsg</property>
6    <property part-name="lblMessage" name="p:viewer" set="text" order="1">
         cstClear</property>
7    <property part-name="btnClear" name="p:invoker" event="clicked">cstClear<
         /property>
8    <property fact-name="cstClear" name="p:value"></property>
9  </behavior>
```

Listing 5.9: Behavior block of the echo example, Section 5.2.

– On line 2, the interpreting application is instructed to open the main window ("winMain") when it is loaded.

– On line 3, the text box ("txtMessage") is set to provide its text value to the message parameter of Voice$^{\oplus}$. When the command is sent, its parameter will already be set to the latest value entered by the end-user in the text box.

– On line 4, the send button ("btnVoice") is linked as invoker for Voice$^{\oplus}$; whenever the button is clicked, the command is sent to the echo service.

– On line 5, the text label ("lblMessage") is linked as viewer to the message parameter of Echo$^{\ominus}$ to capture the response from the service. Whenever the command is received, the value of its parameter is be displayed.

– On line 6, the same label is also set as viewer for the constant "cstClear".

– On line 7, the clear button ("btnClear") is linked as invoker to the constant . This means that whenever the clear button is clicked, the value of "cstClear" is propagated to its viewers, hence clearing the echo message label.

– On line 8 the value of the constant ("cstClear") is set to empty.

Note that the inner text (value) of the element that represents the constant (line 8) is empty. Furthermore, note how the ordering mechanism is used to accommodate the two viewer links of text label on lines 5–6.

5.5 Interpretation

A *PalCom User Interface Description Interpreter* – also simply referred to as a PML interpreter – is an application that is used to produce fully functional GUIs from PML descriptions. The interpreters provide the application fundamentals needed to display the GUIs that are created from *interpreting* descriptions. As such, each interpreter is specific to one platform, i.e. an interpreter must be developed for each platform that is to support PML. To minimize such development efforts, we have factored out the platform-independent parts as illustrated in Figure 5.3.

The interpretation process starts by loading an existing PML description. In the interpreter application, the input file is processed by the *front end*. The front end parses the content of the file and evaluates if it constitutes valid PML code. If successful, the front end outputs an *intermediate representation* (IR) of the parsed description. This representation models the PML parts, units, and facts, and all properties – including links. Furthermore, the IR functions as a common core for all interpreters: during interpretation startup, the IR is responsible for initializing all PML components; during runtime, it handles all application logic for the GUI, most notably the logic needed to activate links. In particular, when PML units are initialized, connections to the specified PalCom services are established; during

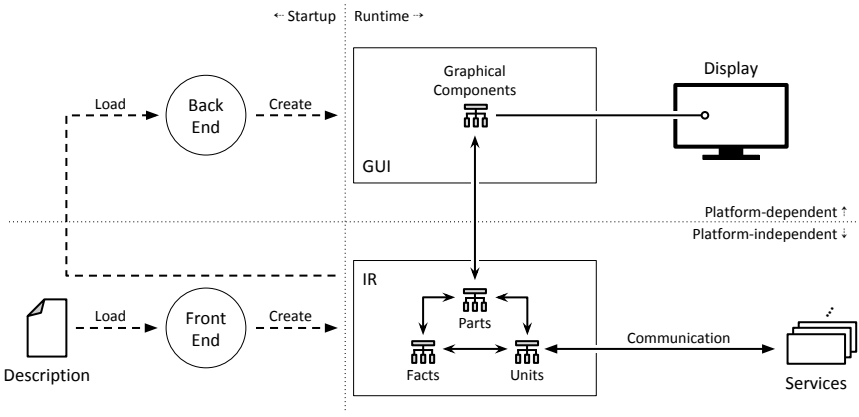


Figure 5.3: Interpretation of a PML description: the front end creates an intermediate representation (IR) of the described GUI; the back end creates the actual GUI based on this IR. Data structures in the IR handle the communication with the specified PalCom services.

runtime, the IR handles all communication with the services, e.g. by sending PalCom commands. Both the front end and the IR have been included in the PalCom library; they can be reused for interpreters on all applicable platforms.

In the final step of the interpretation startup process the *back end* renders the GUI. This happens when the intermediate representation initializes the PML parts of the represented description. Unlike units and facts, which can be initialized internally by the representation, parts are platform-dependent and must be initialized by the back end. This is achieved by passing the IR to the back end via callback methods – one for each available part class. The expanded PalCom library provides an API for this purpose, since a platform-dependent back end must be created for each platform that is to support PML. The back end uses the information in the IR to render its PML parts as graphical components, e.g. text boxes or buttons. The end result is the GUI that is presented to the end-user on a display.

During runtime, the GUI acts as a mere view for the intermediate representation. All input by the user is forwarded to the IR so that the proper links can be activated. Similarly, when the IR is updated with new data, e.g. by receiving a command, it determines which links are to be activated and propagates the data to the GUI accordingly. It is this strict separation of concerns that allows the back end to be the only part that must be re-implemented for each new platform.

5.6 Discussion

In order to support the inverted GUI development approach, PML specifies GUI behavior exclusively in terms of links between components in the intermediate

representations of GUIs. Since all application logic used in GUIs described with PML is modularized as PalCom services, these intermediate representations can contain both an abstraction of the application model (services) and an abstraction of the GUI itself. This makes it possible to link individual components from both abstraction for the purpose of defining GUI behavior, thus completely eliminating the need to write program code in traditional languages such as C++ or Java.

Since PalCom systems can be deployed on a number of platforms, PML was designed to describe GUIs in a platform-independent manner. PML parts represent generic graphical components that are applicable on most platforms; for specialized needs, custom parts can be implemented by developers. PML parts are nested to create structure in the GUI, and style properties can be set to adjust graphical aspects such as layout and component size. As part of this work, interpreters for two platforms have been implemented; we introduce these in the next chapter. To add support for additional platforms, only a platform-dependent back end that renders the GUI needs to be developed, since all other parts of the interpreter are platform-independent and included in the PalCom library.

Chapter 6

Tool Support

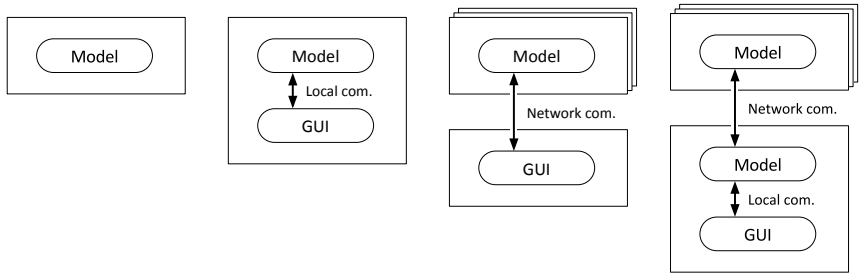
The Graphical PML Editor is a tool that is used to create and edit PML descriptions. The editor implements the inverted approach to GUI development, exclusively through graphical interactions, i.e. no program code is needed. GUIs created with the tool can be deployed in systems with a number of different architectures. Support for these is provided by the classic PalCom tool TheThing, and by contributions from this work: TheAndroidThing, SwingPUIDI, and AndroidPUIDI.

6.1 Deployment Architectures

The focus of the inverted approach to GUI development is on presenting functionality as graphical components. Because of this, a strict separation of concerns between GUI (view) and model can be ensured. For the approach to be possible, however, all functionality must be available in the form of PalCom services. How these services are hosted and accessed by GUIs depends on the system architecture. We discuss four such architectures in order of increasing complexity, as illustrated from left to right in Figure 6.1.

6.1.1 Autonomous Model

TheThing is a launcher and manager of resources in PalCom, and has been used extensively as an architectural tool for the systems we have built. Ideally, functionality should be hosted on the hardware of the device to which it pertains. Although efforts towards making this goal possible have been made, e.g. in the form a lightweight C implementation of PalCom [55], the more common case is to connect the device to some less restricting hardware, e.g. a computer. The intermediate device communicates with the target hardware, and presents it functionality in the form of PalCom services. Furthermore, many types of services do not map directly to any physical device. For example, a service for printing documents maps well to a physical printer device; a service for arithmetic operations, however, could logically be hosted on any device, preferably one that is widely accessible. For these reasons, TheThing is an indispensable feature in most PalCom systems. It



(a) Autonomous model. **(b)** Local model. **(c)** Distr. model(s). **(d)** Locally augmented distr. model(s).

Figure 6.1: Common system architectures in PalCom. Tools include TheThing/ TheAndroidThing for models, and SwingPUIDI/AndroidPUIDI for GUIs.

effectively acts as the application model. In the simplest of architectures (Figure 6.1a), a system could consist of no more than a single TheThing performing some isolated task, e.g. collecting and storing sensor readings.

6.1.2 Local Model

For the work of this dissertation, the focus is on semi-autonomous and non-autonomous systems that require input from the user, to some varying degree. Such input is provided by means of a GUI. A simple architecture that allows for this is illustrated in Figure 6.1b. Like before, the model is managed by TheThing, with functionality in the form of services. The GUI is built and described with the solution of this work. As described in Chapter 5, fully functional GUIs are produced from PML descriptions through interpretation. We introduce **SwingPUIDI**, an interpreter tool that produces GUIs based on the widget toolkit Swing (Java). GUIs produced with this tool run on any hardware that accommodates TheThing. In this architecture, the GUI and the model run on the same hardware and communicate using local (internal) interfaces. They are, however, separate processes and hence the model can prevail without the presence of the GUI. This is essential for systems where user input is only needed sporadically, and the system functions autonomously for the majority of its uptime. From a perspective of tool maturity, SwingPUIDI was a proof-of-concept, and is currently not maintained.

6.1.3 Distributed Models

A more versatile architecture, that takes greater advantage of the distributed capabilities of PalCom, is illustrated in Figure 6.1c: a GUI connects to one or several distributed models over some arbitrary network technology or technologies. The notion of accessing distributed functionality is in itself of interest, allowing for all

manner of systems to be remote controlled via a GUI. Furthermore, the distribution aspect of this architecture opens up for multiple models to be accessed by the same GUI. This implies that the functionality of any number of devices can be aggregated into a single graphical access point for the user. Another implication of the distribution aspect is that the models and GUI run not only in separate processes, but also on separate hardware. This allows for the GUI to be run on hardware that is of a different type from that of the models.

In many cases, e.g. in the itACiH system, we have found that mobile devices offer a preferable method of interaction for these types of systems. The flexibility and portability of such devices prompted us to develop a second PML interpreter: **AndroidPUIDI**. This tool, as its name suggests, runs on devices powered by the mobile operating system Android. Android is currently the primary target development platform and hence, unlike SwingPUIDI, AndroidPUIDI is actively maintained and improved upon.

6.1.4 Locally Augmented Distributed Models

In stationary systems, where the GUI device infrequently or never migrates between different networks, the plain distributed models architecture is satisfactory. Disturbances such as occasional network outages can adequately be handled by the built-in mechanisms of PalCom, e.g. reliable transfer of messages. In systems where loss-of-contact is the rule rather than the exception, however, such mechanisms in themselves are not sufficient.

We have experienced these types of problems as particularly intrusive on mobile devices, where long offline period are common, the battery can run out at any moment, etc. To ensure dependable operation under such conditions, an architecture where the distributed models are augmented by a local (on-device) model can be applied. The architecture is illustrated in Figure 6.1d. The local model can mirror selected application critical functionality from the remote models, thus making it available during offline sessions. Furthermore, the local model can enhance the remote functionality to compensate for the mobile shortcomings, e.g. buffering GUI input locally on stable storage. Naturally, the local model can also provide independent functionality, e.g. giving access to native mobile features such as messaging, phone calls, and GPS positioning.

The results of the work presented in this dissertation have primarily been applied to mobile scenarios. To add support for this architecture on Android, we introduce **TheAndroidThing**. This tool is the mobile counterpart of TheThing, and features the same functionality adapted for non-stationary use.

6.1.5 Discussion

TheThing and TheAndroidThing have features (services) for remote management of services, assemblies, etc. Systems can hence be updated and main-

tained remotely, which is essential in large, distributed systems. SwingPUIDI and AndroidPUIDI have similar features for updating and installing GUI descriptions and custom PML parts (graphical components). In mobile systems, these features for both model and GUI tools can be particularly invaluable since mobile devices – from experience – can be difficult to collect for a coordinated system update.

6.2 The Graphical PML Editor

The focus when developing the Graphical PML Editor (GPE) was to support the inverted GUI development approach. By eliminating the need to write program code, the goal of inviting more types of users to participate in the GUI design process could be achieved. An early version of GPE was developed as part of a master's thesis work [83]. The user's manual for the version of GPE that we report on here has been published as [84].

6.2.1 Editor Overview

The editor is divided into four distinct sections through which the user interacts with the tool; Figure 6.2 shows an overview of the editor with the four sections distinguishable in the positions outlined below.

Network pane (left) Lists all available functionality, i.e. PalCom devices, services, commands, and parameters.

Toolbar (top) Contains input options for interacting with the editor, e.g. opening and saving descriptions, or changing component properties.

Canvas (center) Shows an approximate preview of what the GUI will look like once interpreted.

Application bar (bottom) Lists application-wide components such as constants and notifications.

Following the inverted development approach, the user starts from functionality and gets suggestions for graphical components that can represent that functionality. This functionality is accessed in the form of PalCom components. In GPE, the user can examine all devices that are available on the connected network(s) using the **network pane**. The network pane presents functionality in tree structures with devices at the top (root). Individual components in trees can be expanded and collapsed (orange triangles), thus respectively revealing and hiding underlying components: expanding a device reveals its services, expanding a service reveals its commands, etc.

The user chooses functionality by clicking on and dragging PalCom components from the network pane onto the **canvas**. The canvas shows an approximate

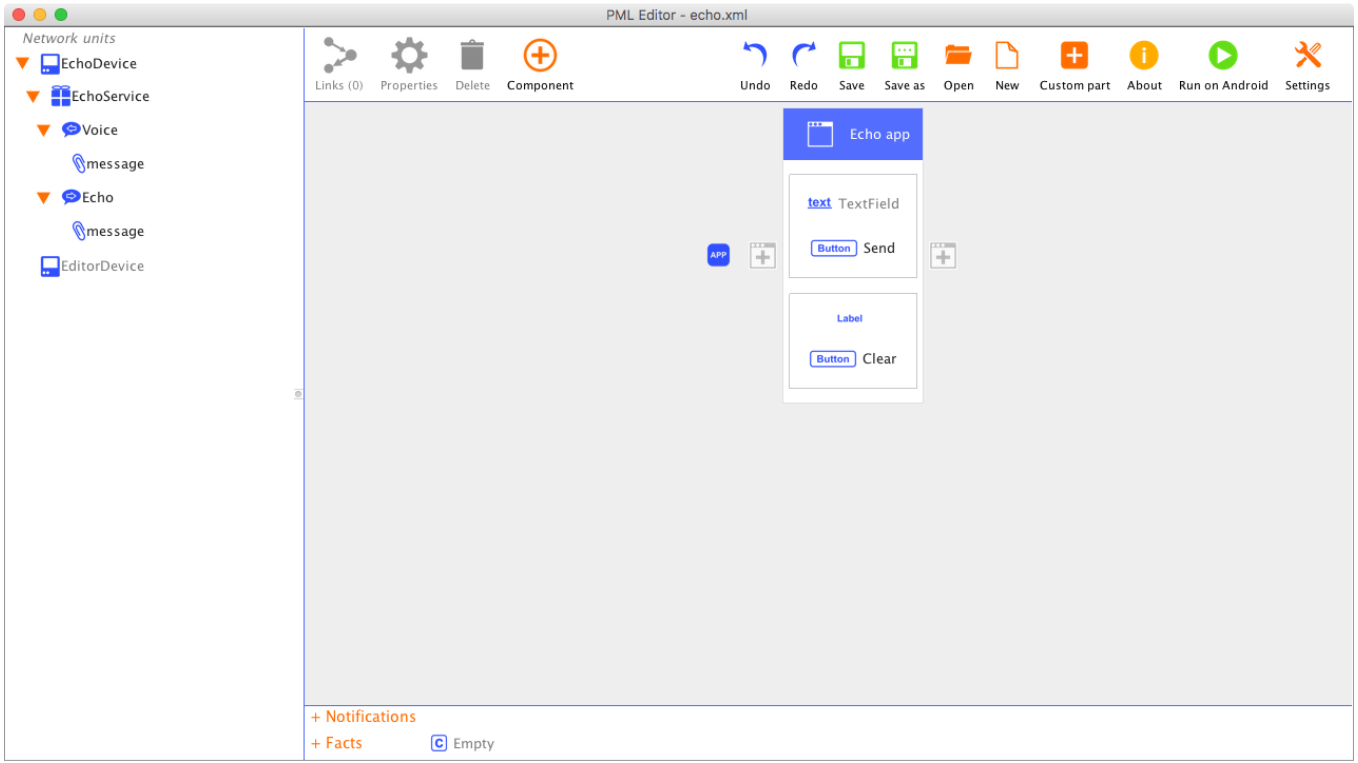


Figure 6.2: Creating the example GUI from Chapter 5 in the Graphical PML Editor. The user drags functionality (PalCom) components from the *network pane* (left) to the *canvas* (center) – the editor suggests graphical components that can present that functionality, and possible links.

preview of what the GUI will look like once its description has been interpreted. Technically, the canvas contains a number of *tiles* for the different windows of the GUI, and controls for adding additional windows. On each tile, graphical components are laid out. Upon dragging any component to a window tile, the editor presents a number of graphical components that can present that functionality, and a number of possible links. Descriptions can be run on targets devices during development, directly from the editor.

The **toolbar** contains a number of input options for interacting with the editor. It is split in two sides:

– If any component is selected in the editor, the left side of the toolbar provides options to review and edit links and general component properties, e.g. text, size, and color. Otherwise, the option to add new components is presented.

– The right side provides general options: open and save descriptions, undo and redo changes, and changing editor settings. Custom PML parts (graphical components) can also be added to expand the standard suite of PML. Furthermore, descriptions can be run on a connected Android device.

Components that do not belong to any specific window, i.e. notifications and PML facts, are managed in the **application bar**. The application bar lists these application-wide components, and provides controls for adding additional ones. Notifications can be used to notify users of events, or prompt them to provide immediate input. Facts are mainly used to update components with the values of specific constants and variables.

6.2.2 Architecture

The Graphical PML Editor is implemented as a specialized PML interpreter; the architecture is illustrated in Figure 6.3. When the user opens a description from the toolbar in the editor, the specified file is passed to the interpreter, where it is

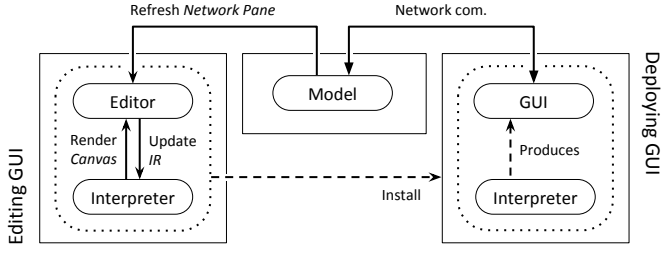


Figure 6.3: Architecture of GPE. An interpreter (left) is used to render the canvas after each change to the intermediate representation (IR). The application model is read to keep the network pane synchronized. Descriptions can be installed remotely on target devices.

processed as described in Chapter 5. In short, the front end parses the content of the file and outputs an (initial) intermediate representation (IR) of the description. The IR is used in the back end to render the PML parts of the description as graphical components on the canvas of the editor. Furthermore, the IR is read to populate the other sections of the editor with component from the description. With a description fully loaded, the user can make changes to the represented GUI through the various features of the editor. Changes are automatically propagated to the IR, which is instantly re-rendered to refresh the content of the canvas. To keep the content of the network pane up-to-date, the editor directly accesses the application model (read only). When the user saves her progress, the IR is exported to a destination file. This file can be installed remotely on a target device from the toolbar of the editor. Once installed, the interpreter of the remote device can use the description to produce a GUI connected to the application model.

6.2.3 Example Development Walkthrough

To convey how the inverted GUI development approach affects the development process in GPE, we present a walkthrough of the steps needed to create the GUI of the echo example that was introduced in Chapter 5. The walkthrough focuses on workflow, and highlights the primary features of GPE.

The functionality and data flow of the GUI resulting from this walkthrough is depicted in Figure 6.4. The GUI connects to EchoService, a "Hello, world!" service with two commands: Voice$^{\oplus}$ and Echo$^{\ominus}$. The user can enter a message (1) that is used as value for the message parameter of Voice$^{\oplus}$ (2). Clicking a button (3) invokes the command (4), thus sending it to EchoService (5). When EchoService receives Voice$^{\oplus}$, it echoes the specified message (parameter) by sending Echo$^{\ominus}$ back to the sender (6). The content of Echo::message is displayed in a text label (7). To clear previous messages, a second button can be clicked (8). This invokes a constant (9), the empty value of which is displayed in the text label (10), thus clearing any previously echoed messages.

We start building the echo GUI from a blank description, i.e. an empty canvas. The inverted GUI development approach, as implemented in GPE, is illustrated in Figure 6.5. The message parameter of Voice$^{\oplus}$ is located in the network pane and dragged to the main window tile on the canvas (Figure 6.5a). Upon dropping Voice::message a dialog appears (Figure 6.5b), presenting suggestions for suitable graphical components to select from. These suggestions are based on the metadata in PalCom, i.e. service descriptions. Since we want to enter a message to be echoed, a text box (TextField) is selected. Based on the selection, the editor suggests suitable links (roles) between the chosen functionality and the selected graphical component. The selected link can be customized by selecting from a list of link qualifiers. In the echo example, the entered text should be provided as value for Voice::message, hence we select the provider link and the "text" qualifier (Figure 6.5b). The purpose of the link is presented in a human readable form: *"The*

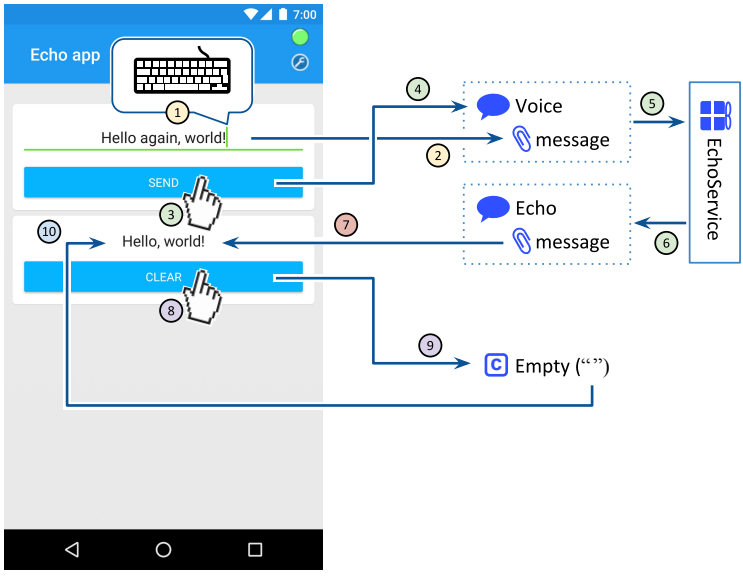
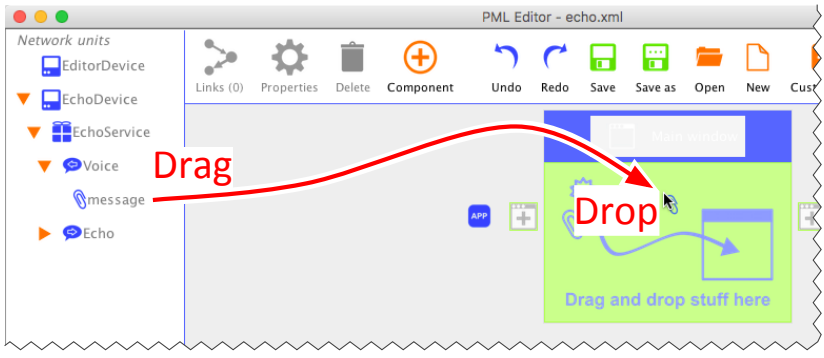


Figure 6.4: Screenshot of the finished GUI for the echo example, depicting functionality and data flow. The user sends messages (1–5) that are echoed by a PalCom service (6) and displayed in the GUI (7). Previous messages can be cleared (8–10).

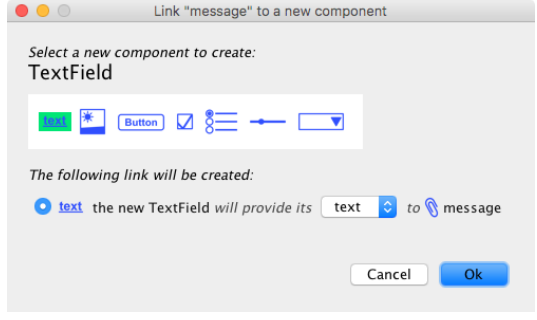
new text box *will* provide *its* text *to the* message *parameter”*. Upon confirming the selection of graphical component and link, a second dialog appears. This dialog allows properties to be set for the new graphical component, e.g. font and size.

The newly created text box is now visible on the canvas, in the tile of the main window (Figure 6.5c). It is at this stage in development customary to test the GUI on a physical devices. With e.g. a mobile phone connected to the development computer, the user can press “Run on Android” in the toolbar to install the GUI description that is being developed on the phone and automatically start the interpretation process. We verify that the text box for Voice::message appears as expected by running our description on a Huawei Nexus 6P where the interpreter tool AndroidPUIDI is installed; the resulting GUI is shown in Figure 6.6. With these few steps, we have created a fully functioning GUI that runs on a physical device and connects to a remote service. The GUI allows a text to be entered into the text box (1–2, Figure 6.4). We note the text is aligned to the left, where we would rather want it to be centered. To rectify this, we select the text box on the canvas and press “Properties” in the toolbar. This opens the same properties dialog that was presented when we first created the component. We adjust text alignment accordingly, and confirm the changes.

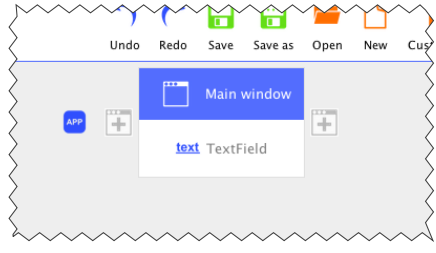
The above workflow is repeated in order to actually send the message in the text box to EchoService. We locate the voice command in the network pane, drag



(a) Functionality is dragged from the network pane and dropped on the canvas.



(b) Suitable graphical components and possible links (here: one) are presented.



(c) The selected graphical component (“TextField”) is added to the canvas.

Figure 6.5: Workflow for the inverted development approach, as implemented in GPE.

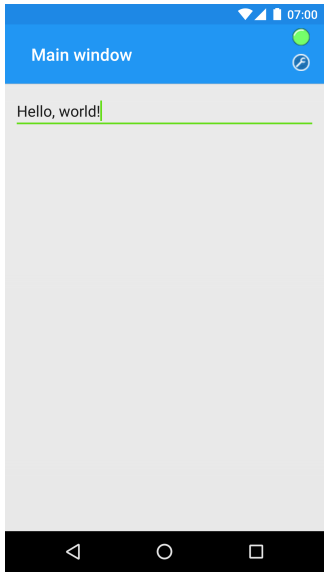
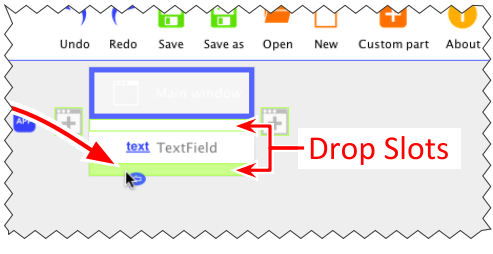
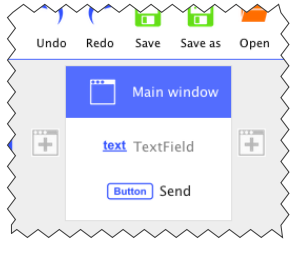


Figure 6.6: Screenshot of the current description, as rendered by AndroidPUIDI on a Huawei Nexus 6P; the corresponding editor canvas is Figure 6.5c. Text ("Hello, world!") can be entered into the text box (1–2, Figure 6.4).

it to the canvas, and position it in relation to the text box we added previously. Figure 6.7 illustrates component positioning in GPE. We drop Voice in the slot below the text box (Figure 6.7a). The editor again presents suitable graphical components and links. We select a button to act as invoker for the command, and change its text property to "Send" (Figure 6.7b). Again, we save our progress and run the description on the Android phone, verifying the work progress thus far: we can enter a text to echo, and send it using the button (1–6, Figure 6.4).



(a) Components can be dropped in slots marked by a green border – current selection (hover) is filled.



(b) Components are added to the canvas in the specified slot.

Figure 6.7: Positioning new components in relation to previously added components.

However, the echo reply is not yet handled by the GUI. We repeat the process one more time: we identify the desired functionality in the network pane, i.e. the message parameter of Echo, drag it to the canvas, and drop it in the slot under the send button. A text label is selected as viewer for Echo::message. We run the description on the phone, and confirm that the message that is typed in the text box is echoed to the text label when the send button is clicked (1–7, Figure 6.4).

For the final piece of GUI functionality, i.e. clearing echoed messages, the workflow is the same with a small addition: we start by creating the empty constant whose value will overwrite the latest received message. In the application bar, we press "+ Facts", select the constant type, and leave the value property as an empty string ("") in the properties dialog. The empty constant is what is referred to as an *unplugged component*, i.e. a component that has no links and therefore has no behavioral effect on the GUI. With the constant in place, the workflow is as before: we drag it from the application bar, and drop it in the bottom slot of the main window tile. A button is selected to invoke the constant once clicked, thus propagating its empty value to all viewers (none yet). The button is labeled with the text "Clear". By testing on the phone, we verify that the clear button – which currently appears to do nothing – has been properly added to the GUI (8–9, Figure 6.4). The workflow of GPE is flexible: the same effect as above could have been achieved by first creating an unplugged button in the main window, dragging that to the application bar, and selecting to create and link a new constant.

Next, we must link the empty constant to the text label, in order for its invocation (by the clear button) to take effect in the GUI. We drag the text label onto the empty constant (Figure 6.8) – or vice versa – and select a viewer link for the text property of the label. This development step exemplifies creating links between existing components. In general, the user can choose any component from the network pane, canvas or application bar, and drag it over any other component. If a link is possible, the border of the target component is colored green, otherwise red. Dropping the source component on a valid target opens a dialog with suggestions for suitable links between the two chosen components. This dialog is identical to Figure 6.5b, except no new component needs to be selected.

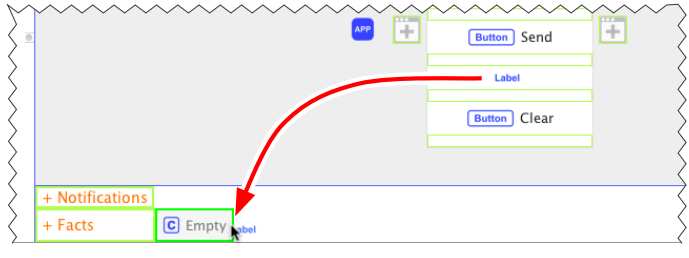


Figure 6.8: Additional links can be create between existing components. Dragging one component over another reveals if a link is possible (green border).

With this mechanism for linking existing components, it is possible to create advanced designs where components have multiple links, and can be part of long chains of events. To increase developer productivity in such designs, GPE has helpful features to visualize and review links, as illustrated in Figure 6.9. In our example, if we click on the text label in the canvas, it will be highlighted in bright green (1). All other components that are linked to the selected component will be highlighted in green as well: the empty constant (2) and Echo::message (3). Moreover, all parent components of the linked components will be highlighted in light green (4), thus making it easier to locate linked components that might be "buried" in collapsed trees. Additionally, the links button in the toolbar displays the total number of links for the selected component, e.g. "Links (2)" (5). Pressing the links button brings up a dialog (6) with a detailed listing of the links to and from the component. In this dialog the user can edit and remove existing links. Furthermore, clicking on a component (7) will reveal it in the editor – even if buried – with a red flashing effect (3).

With the empty constant and its links in place, the described GUI has all specified functionality, as outlined in Figure 6.4 (1–10). What remains to be done is a number of graphical changes. For this, we will need to add unplugged components, i.e. components without any initial links. Unplugged components are typically used for layout work, for marking up GUIs with helpful labels, and for designs where intermediate steps are needed (as with the empty constant). They can be added by pressing "Component" in the toolbar. This brings up a palette, from which components can be dragged onto the canvas or application bar (Figure 6.10). After being added, unplugged components can be linked like any other component, as discussed.

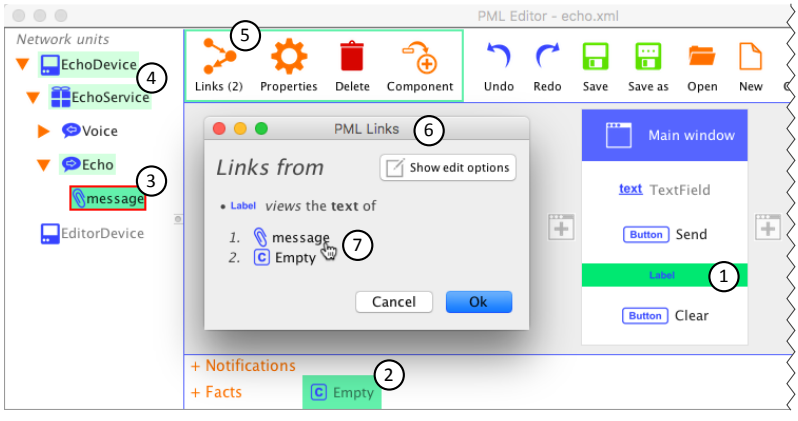


Figure 6.9: Reviewing links in GPE. Selected components are highlighted (1), as are linked components (2–3) and their parents (4). Features for a detailed link count (5), listing (6), and review (7) are also available.

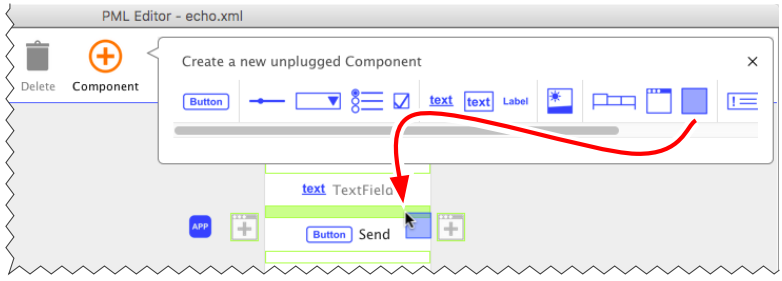
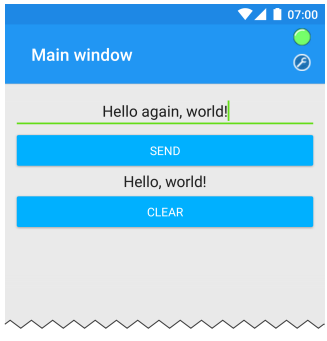
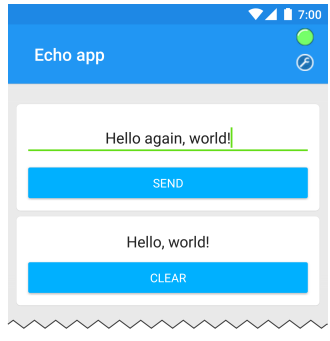


Figure 6.10: Components with no initial links can be added from the toolbar; *unplugged* components are created by dragging from a palette to the canvas or application bar.

For the echo example, the current graphical design of Figure 6.11a needs to be changed to match Figure 6.11b. Two unplugged *containers* are added to the main window; the text box and send button are moved into the top container, and the text label and clear button into the bottom container. Containers are layout components that can contain other graphical components, and the primary means of controlling the layout of GUIs. In GPE, graphical components can be re-arranged at any moment by dragging them to another valid slot on the canvas. We edit the properties of the containers to match our goals: white container style, and vertical (linear) orientation of sub-components. Furthermore, we change sizing and padding properties for a number of components, and set the title of the main window to "Echo app". At this stage, the content of the canvas and application bar matches Figure 6.2. We save our progress and run on the Android phone one last time. The result, both graphical and functionally, matches Figure 6.4.



(a) All graphical components directly in the layout of the main window.



(b) Graphical components in separate layout components (white areas).

Figure 6.11: Screenshot of the current description as interpreted before (a) and after (b) the final graphical design changes.

6.2.4 Discussion

As we have shown, the development process in GPE can be highly accessible, particularly in that no program code needs to be written – all interactions are graphical. Even though we have hinted that advanced designs are possible, the functional expressiveness of GPE is restricted when compared to conventional editors, where any type of functionality can be created with program code. In terms of graphical expressiveness, more component classes and properties can be added to the current suite of PML in order to meet additional recurring needs. Furthermore, custom components can be created by developers for specialized needs. However, what makes the codeless functionality specification of GPE possible is the core mechanism of PML links and the metadata in PalCom, which the editor uses to "understand" the application model. These cornerstones are unlikely to change drastically in future revisions. Hence, the functional expressiveness of GPE will remain at levels similar to now. Although conventional editors are more functionally expressive, this expressiveness comes at the expense of accessibility: the user must have programming skills. There is thus a conflict between accessibility and expressive power. With the goals of the work presented in this dissertation, accessibility was prioritized for GPE and PML, with efforts to ensure practical scalability for created GUIs.

We have used the development of a simple GUI to illustrate the workflow and primary features of the Graphical PML Editor. Although the development process of GPE has its limitations, we conclude that the editor can be accessible, flexible, and accommodating to advanced designs.

Chapter 7

Applications in E-Health

The contributions presented as part of this dissertation have been applied in a number of research projects and scenarios, mostly in the domain of e-health. In the itACiH project, PML has been evaluated in a real-world context, and improved based on real-world requirements. Furthermore, in a number of related research projects, other members of our research group have successfully and independently applied the contributions to create professional grade applications.

7.1 Language Scalability Evaluation

In order to determine the practical viability and scalability of PML and the corresponding tools, these where applied to create the mobile application for the advanced home care scenario in the itACiH project.

7.1.1 Introduction

The scalability evaluation of PML was performed empirically in the context of the itACiH project. A PML described GUI was developed, and deployed on Android tablets at the unit for advanced home care (ASIH) in Lund. The tablets were targeted towards the mobile ASIH teams that consist primarily of nurses, but to some degree also of other healthcare professionals, e.g. physiotherapists. The purpose is for the teams to treat the patients in their homes; to check up on them, follow up on previous issues, deliver medication and other consumables, take measurements, etc. The aim of the developed GUI was to streamline the workflow of the ASIH teams, which was previously mostly paper-based. In return, the scenario provided a natural test bed for the solution we present in this dissertation.

The ASIH application, i.e. the ASIH GUI and its components in the itACiH system, was developed using participatory design [32], i.e. prospective users such as nurses actively participated in the design process. Furthermore, the development process was iterative: the application was built and delivered in increments to the users, where it was tested by professionals in a real-world, practical environment. From using our solution in such an environment, PML could evolve beyond the

early prototype stage. When new features – user requested or otherwise – were added to the application, new requirements for PML followed as well. This resulted in a mutually beneficial loop of test-feedback-improve-repeat, where PML was improved based on real requirements and tested in a real setting.

The formal goal of developing the ASIH GUI was to analyze PML for the purpose of evaluation, with respect to language scalability in the context of a real-world development project. Informally, we wanted to investigate whether PML would scale beyond small prototype GUIs such as the echo example (Chapters 5 and 6), and whether the language could be used to describe professional grade GUIs. Note that development on the ASIH GUI was started before the work on the Graphical PML Editor (GPE) had been finalized. Hence, the GUI has only been partially developed in GPE, and as such, this evaluation pertains to the scalability of PML only – not the editor.

7.1.2 System Architecture

From the perspective of the mobile ASIH teams, the itACiH system is divided into two primary parts: the Android tablets that the nurses bring when visiting patients in their homes, and the servers that provide functionality e.g. storing patient data. Figure 7.1 illustrates this simplified view of the system. The ASIH GUI is deployed in a system with locally augmented distributed models, i.e. the application model is distributed across multiple servers, with augmentation by a local model. On the tablets, two Android applications are installed. AndroidPUIDI reads the PML description of the ASIH GUI and interprets it as such. This application is started by the nurses when they want to interact with the system. The second application is TheAndroidThing, which is started automatically when the tablet boots up, and thereafter runs silently in the background with no need for user interaction. Both applications are represented as individual PalCom devices, even though they are technically running on the same physical device (tablet). As the two devices run

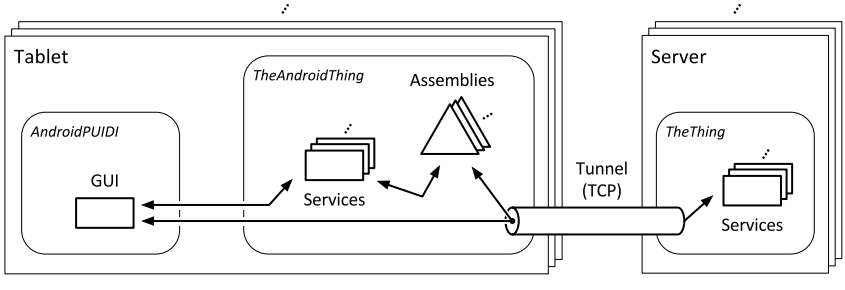


Figure 7.1: Architecture of the itACiH system, from the perspective of the ASIH tablets. Local services augment the functionality on the servers, making both offline and real-time support possible. Tablets connect over encrypted TCP tunnels and 3G/4G.

on the same tablet, AndroidPUIDI has uninterrupted access to the services of TheAndroidThing. The services on the server (TheThing) are accessed through encrypted TCP tunnels, and are thus only accessible when the tablet has an active Internet connection.

It is not uncommon for patients to live in rural parts of the country, where 3G/4G coverage is spotty at best. Hence, the augmented architecture, which enables the ASIH GUI to operate locally when the servers are unavailable (*offline mode*) and with real-time updates and commits otherwise. For features that do not have to be available to the user in offline mode, the ASIH GUI connects *directly* to the corresponding services on the servers. In offline mode, the services deployed locally in TheAndroidThing mirror select application critical functionality from the servers. This applies in both directions, i.e. for communication to and from servers. For example, the local services can buffer patient data collected during an offline period. When the connection is re-established, the data can be sent to its final destination. Reversely, the local services can cache server data to make it available when offline. To keep synchronized, local services are connected to distributed services via assemblies, creating an *indirect* connection between the GUI and the distributed services. Even when in online mode, data to/from these services is transferred via the indirect connections. This ensures that in the case of sudden loss-of-contact, data will not be lost or become unavailable. When online, the difference between direct and indirect connections is insignificant; in practice, the system operates in real-time in both cases.

As a final reflection on architecture, please note that the overview in Figure 7.1 is a coarse-grained simplification of the reality. The servers appear to work in isolation, when there are in fact multiple inter-server connections. Furthermore, Figure 7.1 only shows the architecture from the perspective of the ASIH tablets; other applications that push/pull data to/from the server are not presented.

7.1.3 Application Overview

The features of the ASIH application have been allocated on 17 separate screens in the GUI. Some of these screens have been divided into multiple views in order to improve accessibility, e.g. using tabs. In Figure 7.2, all of the in total 25 unique views of the ASIH GUI are depicted as miniatures. The purpose is to illustrate the scope and the flow of the GUI; individual views are not intended to be legible. The *patient selected screen* (B8) and the *pain analysis screen* (B1) will be discussed in greater detail.

In the upper half (yellow, A) of Figure 7.2, the views for basic and non-patient-specific features are outlined. When the user starts the ASIH GUI for the first time, the *login screen* (A1) is presented: user credentials are entered and sent to the server for verification. Based on the response of the server, either a dialog box appears to explain why the login attempt failed, or the *main screen* (A4) is opened. From the main screen, staff-oriented features can be accessed. For example, the

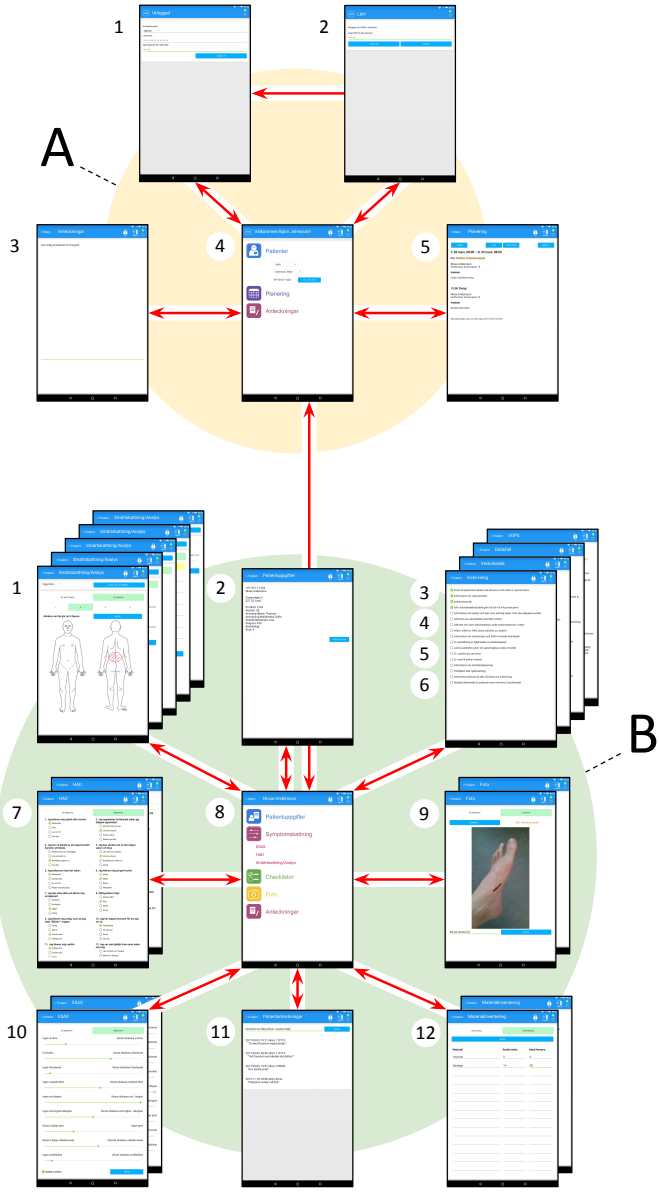


Figure 7.2: The ASIH GUI is composed of 25 unique views on 17 different screens. Most features are patient-oriented and accessed from the patient selected screen (B8). Some screens, e.g. the pain analysis screen (B1), are divided into views for ease-of-access.

user can view their personal work schedule (A5), including which patients to visit and what tasks to perform. The planning is based on live updates while online, and buffered content otherwise. From the main screen, the user may also choose to navigate to the patient selected screen (B8): from a list of active patients that can be filtered based on groups, the user selects a specific patient, thus opening the screen for that patient. Patient listings are managed by a local service which is synchronizes with the server while online.

At any time during operation, the user may choose to log out, thus returning to the login screen. Furthermore, she may also choose to lock the GUI in order to prevent unauthorized access of e.g. patient data. This opens the *locked screen* (A2), from which there are only two transitional options: either log out, or unlock the GUI with the session-specific PIN code that was chosen at login. While login credentials are verified remotely, the state of the GUI is managed by a local service. This means that even when the login server is unavailable, the GUI can still transition between some states, e.g. the lock and unlocked states. This allows for previous sessions to be resumed e.g. after a tablet reboot, even when offline. Due to security and battery life considerations, the ASIH tablets have been configured to turn off the display after two minutes of user inactivity. When this happens, the GUI is automatically locked. A local service monitors the display state of the devices to make this possible.

In the lower half (green, B) of Figure 7.2, the views for patient-related features are outlined. This half is significantly more populous than the upper, as most features of the ASIH GUI are currently patient-centric. The features are centered around and accessed from the patient selected screen (B8). For the fictive patient Olle Nilsson, this screen is illustrated in Figure 7.3. At the top of every screen in the ASIH GUI there is a blue strip referred to as the *toolbar*. The toolbar shows the title of the current screen to help with orientation in the GUI, and navigation options to transition between screens. On the patient selected screen, the name of the selected patient is shown, with the option to return to the main screen (*Meny*). The options to lock (*Lås*) and log out (*Logga ut*) are present in the toolbar of all screens.

The features that can be accessed from the patient selected screen are presented as colorful icons. Pressing one such icon transitions the GUI to the screen for the corresponding feature. Some icons are "stitched" (dashed border) and reveal additional options once pressed. For example, in Figure 7.2 the assessment icon (*Symptomskattning*) has been pressed to reveal several methods for patient symptom assessment. The currently available features are briefly described below, in order of appearance on the patient selected screen from top to bottom.

Information On the *patient information screen* (B2), information about the currently selected patient is presented: name and Swedish civic number, address and phone number, physician in charge, etc. The number of entries are dynamic, and can be customized for each patient. Patient information is

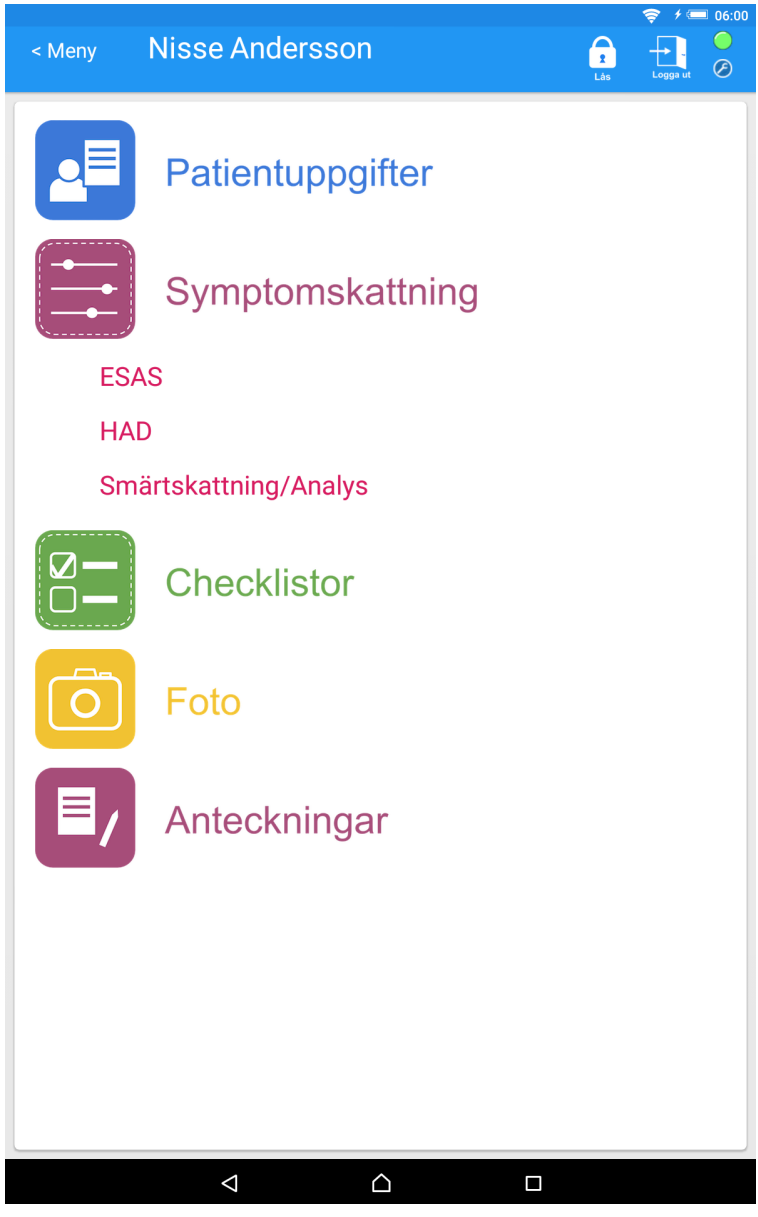


Figure 7.3: Patient selected screen for Olle Nilsson, with options (top) to return to the main menu, lock the GUI, and log out. Available features are presented as icons; "stitched" icons, e.g. *Symptomskattning*, can be pressed to reveal additional features.

presented as text, with one entry per line, and is pulled from a local service that caches information from the server. For user convenience, there is an option to show the address of the patient on a map. This opens Google Maps, where driving directions can be obtained. The GUI uses a local service to initiate the application switch.

Assessment Three methods for patient symptom assessment are available: the Edmonton Symptom Assessment System (ESAS) [20], the Hospital Anxiety and Depression Scale (HAD) [91], and a method for pain assessment and analysis that is to be discussed in Section 7.1.4. On the screens of the two former methods (B10, B7), there are two views: one for viewing the latest assessment, and one for submitting new ones. Assessments are fetched directly for the server, with no offline support – the view is blank when offline to avoid accidental usage of outdated assessments. It is, however, possible to submit new assessments for all three methods in offline mode. Such assessments are buffered on the stable storage of the tablet until uploaded and acknowledged by the server.

Checklists Checklists are provided for tracking the progress of multi-step activities. These list all the tasks that must be carried out e.g. when initiating new patients or during weekly visits; each task can be marked as completed (checked) individually. In total there are four checklists available, each one on a separate screen (B3–B6). Checklists are an online-only feature, with real-time updates to facilitate cooperation: changes are automatically submitted to the server, and are instantly propagated to other users. The content of the checklists are statically defined in the GUI description. A screen for material stocktaking (B12) also falls under this feature category. It allows the user to track and update the materials that are currently available in the home of the selected patient. Stocktaking is only available when online.

Photography The photos feature is primarily used to document the physical disorders of patients. When online it can be used in real-time, e.g. by sending a photo while on the phone with the doctor to get instructions on how to proceed with a patient. Its screen (B9) is divided into two views: one for viewing photos and one for taking new ones. Previous photos are not cached for offline viewing. Furthermore, to save on bandwidth photos are not automatically pushed to tablets; the user needs to explicitly request them in the GUI. To take new photos, the native camera application of the device is opened. After taking the photo, it is presented for the user to review and attach a comment. Committed photos are buffered locally. Due to the relatively long transfer times of photos (typically a few seconds), the input controls are disable during transmission to prevent multiple, accidental commits.

Notes On the *patient notes screen* (B11), the user can author patient-specific notes. These notes are not part of the medical record of the patient, and their nature should thus be more casual, e.g. "Patient has bad hearing – knock loudly!". The feature was designed for asynchronous coordination among users, although real-time chatting is also possible. The notes are presented in a list, stacked on top of each other in reverse chronological order; date, time and author is shown for each note. Offline support is provided for both viewing and authoring notes, with local caching and buffering.

A thorough walkthrough of every feature is outside the scope of this dissertation; the above gives a quick overview of most features. Next, the pain analysis feature will be presented in greater detail as an example.

7.1.4 Sample Feature

In Figure 7.2, the pain analysis screen (B1) can be seen to be divided into five separate views. These have been designed based on the Handbook of Healthcare [18], a Swedish national resource for caregivers and healthcare personnel. The first and simplest of the views deals with pain assessments. Assessments pertain to one specific pain that the patient is currently experiencing. Since a patient may experience multiple – but separate – pains simultaneously, all pains are registered with a working title (name), e.g. "back pain", in order to distinguish between them. The user input for an assessment is trivial: a value of 0–10 indicating the intensity of the pain, as experienced at the moment of the assessment. Due to this simplicity, pain assessments can be performed frequently, e.g. once per visit. Pain analyses, however, are more involved. They are typically performed once when registering a new pain, and are thereafter updated sporadically as the pain changes character.

The views of the four steps in the pain analysis process are shown in Figures 7.4 to 7.7. Each view is presented as a separate tab on the pain analysis screen: "1", "2", "3", and "4" respectively. The topmost layout section on the screen is present in all four steps. In this section, a drop-down list (top left, Figure 7.4) can be activated to reveal the names of all registered pains. From this list, the user selects which pain to analyze. To register a new pain, the user presses the button (right). This brings up a dialog box where the name of the pain can be entered, with options to confirm or cancel. Pain names are buffered and cached locally on the ASIH tablets, and can hence be added and listed, respectively, even when offline. When a new pain is added, it is automatically selected in the drop-down list. In offline mode, a dialog box appears to inform the user that the new addition has been buffered. The user must actively confirm this dialog to dismiss it. When online, feedback is delivered in a more subtle form: a quick-note, i.e. a self-dismissing message strip (Android *toast*). Different feedback methods (attention levels) are used to make the user aware of the effects of working offline, i.e. that the new addition is not yet available to other users in the system.

The first step of the pain analysis is shown in Figure 7.4. In this step, the

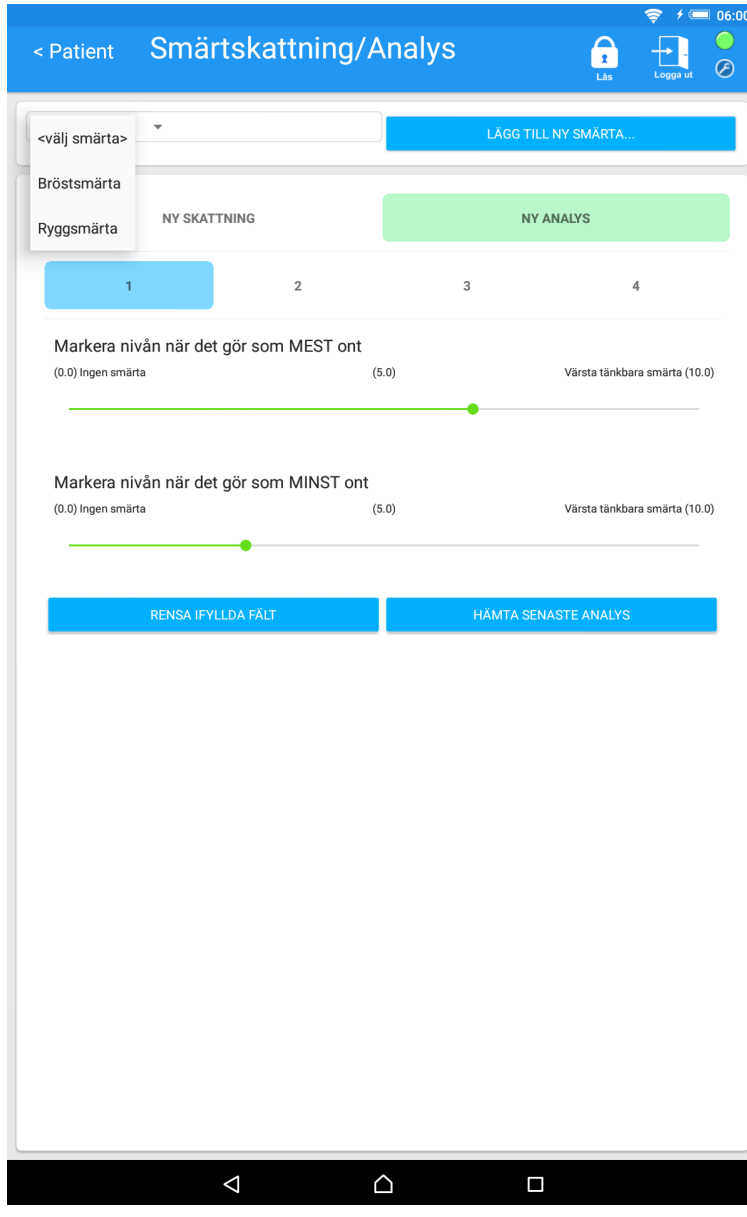


Figure 7.4: Step 1 of pain analysis. The user slides the green knobs horizontally across the two bars to indicate the intensity extremities of the selected pain (top left, *Ryggsmärta*).

intensity of the selected pain is analyzed. Unlike assessments, which are snapshots of individual intensities, the analysis focuses on the extremities, i.e. the highest and lowest intensities experienced since the pain was first perceived. The user describes these extremities on a scale from 0 to 10, where the former is interpreted as "no pain" and the latter is interpreted as the "worst pain imaginable". Numeric sliders are used to enable intuitive input of the two values: the user slides the knob horizontally across the bar, gradually coloring it green from left to right. At the bottom of the view for Step 1, two buttons are laid out side-by-side. The left button is used to clear the content of an analysis: all input component for Steps 1–4 are reset to their respective default values. The right button is used to retrieve the content of the most recent analysis for the given pain. If no previous analysis exists, the user is notified in a quick-note. Analyses are not cached locally, and must hence be pulled directly from the server. In offline mode, the right button is disabled and cannot be pressed.

In the second step of the pain analysis (Figure 7.5), the user must describe the position on the body where the selected pain is experienced. To do this in an intuitive manner, a sketch is shown of a person both from the front and from behind. The user draws on this image to illustrate the pain; the annotation can be cleared by pressing a button above the image. Figure 7.6 shows the third step of the analysis. The user simply checks the boxes next to the characteristics that best describe the pain: dullness, burning, numbness, etc.

The fourth and final step of the pain analysis sees the user further explain the details regarding the selected pain. The view is shown in Figure 7.7. Questions such as "What triggers the pain?" and "What medications do you take?" are answered as free-form text in text boxes. In total there are six such questions. The 7th and final question of Step 4 is of a different type: multiple choice. The user selects one out of the three possible answers by clicking on the corresponding radio button. Unlike the check boxes in Step 3, at most one option can be marked at any time. The button to save the content of the analysis is located at the bottom of the view. When it is pressed, a local service checks the input from the user. In the case of errors, e.g. unanswered questions, a dialog box informs the user. Otherwise, the analysis is buffered locally on the tablet, before being sent to the server. Feedback is presented following the same pattern as when registering new pain names: self-dismissing quick-note while online, dialog box otherwise.

7.1.5 Discussion

In PML, advanced users can develop custom parts (graphical components) to complement the standard suite of PML parts. For the ASIH GUI, no such custom parts were developed; the standard parts covered all basic component needs. There are, however, a number of noteworthy borderline cases. The ASIH scenario was used to improve PML beyond the prototype stage. When development on the ASIH GUI started, custom parts had not yet been introduced to the language. As

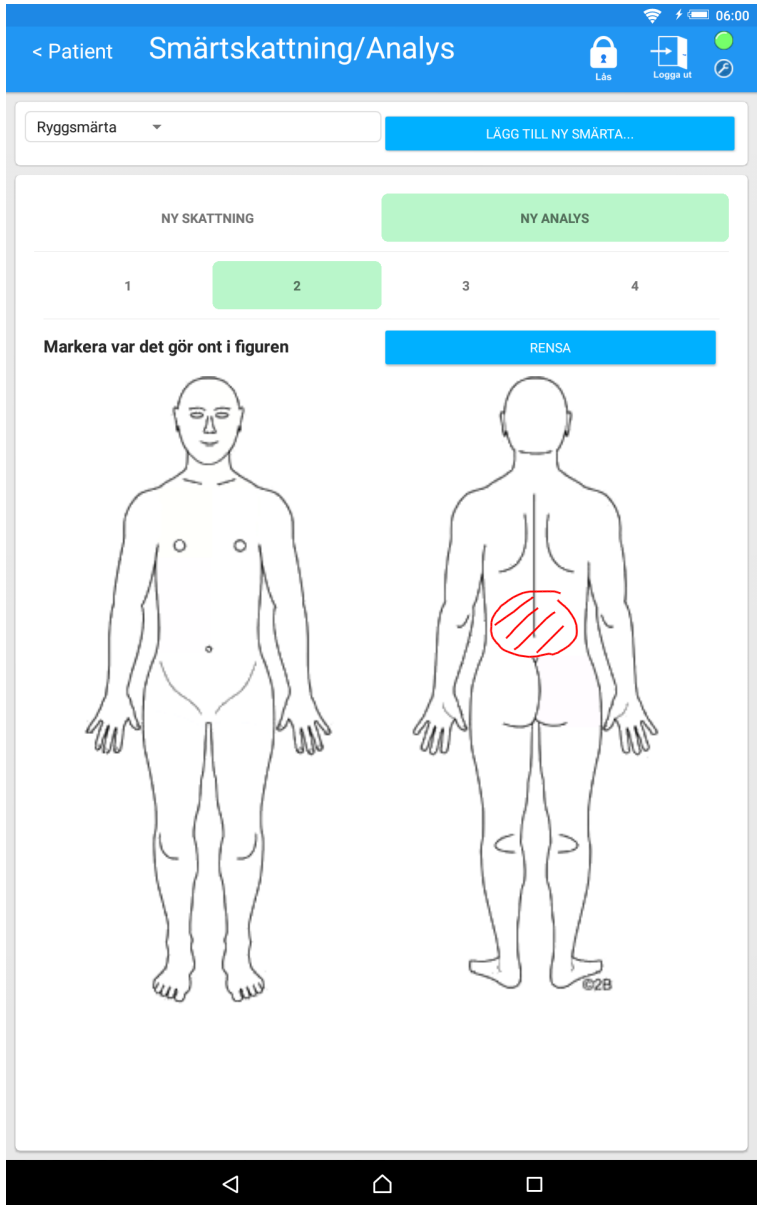


Figure 7.5: Step 2 of pain analysis. The user draws on the sketch of a person to describe the position on the body where the selected pain (top left, *Ryggsmärta*) is experienced.

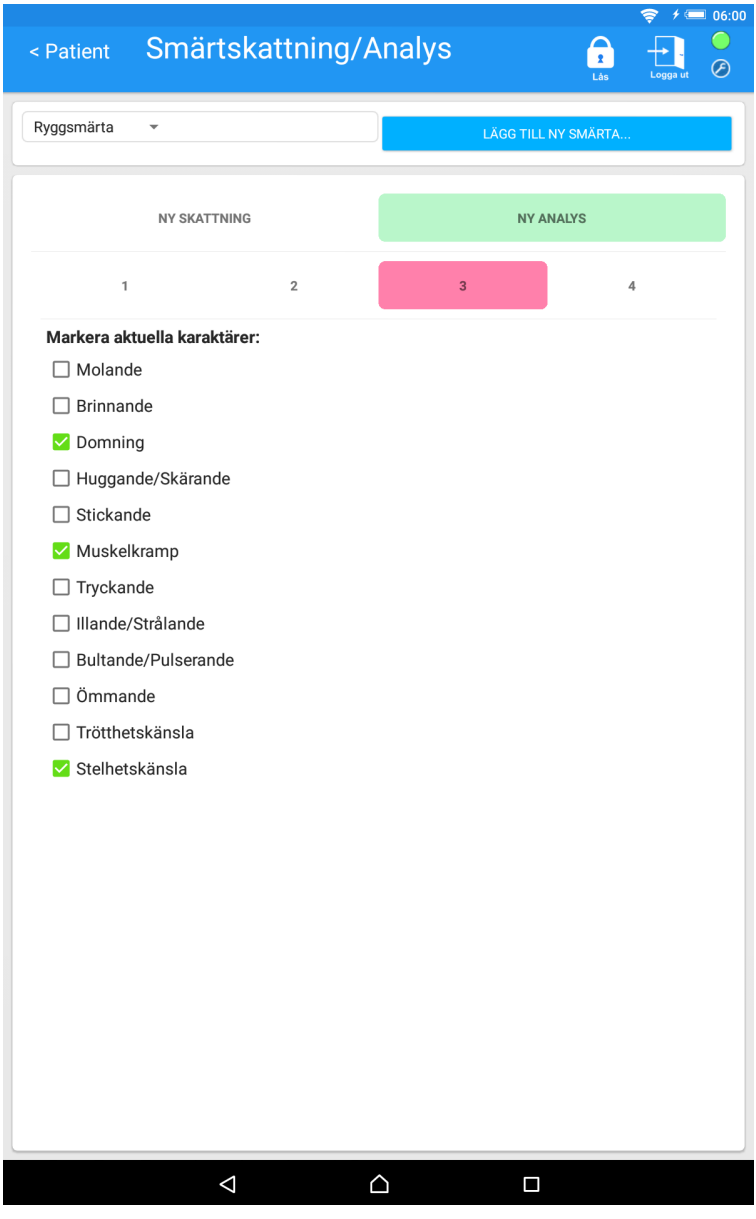


Figure 7.6: Step 3 of pain analysis. The user checks the boxes next to the characteristics that best describe the selected pain (top left, *Ryggsmärta*), e.g. numbness (*Domning*).

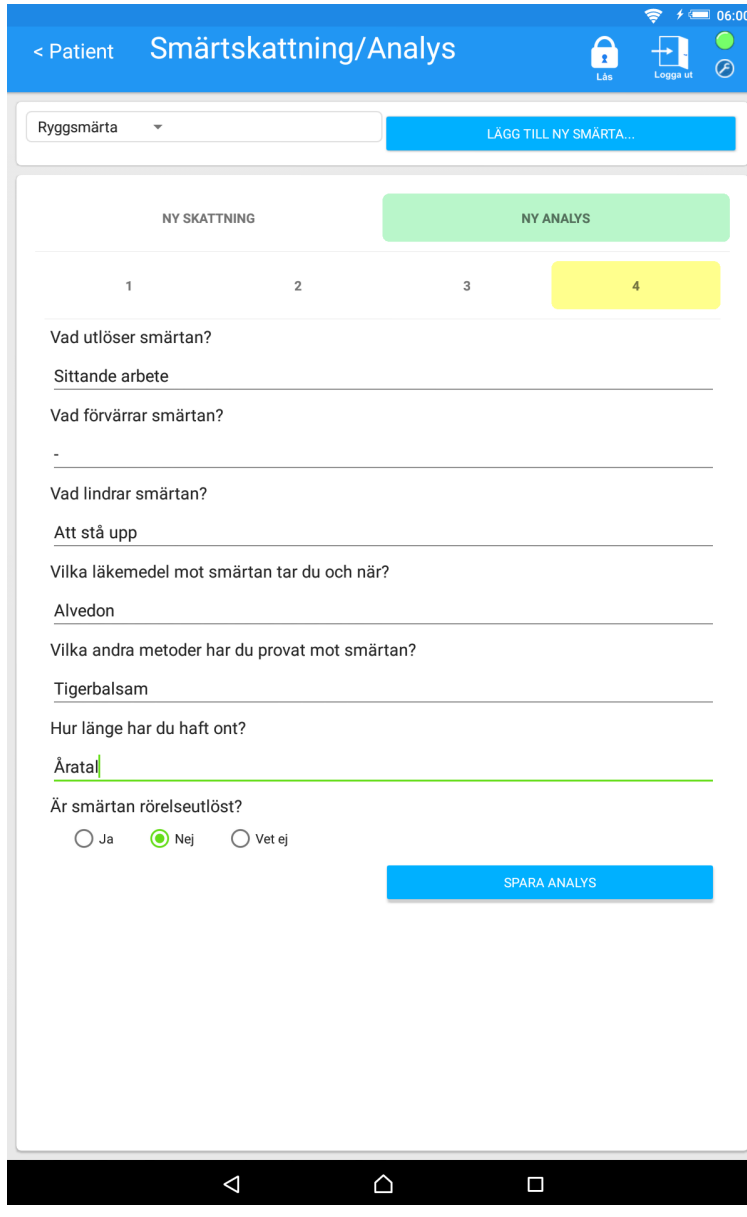


Figure 7.7: Step 4 of pain analysis. The users explains the selected pain (top left, *Ryggsmärta*) by answering questions as free-form text and by selecting from multiple choices.

such, some standard PML parts are more specialized than what would otherwise be expected. One example of this can be found in Step 2 of the pain analysis feature (Figure 7.5). The component that is used to display images also provides functionality for drawing. Such specialized behavior could be argued to better match the purpose of a custom part, rather than a standard part.

Besides graphical components, the other aspect to consider when building PML GUIs is the model, i.e. the PalCom services that are accessed. In the ASIH application, some service are highly specialized, e.g. the service that calculates HAD assessment scores, while others are more generic, e.g. the services for buffering and caching data. For advanced applications, the development of specialized services might be unavoidable. However, for the vast majority of applications it is entirely possible to avoid having to write program code. Conventionally, PalCom services are offered by physical devices, e.g. household appliances. To complement these, we envision a community library or "app store" that could provide commonly applicable services. For most applications, such services could bridge the functionality gaps that would otherwise require programming.

The benefits of working iteratively and in close contact with the users have been apparent in the itACiH project. One example is the photos feature, which was requested by the ASIH staff. Due to the staff's participation in the design process, they could independently identify desirable features that would fit well in the application. With the iterative work process, such features could be implemented and released to the users within a short time frame. Furthermore, the work on the ASIH GUI has allowed PML to evolve as a language. Screen transitions is one example of how PML has improved. Prior to ASIH, most GUIs had been simpler, one screen prototypes. As the ASIH GUI grew, multiple screens became a necessity for ensuring high accessibility. After revising PML, the iterative work process enabled the new capabilities to be applied in the ASIH GUI, and the updated product to be released shortly thereafter.

7.1.6 Conclusions

From evaluating PML in the real-world context of the itACiH project and ASIH Lund, and based on the discussion above, we conclude the following:

- As a language for describing GUIs, PML is scalable; building professional grade GUIs is practically viable.
- The standard suite of PML parts is comprehensive; intricate GUI designs are possible without writing program code.

7.2 Related Projects

In addition to the itACiH project and the ASIH application, the contributions presented as part of this dissertation have successfully been applied by other

members of our research group in a number of related projects in the domain of e-health.

7.2.1 Home-Based Peritoneal Dialysis

Renal failure is a medical condition where the function of the kidneys is impaired, resulting in inadequate filtering of metabolic wastes from the blood stream. The condition also causes patients to retain fluids that, if left untreated, can lead to further complications. As an alternative to traditional dialysis, peritoneal dialysis can be performed by the patients themselves in their homes. In short, a liquid solution is introduced to the abdominal cavity through a permanent tube in the lower abdomen. The solution is then drained, along with any excess fluid and the toxins it has extracted. The weight of the drainage is measured to evaluate the effectiveness of the solution. Depending on the outcome, a solution with a higher of lower level of concentration can be prescribed for the patient. Dialysis is typically needed a few times per day.

Previously, peritoneal dialysis patients from Lund and Malmö kept paper records of drainage results, their weight, and their blood pressure. Once every few months, the patients would schedule a visit with the doctor to review the measurements. For the patients, this meant that corrections to the treatment would be delayed until the next visit. For the doctors, interpreting several month worth of handwritten notes with no visual aid was tiresome. To alleviate both these problems, a digital support system was implemented as part of a special track in the itACiH project. The system continues to build on the PalCom framework, and consists of two applications: one web-based, and one for Android. In the former, the doctor can review graphically structured measurements that are updated in real-time, and communicate with the patient. The GUI of the latter was built with GPE, and is interpreted by AndroidPUIDI on Android tablets deployed to the patients. The system has been evaluated by some 10 patients in Lund and Malmö during 2015–2016, and is currently being expanded in Lund to include more users.

The architecture of the patient application is similar to that of the ASIH application, with both local and distributed models, and has been implemented with the architectural tools presented in this dissertation. In addition to the tablet itself, the patient is also issued a blood pressure monitor and a scale. These communicate wirelessly via Bluetooth with local services on the tablets. Whenever the patient uses any of the equipment, the results are handled by these services and automatically propagated throughout the system. Measurements are sent in real-time when possible, and are buffered locally otherwise. The patient GUI gives an audible queue when a new measurement has been successfully transmitted. Hence, when operating the equipment, the patient needs not interact with the actual tablet.

The main screen of the GUI consists of three views: send measurements, view measurements, and photos/chat. These views are presented as tabs in respective order from left to right, as shown in Figure 7.8, where the first view is on display.

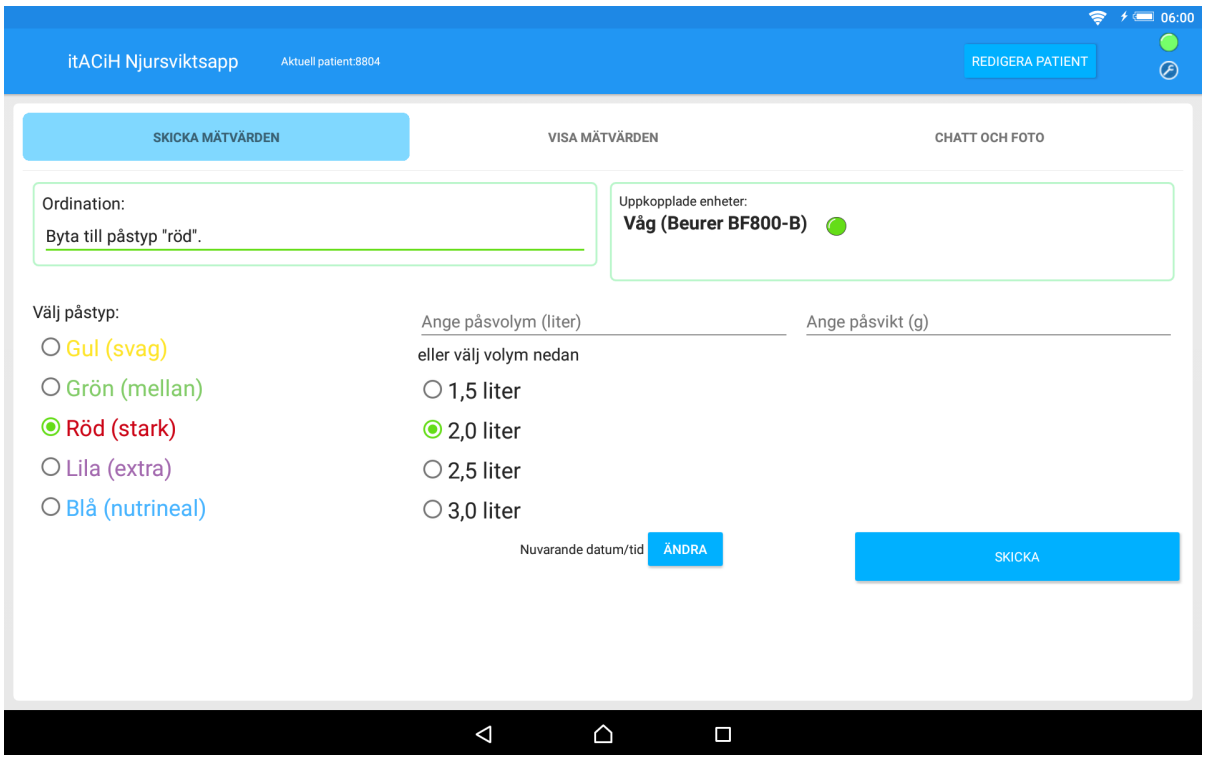


Figure 7.8: The patient GUI for the peritoneal dialysis scenario. On the first (of three) views (tabs) on the main screen, dialysis results are reported as solution type/color (left), solution volume (middle), and drainage weight (right). The states of connected devices are displayed (*Våg*, top right).

As feedback to the patient, this view shows all connected equipment and their status (*Våg*, top right) as reported by the local Bluetooth services. After each dialysis session, the patient reports the results in terms of solution type (color i.e. concentration), solution volume, and drainage weight. Like the other measurements, these are uploaded in real-time when possible. If the doctor, who reviews these measurements remotely, prescribes another type of solution, the patient is notified in the GUI and the prescription is shown on the first view (*Ordination*, top left). For more elaborate discussions, both the patient and the doctor can initiate a chatting session; from the third view, the patient selects from a list of doctors to start chatting. When further involvement is needed, the doctor can initiate a video conference with the patient. In such cases, the patient GUI will ring, and present options to either answer or decline the call. To document physical disorders, e.g. swelling around the dialysis tube, the patient can access a separate screen (from View 3) for taking and uploading photos. On the second view, the patient can review previous measurements, with options to filter by measurement type and time period.

7.2.2 Pre-Hospital Care

People call for an ambulance for all kinds of more or less motivated reasons. One task for paramedics in Skåne County is hence to establish the proper level of treatment. This may range from cases where the patient must be rushed to the emergency room, to complete prank calls where no action by the staff is necessary. In many cases, however, the level of treatment is not obvious. The paramedics may be considering less acute options than the emergency room, such as referring the patients to her doctor or treatment team. When in doubt, they can call "Regionalt Läkarstöd" (RLS), a regional team of on-call doctors assigned to provide support in medical cases. After getting in contact with a doctor, the paramedics verbally describe the situation and the vital parameters of the patient. The doctor takes manual notes, based on which the final decision for level of treatment is made.

This method of communication and decision making has its deficiencies. First, visual queues can have a deciding impact on how to proceed with a patient. However, the on-call doctor cannot see the patient. Because of this, the doctor must decide on the more acute options when unsure in order to not endanger the patient. This can be an inconvenience to the patient, and possibly wastes medical resources that could otherwise have been better allocated. Furthermore, the number of vital parameters that need to be negotiated between paramedic and doctor can be substantial – see Figure 7.9. Performing this synchronization task over the phone under time constraints is prone to human error. In a research project funded by Vinnova [80], a support system for ambulance consultations that addresses these problems has been developed. It was developed collaboratively with a small group of RLS doctors and paramedics, and is currently entering the beta testing stage, where all RLS doctors and several ambulance stations will be included.

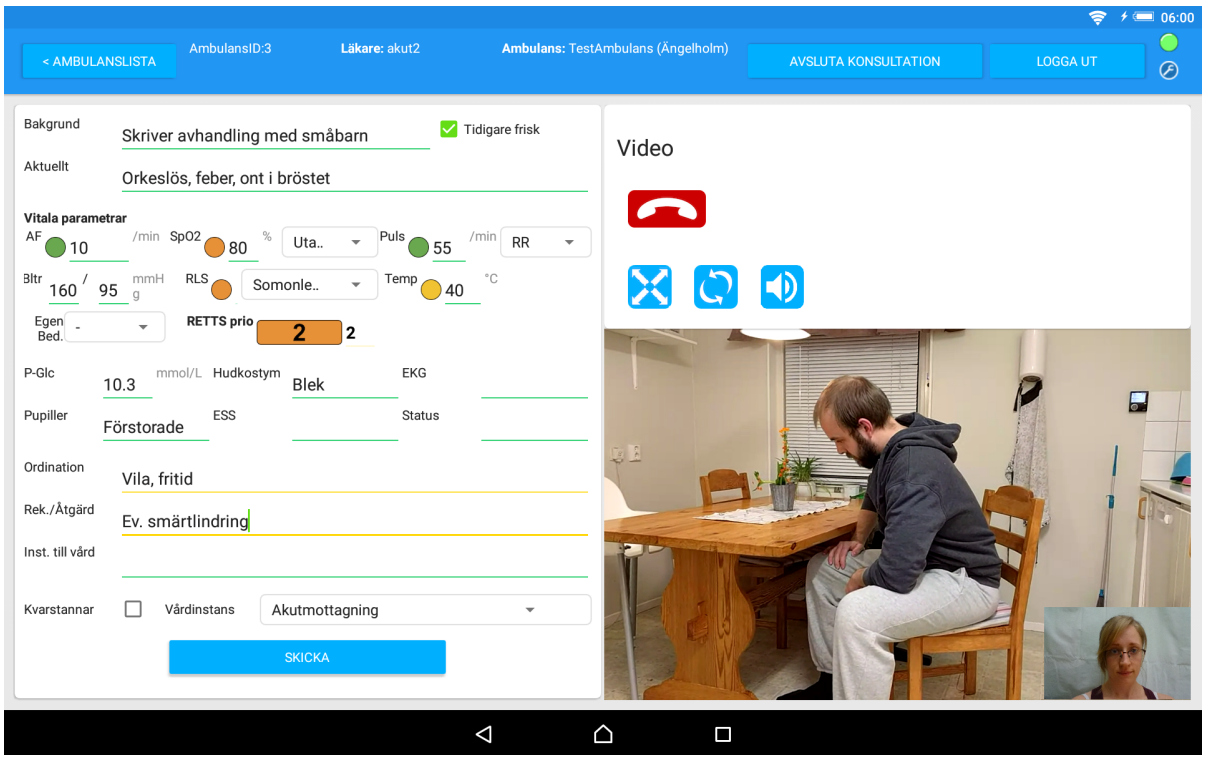


Figure 7.9: The doctor GUI for the pre-hospital care scenario during an ongoing consultation. The doctor and paramedics cooperate in real-time to enter patient data and vital parameters (left). The patient is shown on video (right) to complement the audio of the policy-mandated phone call.

The system, which is built on the PalCom framework, consists of two applications: one for the doctors and one for the paramedics. Both applications are dispatched on Android tablets, and are built on an architecture of locally augmented distributed models, using the architectural tools presented in this dissertation. Due to the immediate nature of the consultations, no offline functionality is provided in this scenario. Local services are used for calculations and transformations of data, while distributed services are used for pushing and fetching data. The GUIs of both applications were created in GPE, and the resulting PML descriptions are interpreted by AndroidPUIDI on the tablets. In terms of content, both GUIs are nearly identical. The layout of the main screen is different, however, to best suit the roles of each party. Furthermore, the doctor GUI is more powerful in that it allows for the doctors to edit prescription-related content, whereas the paramedics can only view this part. Figure 7.9 shows the main screen of the RLS doctor GUI.

The developed applications serve as a complement to the current phone-based support system: the doctors and paramedics communicate verbally on their phones, and use the tablets to enter patient information and vital parameters, and for video conferencing. At the start of their shifts, the paramedics identify their ambulance in a list in the GUI. From the main screen, new patient cases can be started (and terminated) for the selected ambulance. Such cases instantly become visible in the doctor GUI, where each new case is represented as an icon that indicates its priority (color, value 1–3). Once a consultation has been made, cases can remain open for additional consultations. The doctors select cases to start new consultations, which opens the main screen (Figure 7.9). In their respective GUIs, both paramedics and doctors can cooperate in real-time to fill out the various fields for patient information and vital parameters; paramedics can start entering values before calling the doctor. The user decides when changes are propagated to the opposite party; possible merge conflicts are handled in local services. Edited fields are clearly marked in the GUI (yellow underlining). Some parameters are used to calculate the priority of the case. The fields of such parameters are marked with circular icons that are colored based on the input. If the doctor sees the need, a video conference can be initiated. As a complement to the ongoing phone call, the paramedics can thus show the patient on video.

7.2.3 Home-Based Neonatal Care

Long hospital stays are universally undesirable, and would ideally be avoided whenever possible. One possibility for this is hospital-based home care (HBHC), where patients are enrolled at a hospital but are treated in their homes. From the perspective of the healthcare provider, this frees up beds earlier. For the patient, the discomfort of spending time away from home is avoided. These benefits become even more pronounced when the patient is a child. In such cases, the consequences of a hospital stay are significantly worse: not only does the patient have to be in the hospital, but so does at least one parent. In practice, the families must coordinate

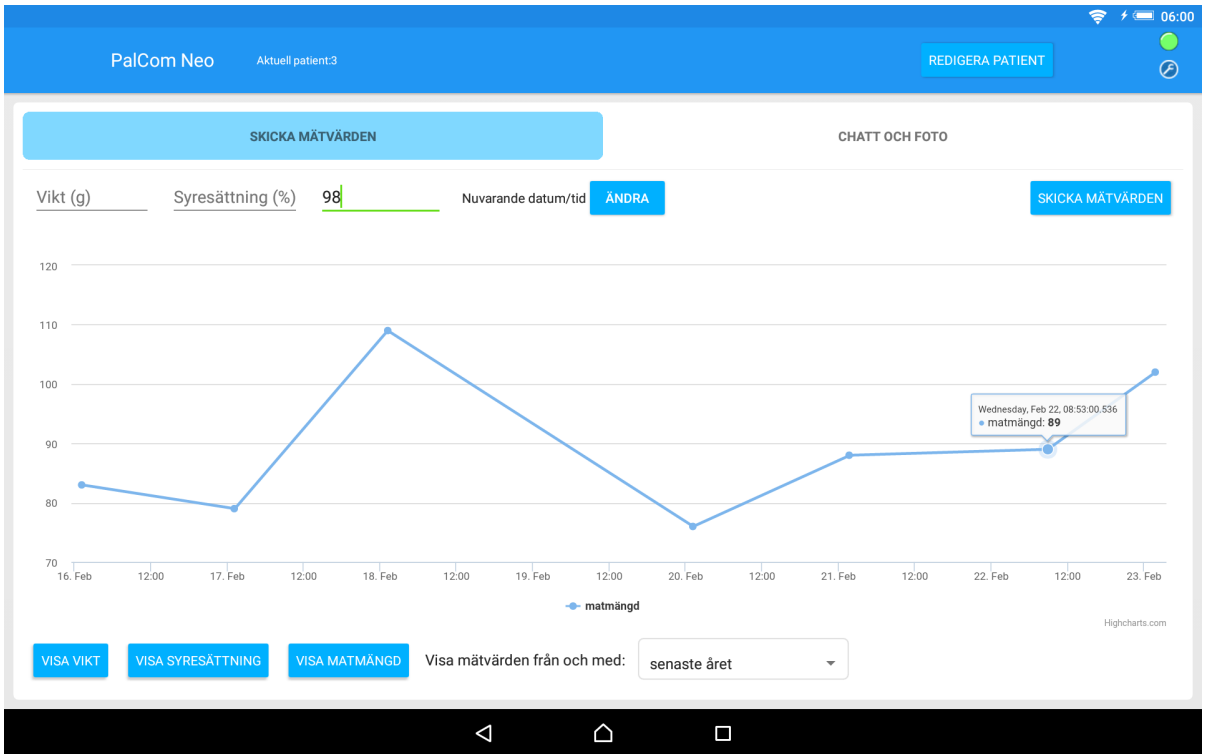


Figure 7.10: The patient GUI for the home-based neonatal care scenario. On the first (of two) views (tabs) on the main screen, new measurements are entered: 3 fields, top left (*Vikt*, etc.). Entered values are shown as diagrams (center) for observing trends, with options for filtering (bottom).

two lives: one at the hospital and one at home. When introduced to families with a hospitalized child, HBHC becomes a necessity rather than a commodity.

One example of a group that would benefit from HBHC is parents with prematurely born babies. Initially, these babies may require advanced neonatal care at the hospital. After that, however, the babies typically stay at the hospital for an extended period of time for routine observation. At this stage, it might be possible to take the child home, given a sufficiently advanced support and monitoring system. Such a system puts high demands on technical reassurances: the doctor must be comfortable with discharging the patient, and the parents must be certain that their baby will receive proper care even at home. As part of a the research project LUC3 (Lund University Child Centered Care), a prototype for a system that supports this case was developed. The system builds on the results from the itACiH project and uses PalCom for infrastructure. A web-based application is used by the doctors, and families are issued Android tablets with an application for communication and delivery of measurements. The project was funded by Forte [28].

The architecture of the patient application uses both local and distributed models, as supported by the architectural tools presented in this dissertation. It implements one-way offline support, i.e. outbound data is buffered locally and sent when the server is available, however data is not cached locally for offline viewing. The GUI was built in GPE, and is interpreted on the tablets in AndroidPUIDI. The main screen is divided into two views, represented as tabs; Figure 7.10 shows the first of these views. Here, measurements for the newborn can be entered: weight, oxygenation, and portion sizes (food). Due to technically limited (no Bluetooth) measuring devices, manual input of measurements is currently required. Previously measured values are presented as diagrams to allow for trends to be observed, with options to filter by measurement type and time period. In the second view, the user can send chat messages to the doctors. The photography screen is also accessed from View 2, allowing users to take and upload photos for the doctors to review. In cases where more direct communication is necessary, either party can initiate a video conferencing session. There is also an option to automatically respond to incoming video calls in the patient GUI. With a properly mounted tablet, this enables the doctors to observe the child sporadically, e.g. during periods where frequent observations are particularly important.

The patient application was developed in collaboration with the neonatal care unit in Lund. It has been alpha tested by a small group of patients in a “simulated home” at the hospital, in order to evaluate the system without endangering the patients. Currently, additional treatment forms are being considered.

7.2.4 Discussion

There is a common thought in all of the introduced projects: the ambition of avoiding costly face-to-face meetings between patient and doctor when these may ultimately prove unnecessary. Such meetings are a waste of scarce medical

resources (doctors) and an inconvenience to the patients – travel time and expenses, waiting time, etc. To accomplish this goal, all three projects have focused on providing more patient data to the doctors, so that informed decisions can be made. In developing the various applications to support this, other members of our research group have successfully managed to apply the contributions that we present as part of this dissertation. The practical viability of the contributions have hence been further evaluated in real-world scenarios, and in the context of real-world developers.

In all four of the Android-based applications, GPE was used to build the application GUIs. The resulting PML descriptions are interpreted in AndroidPUIDI on the tablets used by patients and doctors. Locally, the architectures of the applications have been implemented using TheAndroidThing, as a complement to TheThing on the servers. Many of the graphical components in the GUIs are highly specialized, and were developed using the API for custom PML parts. There is a certain degree of overlap in functionality between the four applications. For example, the video conferencing feature has been included in some form in all four applications. Such universal features were duplicated between applications with little effort, thus illustrating the flexibility of PalCom services and PML custom parts as reusable modules.

Regarding scalability, Table 7.1 summarizes the sizes of the four GUIs, in terms of the number of PML components and links in their descriptions. On both counts, all four GUIs are of the same order of magnitude: two (10^2). When compared to a simpler GUI such as the echo example from Chapters 5 and 6, one order of magnitude greater is observed. For the ASIH GUI, however, the order of magnitude is three (10^3). The four presented GUIs are hence not trivial in size, although bigger GUIs are possible. Note that the metrics in Table 7.1 are not necessarily representative of GUI complexity and refinement. As mentioned above, custom parts have been used extensively for advanced features. Such components can add much in terms of functionality, but add little towards the component count. As such, the metrics are only interpreted as indicate of relative size.

In the four presented projects, other researches in our group have been introduced to the contributions presented in this dissertation. The results of this

Table 7.1: Sizes of the GUIs from the related projects, in contrast to previous GUIs.

Application	Components	Links
Peritoneal dialysis	279	153
Pre-hospital care (paramedic)	342	320
Pre-hospital care (doctor)	329	350
Neonatal care	196	157
Echo example	16	6
ASIH	1575	1002

strengthen the conclusion drawn from the scalability evaluation, i.e. that PML is scalable and practically viable for building professional grade GUIs.

Chapter 8

A Controlled Experiment

As part of the evaluation efforts for the contributions presented in this dissertation, a controlled experiment was planned and executed. In the context of PalCom systems, the experiment directly compared the efficiency of the Graphical PML Editor to Android Studio. Analysis shows that the inverted GUI development approach is easy to use and effective in terms of user productivity.

8.1 Introduction

One ambition with the PalCom framework is to simplify the implementation of Internet of Things systems, thus allowing end-users with little or no programming experience to assemble systems of their own. With the contributions of this dissertation, we are making efforts towards also enabling these users to build GUIs for the systems they assemble. However, from what we have seen for PalCom, it is still mostly software developers that create these types of systems. Until the concept of end-user composition reaches mainstream adoption, we expect this to remain the case. Hence, the ambition with this experiment is to investigate whether it is possible for trained developers to effectively operate the Graphical PML Editor. We see this as a first step: in future research we want to test our solution on, and adapt it for, less capable subjects. The goal of this experiment is to investigate if the inverted GUI development approach – as implemented by the Graphical PML Editor – is more efficient than an alternative tool that uses the conventional approach. In many of the projects where PalCom has been applied, we have observed a need for system interaction from mobile devices, in particular Android tablets. Hence, Android Studio – a market leading product for developing Android applications – was selected as the tool to compare against.

8.2 Planning

In order to ensure proper execution and secure the validity of the experiment results, the execution stage of the experiment was preceded by the planning stage.

8.2.1 Goals

The object of study for the experiment is the inverted approach to GUI development provided by PML through the Graphical PML Editor (GPE). The purpose is to evaluate the performance of the new development approach in relation to the traditional one provided by Android Studio (AS), i.e. the industry standard for developing Android applications. During the evaluation, the quality focus is on the efficiency – in terms of subject productivity – of the tested tools (development approaches). Productivity is measured as the mean number of time units per task completed during the experiment. We perform the evaluation from the perspective of the researchers, to determine if the subjectively experienced performance gains of the new approach are statistically significant. The subjects of the experiment are software developers; they solve GUI development tasks in the context of a problem created specifically for this experiment. The goal of the experiment is summarized as follows:

> Analyze *the inverted GUI development approach as implemented by the Graphical PML Editor*
> for the purpose of *evaluation*
> with respect to *efficiency in terms of subject productivity*
> from the point of view of *the researchers*
> in the context of *software developers solving GUI development tasks for a designed problem*.

Based on this goal, we consider the following research questions:

RQ 8.1 Which tool (development approach) is most efficient?

RQ 8.2 Is the quality of produced solutions the same for both tools?

RQ 8.3 Do subject impressions from using one tool differ from the other?

8.2.2 Participants

The goal of the experiment was to investigate the effect of the two development approaches (treatments) on the productivity of software developers. Subjects were therefore selected from the population of engineering students at Lund University that had gotten good grades in programming courses. Even though interest in the experiment was high, it proved difficult to find a single date that matched the schedules of all applicants. Because of this, the number of considered sessions was increased. Furthermore, the type of students that were invited was broadened in order to increase the population size. Students registered their interest for participating in the experiment through an online form. All members of Code@LTH – a student driven recreational programming community – were invited. Invitations were also sent by e-mail in three waves to:

1. typical 3rd year Computer Science and Engineering (CSE) students with good grades.

2. typical 2nd year CSE students with good grades.

3. engineering students, typically 2nd year, with an interest for Computer Science courses, good grades, and experience with Android Studio.

As a last measure in response to the lack of registered applicants with Android development experience, the staff at the Department of Computer Science, Lund University, was invited to participate; two volunteered and were assigned to the AS treatment. Because of how invitations recipients were selected, all subjects had a common base of general programming knowledge. All applicants that registered were included in the experiment, to a total of 24 subjects. Any applicant that reported previous experience with Android Studio (12 in total) was selected for that treatment. The remaining 12 were assigned to the GPE treatment.

The applicants committed to a half-day of experimentation time by filling out the registration form. In doing so, they gave consent for the experiment administrators to handle their data in a confidential manner. The subjects were paid to participate in the experiment. To avoid non-serious applicants, no concrete figure was specified in the invitation, only the mention of a "symbolic" monetary compensation.

8.2.3 Experimental Material

To register interest for participating in the experiment, subjects filled out an online form. The form included questions to collect name and contact information, which of several dates the subject could attend, and whether the subject would consider using their own computer during the experiment. Furthermore, two questions covered general programming experience and Android programming experience. These questions were posed as multiple choice, both with the same four possible answers:

a. No or little prior experience.

b. Casually tested development tool(s), or novice amateur developer.

c. Educated, or advanced amateur developer.

d. Professional.

To capture impressions of the experimental material and the applied treatments, a second online form was filled out after completing the experiment. The questions of primary interest for the experiment analysis were answered on scales from 1 to 5, and in translation read as follows:

1. *"Would you consider using* [tool] *to develop Android apps in the context of PalCom systems?"*

   Absolutely not (1) (2) (3) (4) (5) Definitely

2. *"How confident are you that your submitted task solutions are compliant with the specification?"*

   Not at all confident (1) (2) (3) (4) (5) Fully confident

Further questions included whether sufficient information was provided to complete the tasks, general comments and comments on individual tasks, and whether the subject would like to take part of the results of the experiment, i.e. the paper that was eventually published as [43].

The application logic (functionality) needed to complete the tasks of the experiment was provided to the subjects on a server. The server was represented as a PalCom device (TheThing) hosting four services:

PatientService Handles patients. Has commands for getting the list of all admitted patients, and patient information for individual patients. Patients are uniquely identifiable through an integer assigned by the service.

LoginService Handles login attempts and staff members. Has commands for verifying login credentials. Staff members are identified through a unique username and a personal password.

AssessmentService Handles assessments regarding the well-being of patients. Has commands for submitting new assessments of various types. Assessments are marked with the identifier of the patient to which they belong, and the username of the staff member that created them.

ChecklistService Handles a ten-item checklist. Has commands for getting and setting the checklist. The checklist is managed separately for each patient, and is marked the username of the staff member that edited it last.

The architecture of the experiment system is illustrated in Figure 8.1. Both treatment groups worked on laptop computers to produce solutions – in the form of GUIs – to the same set of tasks. The solutions were tested on Android tablets from Sony, model Xperia Tablet Z2 (SGP521). These ran Android version 6.0.1, which was at the time the most up-to-date major release. Subjects in the AS group used Android Studio version 2.1.x when solving the tasks, producing a custom Android application that could be directly installed and tested on the tablets. In the GPE group, the Graphical PML Editor was used instead. The resulting PML descriptions were interpreted in AndroidPUIDI, thus producing the solution GUIs. All solutions connected to the same server, and used the same set of services.

The subjects started working from a *zero feature release* (ZFR) [82, Michael Hill, p. 59] specific to their respective treatments, in order to avoid having them

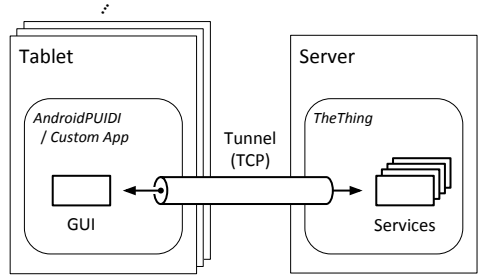


Figure 8.1: Architecture of the experiment system. GUIs built by subjects were deployed on Android tablets, and connected to services on a server over TCP tunnels and WiFi.

waste experimentation time on activities related to initial project startup. A ZFR does nothing in terms of features, but establishes the solution architecture and provides a basis to start working from. Since the subjects in the AS group should not have to bother with the intricacies regarding how their application communicates via PalCom, an "adapter" was included as part of the ZFR of that group. This adapter included methods for connecting to PalCom services, sending and receiving command, etc. Due to the PalCom integration of PML, no such measure was necessary for the GPE group.

The complete specification of the four PalCom services was listed as part of the experiment compendium [41]. Each participants was handed a printed copy of this document, which also included general experiment instructions and the description for a warm-up task. Digital experimental material was handed out on USB flash drives containing:

- Copy of compendium.
- Source file(s) for ZFR.
- PalcomBrowser for exploring the server.
- Tool for tracking development time.

For the GPE group, the flash drives also contained the executable file for GPE, and its manual [84].

8.2.4 Tasks

One warm-up task and eight proper tasks were prepared for the experiment. The purpose of the warm-up task was to allow the participants to get acquainted with their tool; development time was not recorded. The subjects started working from the zero feature release and performed tasks in order, with each task adding or changing functionality and/or design from the previous tasks. The tasks were

Table 8.1: Behavioral requirements of Task 3; builds on requirements from Tasks 1 and 2.

Req.	Description
Application	
A1	Login screen is opened upon application startup.
Login Screen	
L1	Text box for password entry masks its content, e.g. asterisks.
L2	Text boxes for username and password entry are cleared upon successful login.
L3	Main screen is opened upon successful login.
L4	List of patients is requested from server upon successful login.
Main Screen	
M1	No patient is selected when opened (successful login).
M2	Patient selection list shows patient names only (no IDs).
M3	Selecting a patient causes its information to be display.
M4	Selecting no patient clears the patient information section.
M5	Full name of logged in staff member is displayed.
M6	Logout button opens the login screen.

designed to mimic the evolution of software in real-world development projects. For the experiment problem, we drew inspiration from the results of the itACiH project, and the application that was developed to support the mobile nurses at ASIH in Lund (Chapter 7). No natural order of increasing difficulty was planned for the tasks. Instead, the tasks were allowed to organically evolve the application. However, the ambition was to have the tasks cover a wide range of typical GUI development duties (e.g. adding new screens) and graphical components (drop-down lists, popup dialogs, etc.). Furthermore, the ambition was to prepare unbiased development challenges; all tasks were solvable in both development tools.

Task descriptions were presented both in text and as images, the former primarily describing *behavioral requirements* and the latter describing *graphical requirements* in the form of screen mock-ups. The descriptions have been published in full as [41]. Table 8.1 lists the behavioral requirements of Task 3, which build upon the those of Tasks 1 and 2. The descriptions of the warm-up task and the first three tasks are presented in translation below.

Warm-up "Basic patient selection"

Mock-up. Figure 8.2.

Description. The main screen is opened when the GUI is loaded. In the text box (1), the user can type freely to select patients based on their internal ID, e.g. "1". The button (2) sends this ID to PatientService via GetPatientInfo$^{\oplus}$. If the requested patient exists, the service responds by sending PatientInfo$^{\ominus}$. The parameter of this command is shown

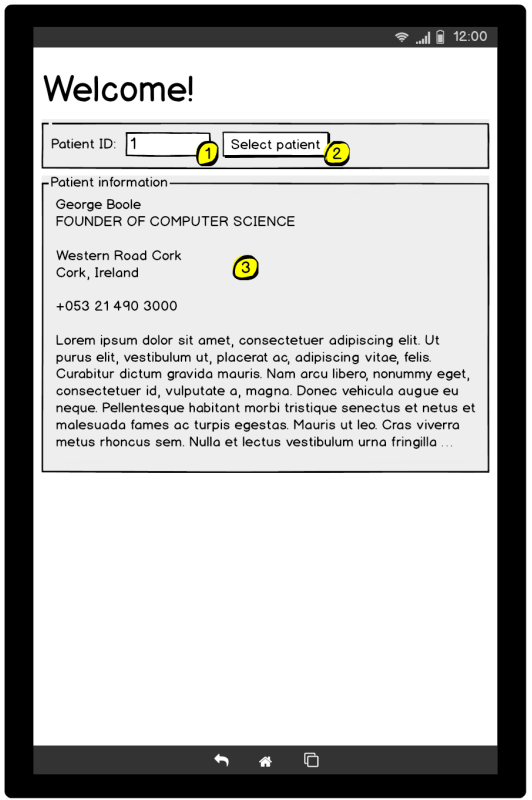


Figure 8.2: Main screen after completing the warm-up task.

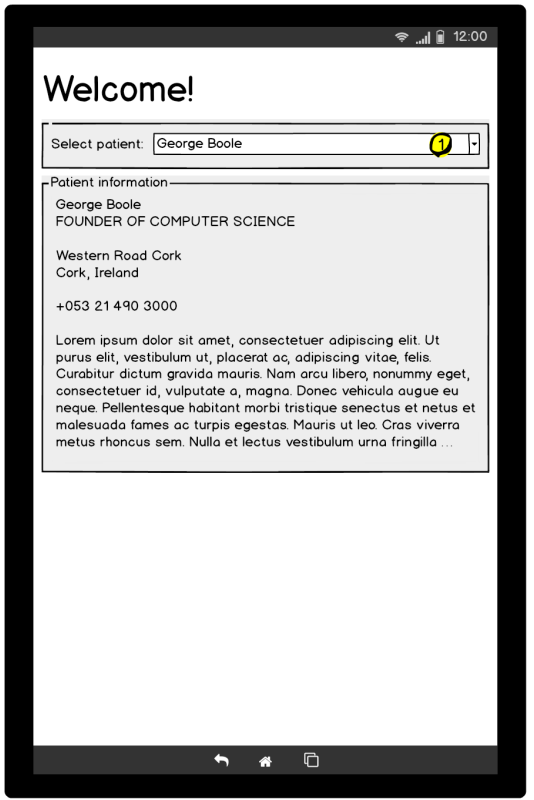


Figure 8.3: Main screen after completing Task 1.

in the text area (3) – pipe ('|') is used as line separator. If no patient exists for the given ID, the service response is NoPatientInfo$^{\leftarrow\bigcirc}$ which clears the patient information area (3).

Requirements. M4

Task 1 "Improved patient selection"

Mock-up. Figure 8.3.

Description. The drop-down list (1) shows a list of all admitted patients. The list is populated from the content of the list parameter of PatientList$^{\leftarrow\bigcirc}$. This command is sent in response to GetPatientList$^{\rightarrow\bigcirc}$, which the GUI sends to PatientService when it is loaded. PatientList::list contains both the internal ID of patients and their full name; the drop-down list should only show the latter. When a new patient is selected in the list, its information is requested and display as before.

Requirements. M2, M3

Task 2 "Basic login"

Mock-up. Figure 8.4.

Description. When the GUI is loaded, the new login screen is opened instead of the main screen. In the upper text box (1), the user enters her username, e.g. "nightingale". In the lower text box (2) the password is entered – the content is masked. The button (3) sends the input to LoginService via LogIn$^{\rightarrow\bigcirc}$. The service responds with Success$^{\leftarrow\bigcirc}$ if the credentials are valid, in which case the main screen is opened – no patient is selected. A parameter of the response contains the full name of the user. Invalid credentials are handled in a later task. On successfully logging in, the two text boxes (1, 2) are cleared, and GetPatientList$^{\rightarrow\bigcirc}$ is sent to PatientService – loading the GUI no longer sends this command.

Requirements. A1, L1, L2, L3, L4, M1

Task 3 "Clearer user identity"

Mock-up. Figure 8.5.

Description. The greeting message (1) on the main screen is more complete and personal than before: it states the name of the product, and the name of the current user, e.g. "Florence Nightingale". The message is updated upon successfully logging in. The button (2) enables the user to log out, thus re-opening the login screen; no command needs to be sent. The image box (3) decorates the main screen with the logotype of owlCARE's fictive company: Code4CARE.

Requirements. M5, M6

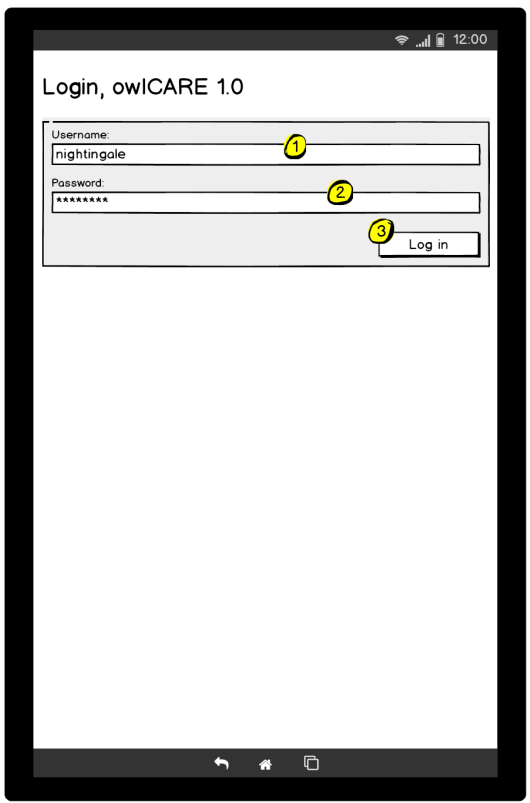


Figure 8.4: Login screen after completing Task 2.

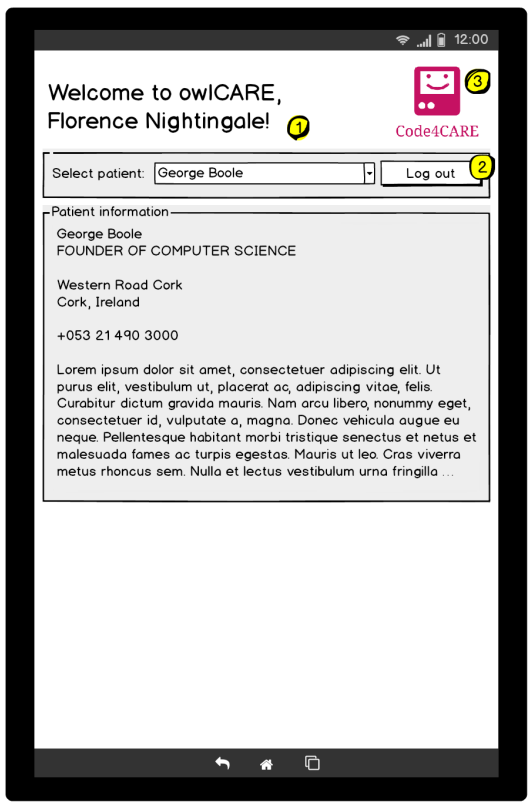


Figure 8.5: Main screen after completing Task 3.

8.2.5 Parameters and Hypotheses

The experiment has a single independent variable: GUI Development Approach (Dev.Appr.). The variable is measured on a nominal scale by categorization based on tool as either 'GPE' for the Graphical PML Editor or 'AS' for Android Studio. Additionally, the following variables are identified:

- Experience of Android development (An.Exp.) is measured on an ordinal scale (1–4) through a four level classification system. The levels correspond to the possible answers to the pre-experiment survey questions regarding programming experience (Section 8.2.3).
- General programming experience (Gen.Exp.) is measured on a scale that is analogous to that of An.Exp.

The experiment factor is Dev.Appr., which has two treatments. Gen.Exp. is used to verify subject selection, and An.Exp. is used to determine treatment assignment. The dependent variables for the experiment are:

- Tool efficiency (Tool.Ef.), measured on a ratio scale as mean development time per task, in minutes.
- Quality of produced solutions (So.Qual.), measured on a ratio scale as the mean number of specification deviations for a set of tasks. The deviations are unweighted, i.e. all deviations are considered equal in terms of severity. Two types of deviations are considered: behavioral and graphical.
- Experiment impressions (Ex.Impr.), measured on an ordinal scale (1–5) as the mean value of the answers to the two post-experiment survey questions considered for analysis.

The primary dependent variable is Tool.Ef., with So.Qual. and Ex.Impr. serving as variables to strengthen conclusions drawn from Tool.Ef.

Three hypotheses are formulated based on the research questions. Informally, we hypothesize that the mean development time of the GPE group will be less than that of the AS group (RQ 8.1). Regarding solution quality we do not expect that any treatment will outperform the other (RQ 8.2). Likewise, we expect both groups to consider using their tool, and for the confidence in solution quality to be comparable (RQ 8.3). Formally, the set of all experiment tasks is defined as $A = \{1, 2, \ldots, 8\}$. For all $a \in A'$ for some $A' \subset A$

$$\begin{aligned} H_{01} &: \mathrm{time}(\text{'GPE'}, a) \geq \mathrm{time}(\text{'AS'}, a) \\ H_{a1} &: \mathrm{time}(\text{'GPE'}, a) < \mathrm{time}(\text{'AS'}, a) \end{aligned} \tag{8.1}$$

where $\mathrm{time}(t, a)$ refers to the mean development time of task a for treatment $t \in T$ with $T = \{\text{'AS'}, \text{'GPE'}\}$. Furthermore, the elements of $D = \{\text{'B'}, \text{'G'}\}$ refer to

behavioral and graphical deviations respectively; for all $d \in D$ and some $a \in A$

$$\begin{aligned} H_{02} &: \text{bugs}(\text{'GPE'}, d, a) = \text{bugs}(\text{'AS'}, d, a) \\ H_{a2} &: \text{bugs}(\text{'GPE'}, d, a) \neq \text{bugs}(\text{'AS'}, d, a) \end{aligned} \tag{8.2}$$

where $\text{bugs}(t, d, a)$ refers to the mean number of specification deviations of type d for task a and treatment $t \in T$. Finally, for all $q \in \{1, 2\}$

$$\begin{aligned} H_{03} &: \text{answer}(\text{'GPE'}, q) = \text{answer}(\text{'AS'}, q) \\ H_{a3} &: \text{answer}(\text{'GPE'}, q) \neq \text{answer}(\text{'AS'}, q) \end{aligned} \tag{8.3}$$

where $\text{answer}(t, q)$ refers to the mean value of the answer to question q for the group with treatment $t \in T$.

8.2.6 Design

The experiment design for the stated hypotheses is of the standard type "one factor with two treatments" [90]. The experiment factor is Dev.Appr. and its treatments are GPE and AS. The main hypothesis (Hypothesis 8.1) states that development time will be lower for treatment GPE than AS, i.e. that GPE will be more efficient to use than AS. Hypotheses 8.2 and 8.3 (auxiliary) are used to strengthen the validity of the main hypothesis. The dependent variables Tool.Ef. and So.Qual. are measured on ratio scales and are tested with (non-parametric) Mann-Whitney-Wilcoxon (MWW) tests in R. We favor MWW tests over *t*-tests since we cannot assume that the measured data will follow the normal distribution. The Ex.Impr. variable is measured on an ordinal scale and is also tested with MWW tests.

8.3 Execution

Data was collected throughout six distinct experiment sessions, over a period of six weeks. Four sessions included participants for both treatments, and two covered only the AS treatment. Each session was preceded by a registration phase, and succeeded by a follow-up phase.

8.3.1 Registration

Before attending an experiment session, the subjects had to register. The link to the registration form was distributed together with the various invitations that were sent out via e-mail. By filling out the form, the applicants expressed their interest to participate in the experiment. The registration form served a double purpose: it was used to collect data regarding programming experience, i.e. independent variables An.Exp. and Gen.Exp. The data was automatically recorded in spreadsheets. Experiment subjects were selected based on these variables and other relevant data,

e.g. possible experiment dates. The selected applicants were formally invited to a specific experiment session via e-mail.

8.3.2 Experiment Session

All experiment sessions were scheduled for an afternoon. After an initial hour of introduction and training, the participants worked independently on solving tasks for the remaining three hours. This time restriction was not strictly enforced; the subjects were granted additional time to complete tasks that were started before the deadline. The sessions were monitored by one or more experiment instructors, and were held in seminar rooms at the Department of Computer Science, Lund University. Upon arrival, subjects were assigned a desk where equipment and documents had been laid out. Most subjects brought their personal laptops, while a minority borrowed laptops from the department.

The introductory part of the sessions included an overview of the experiment (goals, procedure, etc.), an introduction to the various technologies (PalCom, PML, etc.), training, and information about practical matters. Since the subjects of the GPE group had never used the Graphical PML Editor before, they were introduced to the tool in an approximately 20 minutes long practical training seminar. Additionally, the warm-up task was considered part of the training; the subjects were given the answers to any questions asked during the process of solving this task. The subjects of the AS group were expected to have an adequate amount of experience with Android Studio; no training was provided. While the GPE group was receiving training, the AS group was given an introduction to the Android Studio project of the zero feature release. Intricacies regarding the adapter that connects the project to the PalCom world were discussed. They were encouraged to ask questions about this during the warm-up task.

Figure 8.6 shows a photograph of four participants during the active phase of one of the experiment sessions. During this phase, participants worked independently on solving the provided tasks. Development was carried out on laptop computers, and testing was performed on physical Android devices (tablets). Participants were responsible for recording development times for individual tasks, i.e. data for Tool.Ef. A custom-built time tracker tool was provided for this purpose. In addition to tracking development time, the tool also ensured task ordering by making task descriptions available only after the preceding tasks had been completed; before proceeding to a new task, the time tracker forced the subjects to upload the file(s) for their current solution, i.e. data for So.Qual. The subjects could pause the time tracking in the GUI of the tracker, e.g. when going to the bathroom. The severity of forgetting to unpause was emphasized, and features of the tool were implemented to minimize this risk. The development time data and solution files were collected on USB flash drives at the end of each session.

No restrictions on material was made during the experiment sessions; subjects had free access to the Internet. When uncertain, the subjects of the AS group

Figure 8.6: Four participants during the active phase of the experiment, independently solving tasks with testing on physical devices (tablets).

would typically resort to online search engines for answers to technical questions. This was, however, not a possibility for the GPE subjects since no online resources existed for their tool. Instead, they could consult the GPE manual or ask an experiment instructor. When asked, a judgment call by the instructor was done on whether to answer the question, or refer to the manual. In general, only questions resulting from technical problems with the tool were answered.

8.3.3 Follow-up

After the conclusion of the active phase of each session, experiment impressions were collected in an online form. The answers to the questions in the form were automatically recorded in a spreadsheet; some answers were used as data for Ex.Impr. The participants were encouraged to fill out the form immediately following the experiment session, in order for the experience to still be fresh in their minds. They were allowed to be anonymous.

8.4 Analysis

The data that was collected before, during, and after the six experiment sessions was processed using a combination of custom Java programs and R, and was analyzed using both one-sided and two-sided Mann-Whitney-Wilcoxon tests.

8.4.1 Procedure

During the experiment sessions, development time data was collected by the time tracker tool, and stored on USB flash drives (one per subject) in JSON format. To analyze tool efficiency (Tool.Ef.), this data was extracted and loaded into an custom Java program. The output from this program was a comma-separated values (CSV) file, containing for each subject: subject identifier, and start, stop and total times for individual tasks. Any paused periods were excluded from the total time. The output file was processed in R. Data for the AS group was divided into tiers based on subject performance in Task 1. Hypothesis 8.1 was tested with Mann-Whitney-Wilcoxon (MWW) tests for the different treatment groups and tiers, and for individual tasks depending on data availability.

The time tracker also collected intermediate solutions for each completed task. Quality of submitted solutions (So.Qual.) was analyzed by extracting this data. For the AS group, the extracted files were compressed Android Studio projects that could be imported in Android Studio to allow for installation and review on an Android tablet. For the GPE group, the files were PML descriptions that could be installed, interpreted and reviewed on an Android tablet. Solutions from all subjects in the GPE group were reviewed, while only the solutions of the top performing AS tier were reviewed. Furthermore, only the solutions for the highest numbered

task that all subjects in this selection had completed was reviewed. Review results, per subject, were manually recorded in a spreadsheet. Behavioral requirements (Table 8.1) were scored as either passed or failed. Deviations from the graphical design (Figures 8.2 to 8.5) were counted by visual comparison. While several standard graphical deviations were identified, the comparison was ultimately made on case-by-case basis, relying on the judgment of the reviewer (the same person reviewed all solutions). The data from the spreadsheet was downloaded as a CSV file and processed in R. Hypothesis 8.2 was tested with MWW tests for the different groups and the identified highest numbered task.

The results from the post-experiment survey were automatically recorded in a spreadsheet when that subjects submitted their answers. These results were used to analyze participant experiment impressions (Ex.Impr.). The data was downloaded as a CSV file and processed in R. As for So.Qual., data for all subjects in the GPE group was included in the analysis, while only data from the top performing AS tier was considered. Hypothesis 8.3 was tested with MWW tests for the different groups and the two questions related to Ex.Impr.

8.4.2 Data Set Preparation

At the end of the experiment sessions, some participants mistakenly submitted the solution for the task they were working on. As these solution were at best partially complete, their end times are invalid. Before processing the JSON files of these subjects, the erroneous end times were manually removed. Furthermore, to avoid clutter in the graphics, unfinished tasks that the subjects had worked on for less than 15 minutes were also removed. In the form of the post-experiment survey, subjects were allowed to identify themselves; some subjects wrote their names, while others entered their assigned subject identifier. Before downloading the data from the spreadsheet where the survey results were recorded, all entries were mapped to subject identifier in order to enable proper analysis.

8.4.3 Results

Figure 8.7 gives an overview of active (pauses excluded) development time per task for all subjects, from all experiment sessions. Tasks are ordered from left to right in ascending order based on task number. Subjects are grouped based on treatment and ordered based on the development time for Task 1. The GPE group and AS group consist of Subjects 1–12 and Subjects 13–24, respectively. Subjects from the AS group are divided into three tiers based on performance:

Top tier Completed the first task in less than 60 minutes (Subjects 13–16).

Middle tier Completed the first task in more than 60 minutes (Subjects 17–20).

Bottom tier Failed to complete the first task (Subjects 21–24).

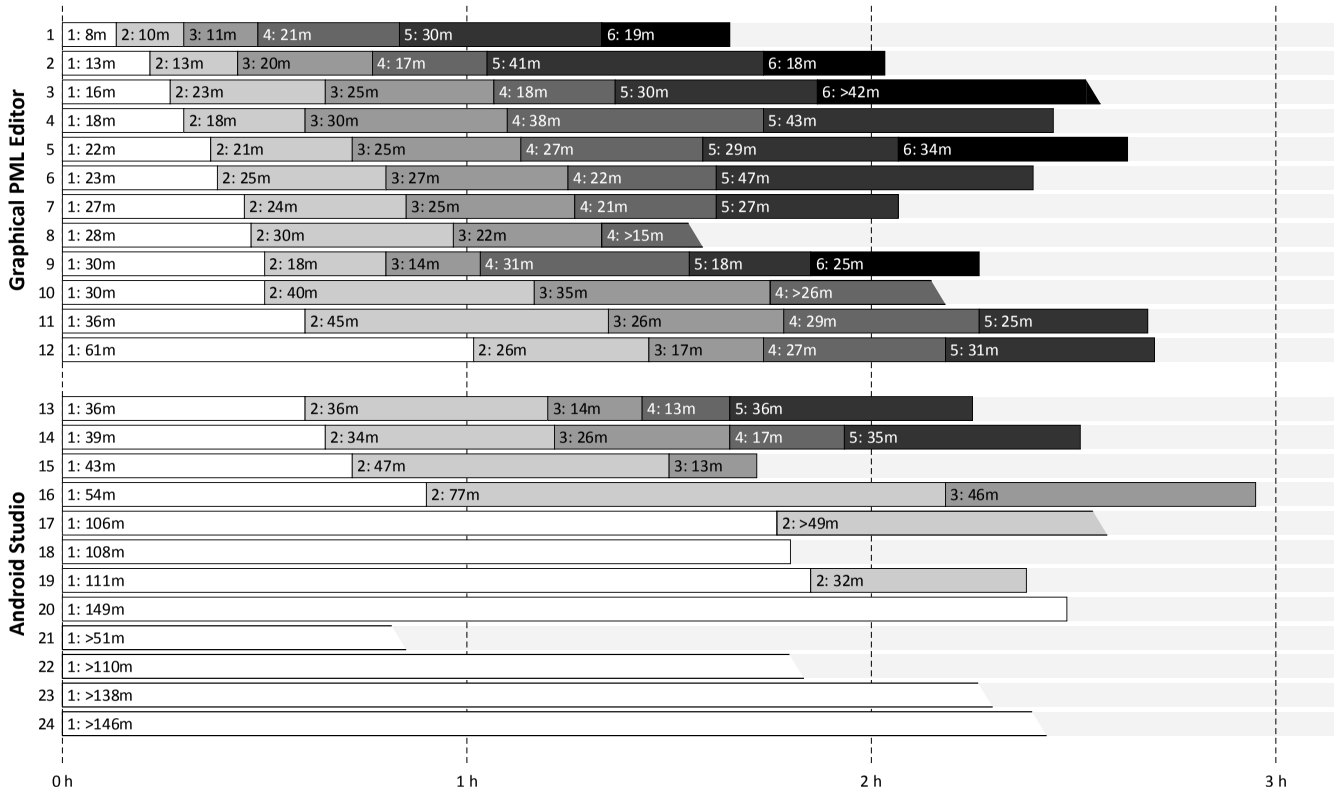


Figure 8.7: Time per task, in ascending order. Subjects grouped by treatment and ordered by time for Task 1. Incomplete tasks cropped (angle).

The results from the bottom tier cannot be analyzed, and are discarded. We proceed to analyze GPE vs. AS for the middle tier, and GPE vs. AS for the top tier separately.

For the **middle AS tier** and GPE, only development time (Tool.Ef.) is analyzed. Furthermore, only Task 1 is analyzed since too few data points are available for the other tasks and the middle tier AS group. The results for the two groups are $\text{time}(\text{'AS'}, 1) = 118.5$ minutes and $\text{time}(\text{'GPE'}, 1) = 26.0$ minutes, respectively. Comparing the groups using a one-sided Mann-Whitney-Wilcoxon (MWW) test, a p-value of 0.0021 is acquired assuming $\text{time}(\text{'GPE'}, a) < \text{time}(\text{'AS'}, a)$. The null hypothesis H_{01} is rejected for the middle tier with $A' = \{1\}$, i.e. for Task 1.

Development time, task solution quality, and the values of the answered survey questions are analyzed for the **top AS tier** and GPE. Only Tasks 1–3 are considered for the development time analysis; the results for Tasks 4–6 cannot be analyzed due to a lack of data points in the top tier AS group. The box plot in Figure 8.8 shows the data being analyzed. The results of comparing the groups using a one-sided MWW test, assuming $\text{time}(\text{'GPE'}, a) < \text{time}(\text{'AS'}, a)$, are reported in Table 8.2. The null hypothesis H_{01} is rejected for the top tier and $A' = \{1, 2\}$, but not for $A' = \{3\}$.

Task solution quality (So.Qual.) is analyzed from the data produced by manually reviewing submitted task solutions. Task 3 was selected for review, since

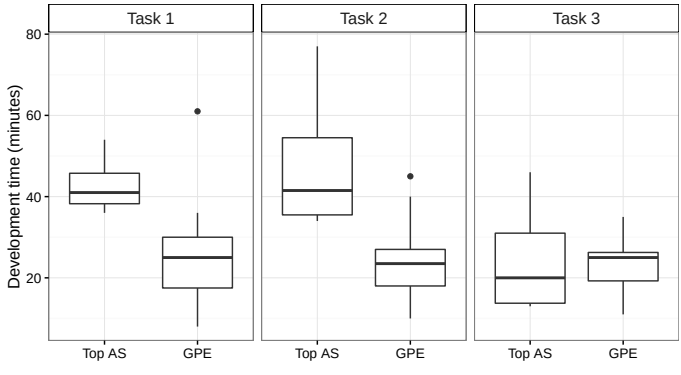


Figure 8.8: Top AS tier vs. GPE: comparison of development times for Tasks 1, 2, and 3.

Table 8.2: Top AS tier vs. GPE, one-sided MWW test (GPE<AS) for development times.

a	$\text{time}(\text{'AS'}, a)$	$\text{time}(\text{'GPE'}, a)$	p-value
1	43.0	26.0	0.0105
2	48.5	24.4	0.0090
3	24.8	23.1	0.5725

Table 8.3: Top AS tier vs. GPE: two-sided MWW test for behavioral (B) deviations.

Requirements	$\mathrm{bugs}(\text{'AS'}, \text{'B'}, 3)$	$\mathrm{bugs}(\text{'GPE'}, \text{'B'}, 3)$	p-value
$R = \langle\text{Table 8.1}\rangle$	0.250	1.08	0.0201
$R' = R \backslash \{\text{M1}\}$	0.250	0.166	0.7883

Table 8.4: Top AS tier vs. GPE: two-sided MWW test for answers to survey questions.

q	Description	$\mathrm{answer}(\text{'AS'}, q)$	$\mathrm{answer}(\text{'GPE'}, q)$	p-value
1	Would use tool	4.8	4.2	0.163
2	Confident in quality	4.2	3.8	0.140

this was the highest numbered task solved by all subjects in both the GPE group and the top tier AS group. The results of comparing the counts of behavioral (B) specification deviations using a two-sided MWW test are reported in Table 8.3. R refers to all requirements listed in Table 8.1. For the top tier AS group, $^4/_4$ submitted solutions satisfied requirement M1, while only $^1/_{12}$ solutions from the GPE group did the same. We refer to Section 8.5.1 for a discussion, and analyze for behavioral deviations again with R' defined as all requirements of R except for M1. For the number of graphical (G) specification deviations, analysis shows that $\mathrm{bugs}(\text{'AS'}, \text{'G'}, 3) = 1.75$ and $\mathrm{bugs}(\text{'GPE'}, \text{'G'}, 3) = 2.08$. Comparing the groups using a two-sided MWW test, a p-value of 0.8521 acquired. In conclusion, the null hypothesis H_{02} cannot be rejected for the top tier for either $d =$ 'B' or $d =$ 'G' when $a = 3$ and when considering R' rather than R.

Lastly, experiment impressions (Ex.Impr.) are analyzed from the data collected as part of the survey at the end of each experiment session. 100% of the participants answered the survey. Table 8.4 compares the answers of the top tier AS group and the GPE group using a two-sided MWW test. Both of the analyzed survey questions were answered on an ordinal scale of 1–5, with 1 and 5 referring to the lowest and highest levels of agreement, respectively. The mean value of the answers is calculated to provide a sense of how the subjects answered. The null hypothesis H_{03} cannot be rejected for the top tier for either Question 1 or 2.

8.5 Discussion

Based on the results from analyzing the data that was collected throughout the experiment, a discussion from a perspective of implications and threats is warranted.

8.5.1 Result Implications

Based on informal interviews with the participants of the AS group, it was concluded that the scale in the registration form for classifying An.Exp. had been

unsuccessful. Some subjects were too modest, registering in a lower category than appropriate, while others misinterpreted the classification and mistakenly put themselves in a higher category than appropriate. Because of this, the subjects in the AS group were divided into tiers based on their performance during the experiment. The data of the bottom tier was discarded, as we consider the level of Android experience for these subject to be too low to be relevant for the analysis.

We perceive the middle tier subjects as good software developers with limited Android experience. Their results are hence indicative of what happens if good non-Android developers are to create Android applications. This has relevance in practice: after a PalCom system has been created, typically some form of GUI has to be built to interact with it, not uncommonly on Android devices. As hypothesized, the mean development time for the middle AS tier was longer than for the GPE group: 118.5 vs. 26.0 minutes for Task 1. Hence, we reject H_{01} for the middle tier and Task 1. In the choice between GPE and AS, the results tell us that non-Android developers should use GPE rather than AS when creating GUIs for PalCom systems.

The subjects of the top AS tier are perceived as good developers with a fair amount of Android experience – some even had professional experience. Their results are interpreted as the expected outcome of using Android developers to create Android applications. The mean development time for the top AS tier was, for Tasks 1 and 2, longer than for the GPE group: 43.0 vs. 26.0 minutes and 48.5 vs. 24.4 minutes, respectively. Hence, H_{01} is rejected for the top tier and Tasks 1 and 2. For Task 3, however, the null hypothesis cannot be reject. From the description of Task 3, we observe that the task had a clear focus on graphical design: changing screen title, adding a logo, etc. As opposed to Tasks 1 and 2, few behavioral matters were covered. The novelty of GPE is in changing how behavior is specified, thus eliminating the need for glue code. Hence, we argue that it is reasonable for a task where little glue code would have to be written in AS to not be significantly faster to develop in GPE. It appears that even Android developers can benefit from using GPE rather than AS when creating GUIs for PalCom systems.

From analyzing the quality of the solutions for Task 3, no difference between the top tier AS group and the GPE group was found in terms of graphical specification deviations: H_{02} cannot be rejected for graphical deviations and Task 3. However, it appears that the GPE group deviated more from the behavioral specification. Looking at the data, we found that almost all GPE subjects failed to satisfy requirement M1 (Table 8.1). This cannot be a coincidence, and we do not believe that so many subjects would deliberately submit faulty solutions, hoping the infraction would go unnoticed. Instead, we believe that the subjects simply missed the requirement since it is intricate to test and hence easy to miss. The reason that no AS subject failed to satisfy the same requirement is that no additional technical effort (development time) was needed on their part; AS guarantees by default that requirement M1 is satisfied. We therefore argue that it is relevant to

test for behavioral deviations again, excluding M1. In doing so, H_{02} cannot be rejected for behavioral deviations and Task 3. In practice, it could be argued that the GPE group has produced solutions with no difference in quality compared to the top tier AS group, in less time.

For experiment impressions, no differences between the groups was found, as hypothesized: H_{03} cannot be rejected for either question. The results tell us that both groups would consider using their tool and that the subject have confidence in their submitted solutions. The latter fits well with our analysis of solution quality. The former is interesting, as the subjects of the GPE group had an agreeable experience, on average a 4.2 on a 1–5 scale.

8.5.2 Threats to Validity

Threats to the validity [90] of the results are considered in no particular order.

Instrumentation. As mentioned earlier, the scale in the registration form for classifying An.Exp. was unsuccessful due to bad instrumentation. Clearer categories to choose from should have been presented. This was solved in analysis by dividing the subject of the AS group into three tiers. Since the tasks were created specifically for the experiment, low quality in descriptions could have affected results for one or both treatments. However, in the post-experiment survey, subjects in both treatment groups reported that they had adequate information to solve the tasks.

History. Originally, the experiment was planned for just two sessions on consecutive days to minimize the effect of external factors such as environment and student schedule. To attract enough subjects, we opted for more sessions over a longer period of time. Although not formally analyzed, no single session stands out in terms of inferior results as a consequence of experiment date. Varying times of day were considered as a possible threat, and hence all the sessions were scheduled in the afternoon.

Compensatory Rivalry. In all communication with subjects (invitations, experiment introduction, etc.) we deliberately expressed ourselves in the most neutral way possible about both treatments, while at the same time not hiding the purpose of the experiment. The aim was to avoid having either group feel like the underdog, which could have affected the results.

Group Stress. A possible threat was identified in that faster subjects could have a stressful effect on slower subjects, causing more error to be made and obstructing progress. To minimize this effect, tasks were distributed digitally, one-by-one. Handing out all tasks in the beginning of the experiment was considered, but that could have compromised task ordering, e.g. by subjects accidentally skipping a task.

Mortality. Ideally, a full day of experimentation would have been preferable. This would have provided more data points, i.e. more finished tasks per participants.

However, to promote a high experiment completion rate and avoid subject fatigue, we decided on a half-day instead. More elaborate experiment designs than what was used were also considered, e.g. assigning both treatments to all participant in random order. This would have provided higher statistical significance, also at the risk of mortality.

Interaction of Selection and Treatment. Having experiment subjects that are not representative of the population that is being generalized for is a threat to external validity. The context of this experiment are software developers. While our subjects – students – are not yet fully trained industrial developers, we believe that our subject selection process has ensured enough programming experience for a valid generalization. Furthermore, in the multi-platform context of the Internet of Things, developers cannot be expected to be experts for specific platforms such as Android. Since some subjects had professional Android experience, we argue that the selection for the AS treatment is representative, and even generous in favour of the AS treatment in this regard.

Professional Developers. We cannot claim that the best possible Android developers were drafted for the experiment, not even in the top tier. If trained Android consultants had been hired, the mean development time of the AS group would probably have been shorter. However, similar logic can be applied to the GPE group: if the subject had gotten more than 20 minutes of training, mean development time would have been shorter. Even more so if experienced GPE developers had been drafted. Therefore, we believe that our comparative analysis is valid. It is however a challenge to identify representative subjects in an experiment like this.

Interaction of Setting and Treatment. A threat when using designed problems is that the experimental material might not be representative of industrial standard. This threat was handled by designing the experiment tasks to mimic a typical industry development scenario. The idea was to represent the most general use cases in order to not favor one treatment over the other. Valid setting was ensured by comparing GPE to the most up-to-date version of the market leading tool for Android development, i.e. Android Studio.

Reliability of Measurements. Since subjects self-monitored development time, there is a certain level of unreliability in the collected data. One possible threat was that the subjects would forget to "unpause" the time tracker tool, thus corrupting the data. Features in the software were developed to prevent this, and more than once per session the severity of forgetting to unpause was emphasized. Experiment instructors also checked in with participants periodically during the sessions. No incidents were reported or observed. Another threat was if subjects would deliberately pause the time tracker to improve their recorded time. However, we do not see what their motivation for doing so would be as the results are anonymous. Furthermore, an observant instructor would have caught such behavior.

8.6 Conclusions

Our analysis shows that for good software developers with limited experience of using AS, the gains in terms of developer productivity are significant when using GPE instead of AS. For developers with a fair amount of experience of using AS the gains are not as pronounced, but still present. The size of the gain also depends on the type of task being solved – GPE is more effective when performing certain tasks, but analysis is inconclusive for others. We found no statistical difference between the treatments in terms of the quality of the solutions produced during the experiment. Furthermore, we found that the GPE subjects could get started with little training, and that they would consider using our tool in the given context. We conclude that in the choice between GPE and AS, developers – even those with substantial AS experience – should consider using GPE rather than AS when creating Android GUIs for PalCom systems. In practice, this is relevant because when a PalCom system is created, typically a GUI has to be built as well, not uncommonly for Android.

Chapter 9

Future Work

We present possible future directions for this research, in no particular order.

- The presented solution has been used to create GUIs in a number research projects. From this we have found that in practice, certain sections of PML code can be reusable. One examples of this is the video conference screen that appears in several of the presented GUIs. Currently, such code sections are manually copied between description, which has resulted in difficulties relating to double maintenance [9]. In future research, we want to address this problem by introducing modularization techniques for PML descriptions. This could include more than simply creating code modules, e.g. enabling PML descriptions to be attached to services so that they can be reused – in part or completely – in other GUIs that use the services.

- As part of the presented research, we have implemented two PML interpreters: one for the widget toolkit Swing (Java) and one for Android. One possible continuation of this work would be to implement interpreters for additional platforms. In particular, we are interested in researching whether our solution can be used to simplify the development of web applications. The goal would be to eliminate the need to write complex program code manually, e.g. JavaScript.

- The Graphical PML Editor uses metadata in PalCom to make qualified suggestions of graphical components that can represent the functionality components (commands, parameters, etc.) selected by the user. Future research on improving the component suggestion algorithm could results in further improved accessibility for the editor. A starting point for this research could be to work on adding language support for PalCom Object Notation (PON) [56]. The notation allows parameters to have complex structures, and is similar to JSON in this regard. The additional metadata of PON can possibly be used for more accurate suggestions.

- With the presented research, we have made efforts towards the goal of enabling users with little or no programming experience to build GUIs

for PalCom systems. As a first step, the need to write program code was eliminated from the development process in our solution. In future work, we want to research whether this allows non-programmers to adopt our solution, or whether further efforts are needed. Furthermore, the ambition is for the non-programmers to not only adopt the solution, but to master it, thus performing on the level of software developers. To achieve this goal, we expect additional challenges in the future, e.g. in adapting the development environment to support users with other backgrounds than programming.

– In the current architecture, PML descriptions are interpreted locally on a device, by a separate interpreter application. In order to allow for more flexible system architectures, we want to do research regarding how to make the contributions of this research more deeply integrated with PalCom. Such integration could for example enable descriptions to be installed on a single central device, which could then be accessed remotely by multiple client devices that render the GUI locally. Furthermore, the current ad hoc GUIs of pre-existing PalCom tools could be replaced by PML GUIs, thus providing remote access to currently local features such as device configuration.

– The mobile manager/launcher TheAndroidThing was needed for the applications of all research projects where the contributions of this dissertation have been applied. Without it, many system architectures would not be possible. Although the tool was created as part of this research, it has not been evaluated since the focus has been on the process of developing GUIs. Some efforts have been made to evaluate aspects such as bandwidth usage [45], however more research is needed in this area.

– A current limitation of PML is the lack of native support for dynamic sets of graphical components (PML parts). A check list, for example, can only be described during development as a static number of check boxes. This limitation can be circumvented by creating custom parts. For the non-programmer, however, this is not an option. We have seen a recurring need for dynamic sets during the development of the presented applications. As such, one possible area of future research could be to investigate how to add native language support for this feature in a way that makes it both efficient and accessible to all users.

Chapter 10

Conclusions

In this dissertation we have presented research concerning the development of applications through which end-users interact with systems and devices in the context of Internet of Things. We have built the presented solution on the PalCom framework to handle the network-oriented problems of such development, and have focused our research on producing a more efficient approach for GUI development. Since we expect that more end-users will want to connect their devices in new interesting ways, we wanted to increase the number of possible GUI developers by including the end-user in the development process. Hence, we expanded the scope of PalCom to cover the construction of GUIs, a task that is generally complicated, time consuming, and requires programming.

In order to enable end-users with little or no programming experience to build GUIs, programming needed to be eliminated from the development process. To that end, we defined an "inverted" approach to GUI development. With this approach, the development focus is on presenting functionality from an application model as graphical components in a GUI, rather than on retroactively attaching functionality to manually added graphical components. The first step in supporting the new approach was to design a language for describing GUIs, and implement interpreters that communicate with PalCom services and render GUI descriptions as fully functional GUIs. In the domain of e-health, the presented language has been used in a number of research projects to build the GUIs of multiple applications; we conclude that it is practically viable for building full-blown, professional grade GUIs. In order to make the language more accessible and efficient to use we implemented a graphical editor. This tool was evaluated by direct comparison to a market leading product in a controlled experiment. We found that the editor is accessible to new users, and can be more efficient to use than the commercial alternative.

With the presented research, we have produced an efficient approach for developing GUIs for PalCom systems (RQ 1.1). Furthermore, we have made efforts towards the goal of increasing the number of possible GUI developers by eliminating the need to write program code (RQ 1.2). Although our solution does not require program code to be written, further research is needed to evaluate whether

our positive results can be generalized for non-programmers, and truly enable end-users to build GUIs for the systems they assemble.

Appendix A

Specifics of PML

The concrete syntax of PML is XML-based, as presented through examples in Chapter 5. The language is further specified by presenting the grammar of its abstract syntax and providing a specification for available PML components.

A.1 Abstract Syntax

The grammar for the abstract syntax of PML is presented in Listing A.1; it is expressed in a BNF [10] notation as summarized in Table A.1. As indicated in the grammar, PML descriptions are structured as six distinct blocks: three component blocks – universe, structure, logic – and three property blocks – style, discovery, behavior.

Universe This block specifies the PalCom devices and services that the resulting GUI needs to communicate with. For each service, the description is precise enough so that the interpreting application can send commands to specific services, act on incoming commands, and manage individual parameters.

Discovery This block specifies configuration information for the PalCom components that were specified in the universe block. For each component, the relevant discovery properties are specified, e.g. the PalCom IDs of devices.

Table A.1: Notation for the grammar in Listing A.1.

Definition	`::=`
Construction	`&`
Alternative	`|`
Repetition (zero or more)	`*`
Repetition (one or more)	`+`
Optional (zero or one)	`?`
Terminal	`name:type`

```
description ::= universe & discovery? & structure & style? & logic
  & behavior?

universe ::= device*
device ::= unit & service*
service ::= unit & command*
command ::= unit & parameter*
parameter ::= unit
unit ::= [unit-id:String] & local-discovery? & local-behavior?
local-discovery ::= property*

discovery ::= unit-property *
unit-property ::= [unit-id:String] & property

structure ::= application
application ::= part & window* & notification* & dialog* & note*
  & sound*
notification ::= part
dialog ::= part
note ::= part
sound ::= part
window ::= part & container
container ::= any-part*
any-part ::= area | radiogroup | radiobutton | tabbed | label | button
  | checkbox | textbox | image | textarea | numberslider | dropdownlist
  | custompart
area ::= part & container
tabbed ::= part & container
custompart ::= part & container?
radiogroup ::= part & radiobutton*
radiobutton ::= part
label ::= part
button ::= part
checkbox ::= part
textbox ::= part
image ::= part
textarea ::= part
numberslider ::= part
dropdownlist ::= part
part ::= [part-id:String] & local-style? & local-behavior?
local-style ::= property*

style ::= part-property*
part-property ::= [part-id:String] & property

logic ::= constant* & variable*
constant ::= fact
variable ::= fact
fact ::= [fact-id:String] & local-behavior?

fact-property ::= [fact-id:String] & property

behavior ::= ( any-property & link? )*
any-property ::= unit-property | part-property | fact-property

local-behavior ::= ( property & link? )*
link ::= [qualifier:String]? & [order:Integer]?
property ::= [key:String] & [value:String]
```

Listing A.1: Grammar for the abstract syntax of PML.

Structure This block specifies the graphical components to be presented in the resulting GUI. The structure of the GUI is specified in terms of how components are nested, e.g. a button within a tab within a window. The grammar includes a number of common parts and the possibility to include custom-made parts, and specifies how components can be nested.

Style This block specifies properties that influence the appearance of the graphical components that were specified in the structure block, e.g. the size of an image box or the text on a button.

Logic This block specifies internal components, i.e. constants and variables. These can be used as default texts in GUIs, or for advanced GUI designs where intermediate states are needed.

Behavior This block primarily specifies the actions to be performed by the interpreting application as a result of the user interacting with the GUI and incoming commands from the connected services.

The essence of PML, i.e. what makes the inverted GUI development approach possible, is that the grammar of the language contains both an abstraction of the application model (universe block) and an abstraction of the GUI (structure block). This makes it possible to link components from both abstractions to define GUI behavior (behavior block), thus eliminating the need to write program code.

A.2 Component Specification

Each PML components in the universe, structure, and logic blocks can be further specified by their properties. For example, a graphical component such as a button has properties (style block) to control its appearance: text, font, size, etc. Each component class has a set of valid properties, the keys of which are listed, but not explained in detail, in Table A.2.

Behavior properties (behaviour block) apply to all types of PML components, and control how they interact with other components. These properties are used to specify links between source components and target components. For example, a link between a button and a PalCom command could cause the command to be sent to its service when the button is clicked. In this example, the button is the source, the command is the target and the role of the link is to invoke the command, i.e. send it to its service. “Clicked” is an example of a qualifier; qualifiers are used to specialize links, in this case to specify when the command should be invoked.

Properties that define links are only valid for the property keys of the four link roles: “p:invoker”, “p:reactor”, “p:viewer”, and “p:provider”. The set of available qualifiers depends on the role of the link and the class of the source component; in Table A.2, the available qualifiers are listed in parentheses for the property keys of links. Furthermore, only certain component classes are valid targets for links

of specific roles; for each class and valid link role, target behavior is presented in Table A.2 as:

- the *action* to be taken when invoked.
- the *event* that other components react to.
- the *property* that other components view.
- the *property* that is set to provided values.

The row is omitted in the table for classes with no link target behavior.

Table A.2: Specification for all available PML component classes.

Class	Description, property keys, link target behavior
	UNITS
P:Device	*PalCom device.*
	Discovery: p:id.
P:Service	*PalCom service.*
	Discovery: p:required, p:alias, p:instance, p:cdid, p:cn, p:udid, p:un, p:pdid, p:pn, p:mdid, p:mn. Behavior: p:invoker (available, unavailable).
P:Command	*PalCom command.*
	Discovery: p:id, p:direction. Behavior: p:invoker (received).
	When invoked *send*, react to *received*.
P:Param	*PalCom parameter.*
	Discovery: p:id, p:data-type.
	React to *updated/received*, viewed for *data*, provided sets *data*.
	PARTS
G:Application	*Logical component that represents the interpreting application internally.*
	Behavior: p:invoker (loaded, unloaded).
G:Window	*Top-level, self-contained, structural component. May contain other components.*
	Style: g:layout, g:layout-gap, g:layout-columns, g:layout-weights, g:layout-orientation, g:title, g:resizable, g:size. Behavior: p:invoker (opened, closed).
	When invoked *open*.
G:Area	*Structural component. Primary purpose is to contain and layout other components.*
	Style: g:layout, g:layout-gap, g:layout-columns, g:layout-weights, g:layout-orientation, g:title, g:scrollable-v, g:scrollable-h, g:border, g:tab-text, g:size. Behavior: p:viewer (present), p:present.
G:Tabbed	*Structural component. Contained components are represented as tabs, and displayed after selection.*
	Style: g:tab-text, g:size. Behavior: p:viewer (selected-index, enabled).

↪

Table A.2: (continued)

G:RadioGroup	*Logical component that groups radio buttons.*
	Style: g:layout-gap, g:layout-orientation, g:scrollable-v, g:scrollable-h, g:border, g:tab-text, g:size. Behavior: p:viewer (selected-value, present), p:provider (selected-value), p:present.
G:RadioButton	*Clickable, two-state component. One instance per group can be checked.*
	Style: g:tab-text, g:size, g:font, g:font-bold, g:font-italic, g:font-size, g:font-color, g:text. Behavior: p:value.
G:Label	*Component for displaying simple text.*
	Style: g:tab-text, g:size, g:font, g:font-bold, g:font-italic, g:font-size, g:font-color, g:align-h, g:align-v, g:text. Behavior: p:viewer (text, present, font-color).
G:Button	*Clickable component.*
	Style: g:tab-text, g:size, g:font, g:font-bold, g:font-italic, g:font-size, g:font-color, g:text, g:alternate-text, g:image-src. Behavior: p:invoker (clicked), p:provider (browse, camera), p:viewer (enabled), p:enabled.
G:CheckBox	*Clickable, two-state component. Multiple instances can be checked.*
	Style: g:tab-text, g:size, g:font, g:font-bold, g:font-italic, g:font-size, g:font-color, g:text, g:alternate-text. Behavior: p:invoker (checked-changed), p:provider (checked-value), p:viewer.
G:TextField	*Component for text input.*
	Style: g:tab-text, g:size, g:text, g:alternate-text, g:align-h, g:sensitive, g:single-line. Behavior: p:editable, p:provider (text), p:viewer (text, enabled).
G:Image	*Component for displaying images.*
	Style: g:tab-text, g:size, g:image-src. Behavior: p:provider (image), p:viewer (image, mask), p:editable.
G:TextArea	*Component for displaying text (advanced).*
	Style: g:tab-text, g:size, g:font, g:font-bold, g:font-italic, g:font-size, g:font-color, g:text. Behavior: p:viewer (text, present, font-color), p:delimiter.
G:NumberSlider	*Component with horizontally slidable handle. Selects numeric value in given range.*
	Style: g:tab-text, g:size. Behavior: p:invoker (released), p:provider (value), p:viewer (value), p:editable, p:min-value, p:max-value.

↪

Table A.2: (continued)

G:DropDownList	*Compact (expandable) component for item selection from a list.*
	Style: g:tab-text, g:size, g:content. Behavior: p:invoker (selected), p:provider (selected-value), p:viewer (content, selected-index, selected-value), p:delimiter, p:row-length, p:option-index, p:value-index, p:editable.
G:SystemNotification	*System wide notification. Visible even when the application is not.*
	Style: g:title, g:text. Behavior: p:reactor (post), p:viewer (enabled), p:priority, p:enabled.
G:YesNoDialog	*Dialog box that may block the rest of the application until confirmed/dismissed.*
	Style: g:title, g:text, g:text-positive, g:text-negative, g:cancelable. Behavior: p:reactor (show, dismiss), p:invoker (positive, negative), p:viewer (enabled), p:provider (input-text), p:enabled.
	When invoked *show*.
G:QuickNote	*Temporarily visible text message.*
	Style: g:title, g:text. Behavior: p:reactor (show), p:enabled, p:viewer (enabled).
	When invoked *show*.
G:Sound	*Plays a notification sound.*
	Behavior: p:sound-src, p:reactor (toggle-play, play, stop), p:enabled, p:viewer (enabled).
	When invoked *play*.

FACTS

G:Constant	*Holds a constant value.*
	Behavior: p:provider (data), p:reactor (announce), p:value, p:init-invoke.
	When invoked *announce*, react to *announcement*, viewed for *value*.
G:Variable	*Holds a value that can be changed.*
	Behavior: p:provider (value), p:reactor (announce), p:value, p:init-invoke, p:auto-invoke, p:viewer (value).
	When invoked *announce*, react to *announcement*, viewed for *value*, provided sets *value*.

Bibliography

[1] Peter Abrahamsson. “Medical Accelerometor Smartphone Application for PalCom Based Systems”. LU-CS-EX:2013-02. ISSN: 1650-2884. MA thesis. Department of Computer Science, Lund University, 2013.

[2] Marc Abrams et al. “UIML: an appliance-independent XML user interface language”. In: *Computer Networks* 31 (11–16), 1999, pp. 1695–1708. ISSN: 1389-1286. DOI: 10.1016/S1389-1286(99)00044-4.

[3] André Ahlfors Dahl and Martin Larsson. “Punkt-till-punkt-autentisering för Sakernas Internet”. LU-CS-EX:2014-38. ISSN: 1650-2884. MA thesis. Department of Computer Science, Lund University, 2014.

[4] Peter Andersen et al. “Open architecture for palpable computing: Some thoughts on object technology, palpable computing, and architectures for ambient computing”. In: *In Proceedings of the Workshop on Object Technology for Ambient Intelligence (OT4AmI) at the European Conference on Object-Oriented Programming (ECOOP 2005), 2005. 238 Troels L. Andersen, Sune Kristensen, Bjørn*. ACM, 2004, pp. 137–138.

[5] Apache Isis. *Domain Driven Applications, Quickly*. The Apache Software Foundation. URL: http://isis.apache.org (visited on Apr. 3, 2017).

[6] AppMaker. *Low-code application development for G Suite*. Google Inc. URL: https://developers.google.com/appmaker/ (visited on Apr. 3, 2017).

[7] Kevin Ashton. “That ‘Internet of Things’ Thing”. In: *RFiD Journal* 22 (7), 2009, pp. 97–114.

[8] Pedro Azevedo, Roland Merrick, and Dave Roberts. “OVID to AUIML - User-Oriented Interface Modelling”. In: *Proceedings of 1st International Workshop – Towards a UML Profile for Interactive Systems Development*. TUPIS ’00. York, UK, Oct. 2000.

[9] Wayne A. Babich. *Software Configuration Management: Coordination for Team Productivity*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-10161-0.

[10] J. W. Backus et al. “Report on the algorithmic language ALGOL 60”. In: *Numerische Mathematik* 2 (1), 1960, pp. 106–136. ISSN: 0945-3245. DOI: 10.1007/BF01386216.

[11] Roberto Baldoni, Carlo Marchetti, and Alessandro Termini. "Active Software Replication through a Three-tier Approach". In: *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.* 2002, pp. 109–118. DOI: 10.1109/RELDIS.2002.1180179.

[12] Guruduth Banavar et al. "Challenges: An Application Model for Pervasive Computing". In: *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*. MobiCom '00. Boston, Massachusetts, USA: ACM, 2000, pp. 266–274. ISBN: 1-58113-197-6. DOI: 10.1145/345910.345957.

[13] Soma Bandyopadhyay et al. "A Survey of Middleware for Internet of Things". In: *Proceedings of Recent Trends in Wireless and Mobile Networks: Third International Conferences, WiMo 2011 and CoNeCo 2011, Ankara, Turkey, June 26-28, 2011*. Ed. by Abdulkadir Özcan, Jan Zizka, and Dhinaharan Nagamalai. Springer Berlin Heidelberg, 2011, pp. 288–296. ISBN: 978-3-642-21937-5. DOI: 10.1007/978-3-642-21937-5_27.

[14] David J. Barnes and Michael Kölling. *Objects First with Java: A Practical Introduction Using BlueJ*. 6th ed. Pearson, 2016. ISBN: 978-1-292-15904-1.

[15] Olle Bergman, Micke Jaresand, and Elizabeth Johansson. *Cancerfondsrapporten 2010*. Ed. by Micke Jaresand. 2010. ISBN: 978-91-978853-0-0.

[16] Olle Bergman et al. *Cancerfondsrapporten 2016*. Ed. by Lisa Jacobson. 2016. ISBN: 978-91-88161-03-1.

[17] Tim Boudreau et al. *NetBeans: The Definitive Guide*. O'Reilly Media, 2002. ISBN: 978-0-596-00280-0.

[18] Anna Lena Brantberg and Renée Allvin. *Smärtskattningsinstrument: Smärtskattning av akut och postoperativ smärta*. Aug. 23, 2016. URL: http://www.vardhandboken.se/Texter/Smartskattning-av-akut-och-postoperativ-smarta/Smartskattningsinstrument/ (visited on Apr. 3, 2017).

[19] F. P. J. Brooks. "No Silver Bullet Essence and Accidents of Software Engineering". In: *Computer* 20 (4), 1987, pp. 10–19. ISSN: 0018-9162. DOI: 10.1109/MC.1987.1663532.

[20] E Bruera et al. "The Edmonton Symptom Assessment System (ESAS): a simple method for the assessment of palliative care patients." In: *Journal of Palliative Care* 2 (7), 1991, pp. 6–9.

[21] A. T. Campbell, G. Coulson, and M. E. Kounavis. "Managing Complexity: Middleware Explained". In: *IT Professional* 1 (5), Sept. 1999, pp. 22–28. ISSN: 1520-9202. DOI: 10.1109/6294.793667.

[22] M. A. Chaqfeh and N. Mohamed. “Challenges in Middleware Solutions for the Internet of Things”. In: *2012 International Conference on Collaboration Technologies and Systems (CTS)*. May 2012, pp. 21–26. DOI: 10.1109/CTS.2012.6261022.

[23] Community Research and Development Information Service (CORDIS). *Palpable computing - A new perspective on ambient computing*. Research project. Grant 002057. 2004–2007. URL: http://cordis.europa.eu.

[24] George F Coulouris et al. *Distributed Systems: Concepts and Design*. 5th ed. Pearson, 2012. ISBN: 978-0-13-214301-1.

[25] Robert Englander. *Developing JAVA Beans*. O’Reilly Media, 1997. ISBN: 978-1-565-92289-1.

[26] Davic Everlöf and Thomas Lidén. “Integrated video solution for telemedicine in home care”. LU-CS-EX:2013-41. ISSN: 1650-2884. MA thesis. Department of Computer Science, Lund University, 2013.

[27] Niklas Fors. “The Design and Implementation of Bloqqi – A Feature-Based Diagram Programming Language”. LU-CS-DISS:2016-5. ISSN: 1404-1219. PhD thesis. Department of Computer Science, Lund University, 2016. ISBN: 978-91-7623-999-5.

[28] Forte. *Programme Grant Care: Family support and Child Centered Care for vulnerable children – knowledge development and translation to care service*. Research project. Grant 2013-2101. 2013–2015. URL: http://forte.se/en/.

[29] Fabiano Freitas and Paulo Henrique M. Maia. “JustBusiness: A Framework for Developing Android Applications Using Naked Objects”. In: *2015 IX Brazilian Symposium on Components, Architectures and Reuse Software*. Sept. 2015, pp. 11–20. DOI: 10.1109/SBCARS.2015.12.

[30] Tony Gaddis and Rebecca Halsey. *Starting Out with App Inventor for Android*. Pearson Education, 2015. ISBN: 978-1-292-08032-1.

[31] Daniel Giusto et al. *The Internet of Things: 20th Tyrrhenian Workshop on Digital Communications*. Springer New York, 2010. ISBN: 978-1-4419-1674-7. DOI: 10.1007/978-1-4419-1674-7.

[32] Joan Greenbaum and Morten Kyng. *Design at Work: Cooperative Design of Computer Systems*. L. Erlbaum Associates Inc., 1992. ISBN: 0-8058-0612-1.

[33] Jayavardhana Gubbi et al. “Internet of Things (IoT): A vision, architectural elements, and future directions”. In: *Future Generation Computer Systems* 29 (7), 2013, pp. 1645–1660. ISSN: 0167-739X. DOI: 10.1016/j.future.2013.01.010.

[34] J. Guerrero-Garcia et al. “A Theoretical Survey of User Interface Description Languages: Preliminary Results”. In: *2009 Latin American Web Congress*. Nov. 2009, pp. 36–43. DOI: 10.1109/LA-WEB.2009.40.

[35] Jens Gustafsson and Alfred Åkesson. “System for Activity Tracking of Patients with Chronic Kidney Disease”. LU-CS-EX:2015-35. ISSN: 1650-2884. MA thesis. Department of Computer Science, Lund University, 2015.

[36] Michi Henning. “The Rise and Fall of CORBA”. In: *Queue* 4 (5), June 2006, pp. 28–34. ISSN: 1542-7730. DOI: 10.1145/1142031.1142044.

[37] itACiH AB. *Vi utvecklar lösningar för att fler skall kunna vårdas hemma*. URL: http://itacih.se (visited on Apr. 3, 2017).

[38] Paul Johannesson and Erik Perjons. *An Introduction to Design Science*. Springer International Publishing, 2014. ISBN: 978-3-319-10631-1. DOI: 10.1007/978-3-319-10632-8.

[39] Bruce Johnson. *Professional Visual Studio 2015*. Wrox, 2015. ISBN: 978-1-119-06805-1.

[40] Björn A. Johnsson. “PalCom Meets the End-User: Enabling Interaction with PalCom-based Systems”. LU-CS-DISS:2014-2. ISSN: 1652-4691. Licentiate thesis. Department of Computer Science, Lund University, 2014.

[41] Björn A. Johnsson. *Tasks and Instructions for a Controlled Experiment on IoT GUI Development*. Dec. 2016. DOI: 10.5281/zenodo.163512.

[42] Björn A. Johnsson. “Towards End-User Composition of Graphical User Interfaces for Internet of Things”. In: *"Future Generation Computer Systems"*, 2016. Submitted.

[43] Björn A. Johnsson, Martin Höst, and Boris Magnusson. “Evaluating a GUI Development Tool for Internet of Things and Android”. In: *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings*. Ed. by Pekka Abrahamsson et al. Cham: Springer International Publishing, 2016, pp. 181–197. ISBN: 978-3-319-49094-6. DOI: 10.1007/978-3-319-49094-6_12.

[44] Björn A. Johnsson and Boris Magnusson. “Supporting Collaborative Healthcare using PalCom – The itACiH System”. In: *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. Sydney, Australia: IEEE, Mar. 2016, pp. 1–6. DOI: 10.1109/PERCOMW.2016.7457141.

[45] Björn A. Johnsson, Mattias Nordahl, and Boris Magnusson. “Evaluating a Dynamic Keep-Alive Messaging Strategy for Mobile Pervasive Systems”. In: *Procedia Computer Science*, 2017. The 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017), Madeira, Portugal. In press.

[46] Björn A. Johnsson and Gunnar Weibull. “End-User Composition of Graphical User Interfaces for PalCom Systems”. In: *Procedia Computer Science* 94, 2016. The 11th International Conference on Future Networks and Communications (FNC 2016), Montreal, Quebec, Canada, pp. 224–231. ISSN: 1877-0509. DOI: 10.1016/j.procs.2016.08.035.

[47] Caitlin Kelleher and Randy Pausch. “Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers”. In: *ACM Computing Surveys* 37 (2), June 2005, pp. 83–137. ISSN: 0360-0300. DOI: 10.1145/1089733.1089734.

[48] Boris Magnusson and Björn A. Johnsson. “Some Like It Hot: Automating an Electric Kettle Using PalCom”. In: *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*. UbiComp ’13 Adjunct. Zurich, Switzerland: ACM, 2013, pp. 63–66. ISBN: 978-1-4503-2215-7. DOI: 10.1145/2494091.2494110.

[49] Rob van der Meulen. *Gartner Says 6.4 Billion Connected “Things” Will Be in Use in 2016, Up 30 Percent From 2015*. Press release. Nov. 10, 2015. URL: http://www.gartner.com/newsroom/id/3165317 (visited on Apr. 3, 2017).

[50] Daniele Miorandi et al. “Internet of things: Vision, applications and research challenges”. In: *Ad Hoc Networks* 10 (7), 2012, pp. 1497–1516. ISSN: 1570-8705. DOI: 10.1016/j.adhoc.2012.02.016.

[51] Elissa Murphy. *Customize your G Suite experience with App Maker and Recommended apps*. Blog post. Nov. 30, 2016. URL: https://www.blog.google/products/g-suite/customize-your-g-suite-experience-app-maker-and-recommended-apps/ (visited on Apr. 3, 2017).

[52] Brad A. Myers. “Taxonomies of Visual Programming and Program Visualization”. In: *Journal of Visual Languages & Computing* 1 (1), 1990, pp. 97–123. ISSN: 1045-926X. DOI: 10.1016/S1045-926X(05)80036-9.

[53] Brad A. Myers. *Why are human-computer interfaces difficult to design and implement*. Tech. rep. CMU-CS-93-183. Computer Science Department, Carnegie Mellon University, July 1993.

[54] Matt Neuburg. *iOS 10 Programming Fundamentals with Swift: Swift, Xcode, and Cocoa Basics*. O’Reilly Media, 2016. ISBN: 978-1-4919-7007-2.

[55] Daniel Nilsson and Mattias Nordahl. “Minimal Implementation av PalCom för Små Enheter”. LU-CS-EX:2013-07. ISSN: 1650-2884. MA thesis. Department of Computer Science, Lund University, 2013.

[56] Mattias Nordahl and Boris Magnusson. “A lightweight data interchange format for internet of things with applications in the PalCom middleware framework”. In: *Journal of Ambient Intelligence and Humanized Computing* 7 (4), 2016, pp. 523–532. ISSN: 1868-5145. DOI: 10.1007/s12652-016-0382-3.

[57] Richard Pawson. “Naked objects”. PhD thesis. Department of Computer Science, Trinity College, University of Dublin, June 2004.

[58] Esmond Pitt and Kathy McNiff. *Java.Rmi: The Remote Method Invocation Guide*. Addison-Wesley, 2001. ISBN: 978-0-201-70043-5.

[59] Angel Puerta and Jacob Eisenstein. “XIML: A Common Representation for Interaction Data”. In: *Proceedings of the 7th International Conference on Intelligent User Interfaces*. IUI ’02. San Francisco, California, USA: ACM, 2002, pp. 214–215. ISBN: 1-58113-459-2. DOI: 10.1145/502716.502763.

[60] Aruna Raja and Devika Lakshmanan. “Naked Objects Framework”. In: *International Journal of Computer Applications* 1 (20), 2010.

[61] Mitchel Resnick et al. “Scratch: Programming for All”. In: *Communications of the ACM* 52 (11), Nov. 2009, pp. 60–67. ISSN: 0001-0782. DOI: 10.1145/1592761.1592779.

[62] Giovanni Rimassa, Dominic Greenwood, and Monique Calisti. “Palpable Computing and the Role of Agent Technology”. In: *Multi-Agent Systems and Applications IV: 4th International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2005, Budapest, Hungary, September 15 – 17, 2005. Proceedings*. Ed. by Michael Pěchouček, Paolo Petta, and László Zsolt Varga. Springer Berlin Heidelberg, 2005, pp. 1–10. ISBN: 978-3-540-31731-9. DOI: 10.1007/11559221_1.

[63] Ray Rischpater. *Application Development with Qt Creator*. Packt Publishing Limited, 2013. ISBN: 978-1-7832-8231-9.

[64] Luis Roalter, Matthias Kranz, and Andreas Möller. “A Middleware for Intelligent Environments and the Internet of Things”. In: *Ubiquitous Intelligence and Computing: 7th International Conference, UIC 2010, Xi’an, China, October 26-29, 2010. Proceedings*. Ed. by Zhiwen Yu et al. Springer Berlin Heidelberg, 2010, pp. 267–281. ISBN: 978-3-642-16355-5. DOI: 10.1007/978-3-642-16355-5_23.

[65] Yvonne Rogers. “Moving on from Weiser’s Vision of Calm Computing: Engaging UbiComp Experiences”. In: *UbiComp 2006: Ubiquitous Computing: 8th International Conference, UbiComp 2006 Orange County, CA, USA, September 17-21, 2006 Proceedings*. Ed. by Paul Dourish and Adrian Friday. Springer Berlin Heidelberg, 2006, pp. 404–421. ISBN: 978-3-540-39635-2. DOI: 10.1007/11853565_24.

[66] S. M. Sadjadi, P. K. McKinley, and E. P. Kasten. “Architecture and Operation of an Adaptable Communication Substrate”. In: *The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems, 2003. FTDCS 2003. Proceedings.* May 2003, pp. 46–55. DOI: 10.1109/FTDCS.2003.1204293.

[67] Seyed Masoud Sadjadi and Philip K McKinley. *A Survey of Adaptive Middleware*. Tech. rep. MSU-CSE-03-35. Michigan State University, 2003.

[68] D. Saha and A. Mukherjee. “Pervasive Computing: a Paradigm for the 21st Century”. In: *Computer* 36 (3), Mar. 2003, pp. 25–31. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1185214.

[69] Thomas Sandholm, Boris Magnusson, and Björn A. Johnsson. “An On-Demand WebRTC and IoT Device Tunneling Service for Hospitals”. In: *2014 International Conference on Future Internet of Things and Cloud*. Barcelona, Spain, Aug. 2014, pp. 53–60. DOI: 10.1109/FiCloud.2014.19.

[70] M. Satyanarayanan. “Fundamental Challenges in Mobile Computing”. In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC ’96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 1–7. ISBN: 0-89791-800-2. DOI: 10.1145/248052.248053.

[71] M. Satyanarayanan. “Pervasive Computing: Vision and Challenges”. In: *IEEE Personal Communications* 8 (4), Aug. 2001, pp. 10–17. ISSN: 1070-9916. DOI: 10.1109/98.943998.

[72] Douglas C. Schmidt. “Middleware for Real-time and Embedded Systems”. In: *Communications of the ACM* 45 (6), June 2002, pp. 43–48. ISSN: 0001-0782. DOI: 10.1145/508448.508472.

[73] D. C. Sharp. “Reducing Avionics Software Cost through Component Based Product Line Development”. In: *17th DASC. AIAA/IEEE/SAE. Digital Avionics Systems Conference. Proceedings (Cat. No.98CH36267)*. Vol. 2. Oct. 1998, G32/1–G32/8. DOI: 10.1109/DASC.1998.739846.

[74] Neil Smyth. *Android Studio Development Essentials - Android 6 Edition*. CreateSpace Independent Publishing Platform, 2015. ISBN: 978-1-5197-2208-9.

[75] Nathalie Souchon and Jean Vanderdonckt. “A Review of XML-compliant User Interface Description Languages”. In: *Interactive Systems. Design, Specification, and Verification: 10th International Workshop, DSV-IS 2003, Funchal, Madeira Island, Portugal, June 11-13, 2003. Revised Papers*. Ed. by Joaquim A. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha. Springer Berlin Heidelberg, 2003, pp. 377–391. ISBN: 978-3-540-39929-2. DOI: 10.1007/978-3-540-39929-2_26.

[76] Bernard W. Stewart and Christopher P. Wild. *World Cancer Report 2014*. International Agency for Research on Cancer. IARC Press, 2014. ISBN: 978-92-832-0429-9.

[77] Swedish Foundation for Strategic Research (SSF). *ENGROSS: ENabling GROwing Software Systems*. Research project. Grant RIT08-0075. 2009–2013. URL: http://stratresearch.se/en/.

[78] Harald Sundmaeker et al. *Vision and Challenges for Realising the Internet of Things*. Publications Office of the European Union, 2010. ISBN: 978-92-79-15088-3. DOI: 10.2759/26127.

[79] David Svensson Fors. "Assemblies of Pervasive Services". LU-CS-DISS: 2009-1. ISSN: 1404-1219. PhD thesis. Department of Computer Science, Lund University, 2009. ISBN: 978-91-976939-1-2.

[80] Vinnova. *Innovative Technology for the Future's Emergency Medical Care*. Research project. Grant 2015-00382. 2015–2017. URL: http://www.vinnova.se/en/.

[81] Vinnova. *IT-stöd för avancerad cancervård i hemmet*. Research project. Grants 2011-02796 and 2013-04876. 2011–2016. URL: http://www.vinnova.se/en/.

[82] William C. Wake. *Extreme Programming Explored*. The XP Series. Addison-Wesley, 2002. ISBN: 0-201-73397-8.

[83] Gunnar Weibull. "Graphical Editor for Graphical User Interfaces for an "Internet of Things" System". LU-CS-EX:2015-32. ISSN: 1650-2884. MA thesis. Department of Computer Science, Lund University, 2015.

[84] Gunnar Weibull, Boris Magnusson, and Björn A. Johnsson. *The PML Editor: User's Manual*. Tech. rep. 102. LU-CS-TR:2017-253. ISSN: 1404-1200. Department of Computer Science, Lund University, 2017, p. 77.

[85] Mark Weiser. "Hot Topics: Ubiquitous computing". In: *Computer* 26 (10), Oct. 1993, pp. 71–72. ISSN: 0018-9162. DOI: 10.1109/2.237456.

[86] Mark Weiser. "Some Computer Science Issues in Ubiquitous Computing". In: *Communications of the ACM* 36 (7), July 1993, pp. 75–84. ISSN: 0001-0782. DOI: 10.1145/159544.159617.

[87] Mark Weiser. "The Computer for the 21st Century". In: *Scientific American* 265 (3), 1991, pp. 94–104.

[88] Mark Weiser and John Seely Brown. "The Coming Age of Calm Technology". In: *Beyond Calculation: The Next Fifty Years of Computing*. Springer New York, 1997, pp. 75–85. ISBN: 978-1-4612-0685-9. DOI: 10.1007/978-1-4612-0685-9_6.

[89] WindowBuilder. *WindowBuilder - is a powerful and easy to use bi-directional Java GUI designer*. The Eclipse Foundation. URL: https://eclipse.org/windowbuilder/ (visited on Apr. 3, 2017).

[90] Claes Wohlin et al. *Experimentation in Software Engineering*. Springer-Verlag Berlin Heidelberg, 2012. ISBN: 978-3-642-43226-2. DOI: 10.1007/978-3-642-29044-2.

[91] Anthony S. Zigmond and R. Philip Snaith. "The Hospital Anxiety and Depression Scale". In: *Acta Psychiatrica Scandinavica* 67 (6), 1983, pp. 361–370. ISSN: 1600-0447. DOI: 10.1111/j.1600-0447.1983.tb09716.x.