LUND UNIVERSITY

ADAPTABLE ANDROID APPLICATIONS FOR PALCOM BASED SYSTEMS

Björn A. Johnsson

Master's Thesis at the department of Computer Science
Examiner: Boris Magnusson

Lund, 2011

Abstract

The aim of this thesis is to make it simple and efficient to produce applications for smartphones. The main area of application is within the health care sector, where alarm receiving devices are needed to inform personnel of pending situations. Most medical alarm systems are unique and hence require a unique GUI (Graphical User Interface). However, at the core, the main functionality is to communicate with the medical equipment, and therefore the needed smartphone applications would at the core be very similar.

To address this issue, this thesis establishes the similarity between the different alarm systems by assuming that they are based on a service based system called PalCom. This ensures that the communication between the smartphone application and the system, as well as other basic functionality, is done in the same way for all applications. With a common core that can be reused for all applications for PalCom based systems, the issue of creating the context unique GUIs remained. This was resolved by developing a GUI language specialized for PalCom systems. The generic nature of the language ensures simplicity and the possibility of ports for multiple platforms.

By combining the concept of service based systems with a platform generic GUI language, this thesis resulted in an Android application with a context adaptable GUI. The GUIs are defined in files using the developed language, and by using these as input, an application with a GUI customized for a specific scenario is obtained without the need to rebuild from scratch every time.

Acknowledgments

I would like to thank my examiner Boris Magnusson for the opportunity to work on this project, and for all the support and advice he provided along the way.

Contents

1 Introduction **11**

2 Problem description **13**

3 Background **15**
- 3.1 Introduction . . . 15
- 3.2 Ground work . . . 15
- 3.3 PalCom . . . 16
  - 3.3.1 Palpable computing . . . 16
  - 3.3.2 Architecture . . . 17
  - 3.3.3 Devices . . . 17
  - 3.3.4 Services . . . 17
  - 3.3.5 Assemblies . . . 18
  - 3.3.6 Communication . . . 18
  - 3.3.7 Example scenario . . . 18
- 3.4 NetBeans . . . 19
- 3.5 GUI languages . . . 20

4 Objective **23**
- 4.1 Introduction . . . 23
- 4.2 System control . . . 23
  - 4.2.1 BrowserGUI tool . . . 24
  - 4.2.2 Custom GUI . . . 25
  - 4.2.3 Description based GUI . . . 27
- 4.3 Requirements . . . 27
  - 4.3.1 Description language . . . 27
  - 4.3.2 Description interpreters . . . 28
  - 4.3.3 Example of use . . . 29

5 PalCom User Interface Markup Language **33**
- 5.1 Introduction . . . 33
- 5.2 Overview of PUIML . . . 34
  - 5.2.1 Structure of PUI Descriptions . . . 34
  - 5.2.2 The universe block . . . 36
  - 5.2.3 The discovery block . . . 38
  - 5.2.4 The structure block . . . 38
  - 5.2.5 The style block . . . 40
  - 5.2.6 The behavior block . . . 42

5.3 Summary 43

6 Interpreters for PUI Descriptions **45**
6.1 Introduction 45
6.2 PUID Interpreters 45
6.2.1 Swing/AWT PUID Interpreter 46
6.2.2 Android PUID Interpreter 47
6.3 Details 48

7 Validation **49**
7.1 Introduction 49
7.2 Example of PUI use 49
7.2.1 System configuration 49
7.2.2 Comparison 52
7.3 Validation of PUI Markup Language 53
7.4 Validation of PUID Interpreters 55

8 Future work **57**
8.1 Introduction 57
8.2 PUI Markup Language extensions 57
8.2.1 Variables in PUIML 57
8.2.2 Translatable constants in PUIML 58
8.2.3 Property templates in PUIML 58
8.3 Android PUID Interpreter 58
8.4 Graphical PUI Description editor 59
8.5 Real-life application 59
8.6 PalCom integration 59
8.7 Summary 60

9 Evaluation **61**

10 Conclusion **63**

References **65**

Appendices **65**

A PUIML specification **67**
A.1 Units 67
A.1.1 P:Device 67
A.1.2 P:Service 67
A.1.3 P:Command 69
A.1.4 P:Param 70
A.2 Parts 70
A.2.1 G:TopContainer 70
A.2.2 G:Area 71
A.2.3 G:Tabbed 72
A.2.4 G:Label 72
A.2.5 G:TextArea 74
A.2.6 G:TextField 75
A.2.7 G:Image 75

A.2.8 G:Button 77
A.3 Layout properties 79
A.4 Font properties 80
A.5 Filename values 81

B Implementation 83
B.1 Model 83
B.2 Java front-end 84
B.2.1 se.lth.cs.xbjorn.java_fe_puidi.parser.parts 84
B.2.2 se.lth.cs.xbjorn.java_fe_puidi.parser.units 85
B.2.3 se.lth.cs.xbjorn.java_fe_puidi.builder 86
B.2.4 se.lth.cs.xbjorn.java_fe_puidi.connector 86
B.2.5 se.lth.cs.xbjorn.java_fe_puidi.parser 86
B.3 Android back-end 87
B.4 Swing/AWT back-end 87

C Installation 89
C.1 Android PUID Interpreter 89
C.1.1 Application 89
C.1.2 PUI Descriptions 89
C.1.3 Tunnels 90
C.2 Swing/AWT PUID Interpreter 90

D User's manual 91
D.1 Android PUID Interpreter 91
D.2 Swing/AWT PUID Interpreter 95

E PUIML code samples 99
E.1 Android PUID Interpreter 99
E.1.1 Photo Printer 99
E.1.2 Alarm Receiver 102
E.2 Swing/AWT PUID Interpreter 106
E.2.1 Photo Printer 106
E.2.2 Photo Printer (BrowserGUI imitation) 109

Chapter 1

Introduction

The development of a Graphical User Interface (GUI) is a well established and time consuming stage in many software development projects. The purpose of the GUI is to enable the possibly inexperienced end user to use the software as effectively as possible. However, as many developers will point out, the development of a GUI is no simple task. The problems are varied, but mainly relate to the fact that describing the visual aspects of the interface using code is not very intuitive. This problem can be addressed by using a graphical editor, such as *Interface Builder* [1] or *NetBeans* [2]. With this kind of tools, GUIs are created graphically, and the need to use code is very low, thereby simplifying the development process.

When developing applications for smartphones, a good amount of special skills are required. However, when one examines the available applications, one realizes that many of these are very simple in their nature. Many applications are simply shells, used to display data provided by a server which is doing all the actual computing. The technically challenging cores of these shell application are in many cases very similar, but still have to be rebuilt for every new application. This is one of the central problems addressed in this thesis. The problem is particularly evident in service based systems, where the communication between applications and services is done in the same manner.

In a service based system, each piece of functionality is represented by a service. These services can be used directly, or combined to create new services. Working with a service based system, this thesis aims to simplify the process of creating smartphone applications that serve as shells for one or more services. By using a common service based system, a common core for multiple applications can be formed. To further simplify the development process, the GUIs will not be designed using the platform native language, but rather a language that describes GUIs in a platform generic manner. This will not only ensure that the complexity of the code, and therefore, the amount of special skills needed to create the GUIs is kept low, but also that the applications can easily be ported to other platforms in the future.

The need for a simpler method of developing smartphone applications was observed in the health care sector. In a hospital, there are many situations when there is a need to contact a specific professional, e.g. when a patient presses the call button, or goes into cardiac arrest. Today, most hospitals use primitive systems, such as beepers, to alert the personnel. Such devices provide little to no information about the patient or the situation at hand. By using smartphones instead, considerably more information could be provided and presented in a easily digestible way. However, different hospitals and even different departments within the same hospital might need alarm system for different purposes. Therefore, many different smartphone applications are needed.

By combining the concept of service based systems with a platform generic GUI language, it is the aim of this thesis to produce a system that can create smartphone applications with a context adaptable GUI. In doing so, it will be possible to create a uniform framework that can be used for both patient calls and alarms.

Chapter 2

Problem description

In health care there are many situations when there is a need to contact professionals, for example:

- when a patient consciously and explicitly wants to attract attention
- when some piece of supervising equipment detects a potential problem with a patient
- when a specialist needs to be called in during an emergency

There are various solutions to this problem in use today, such as different visual queues provided by flashing lights in hallways and other spaces, or so-called pagers that invites the wearer to dial the number provided. The power of such technologies are limited to informing the receiver that there is a problem, but cannot to any useful extent explain the nature of the problem. This issue argues for some solution based on so-called smartphones. Such a solution would provide a number of immediate benefits. The medical personnel already carry the equipment — a phone — with them. The phone can function both within the premises of the hospital (over WLAN) and outside (over the 3G-network), e.g. when on standby duty. A notification may contain essential information including text, pictures and graphs, and the same equipment can be used to react to the notification, e.g. respond to whoever raised the alarm.

Since the need to communicate alarms with smartphones has been identified in a number of medical situations, a general solution that works for different kinds of alarms is sought in order to avoid that multiple solutions are developed. Listed below are some medical situations that might have different alarm structures and would require different formatted alarm messages, but that all the same should be handled by the same solution:

Patient request A hospitalized patient needs to get in contact with the medical staff. Today, buttons are strategically placed around the hospital rooms, and when pressed a nurse is summoned. In case of an emergency, the nurse then contacts the doctor.

Staff request There may be cases when the present staff members need to inform more competent personnel about a pending situation. Today, this is done with primitive technologies that provide little to no information on the case.

Patient monitoring To monitor the status of a patient there is a lot of supervising equipment available. Whenever the equipment detects that the patient is having problems the proper personnel must be informed of the situation. Today, this is all handled manually.

Incoming patient In some cases, the staff of the ambulance transporting the patient can collect data on his/her vitals, and forward them to the future doctor in charge. This way, decisions and arrangements can be made *before* the patient even arrives at the hospital.

Standby duty When caring for patients in critical situations, around the clock monitoring of their vitals might be necessary. It would be of great value for a doctor on standby duty to be able to monitor the values provided by the supervision equipment directly in his/her smartphone.

As seen so far, there is a need to create Graphical User Interfaces (GUI) for smartphones, the main reason being the need to present medical alarm messages in a graphically rich manner. Because of the multiple medical situations where these GUIs need to be applied, creating custom smartphone applications from scratch for each one would require a vast amount of resources. Not only that, but working on the assumption that the medical environment has an established internal structure connecting the medical equipment, the applications would internally be very similar in terms of functionality, whilst the GUIs themselves might be radically different. Hence, a major part of the process of creating the GUI for a new alarm system would be wasted on replicating (with minor changes) the functionality of previously developed applications for other alarm systems. From this, the two major problems that need to be resolved in order to create a solution that can be shared between several medical alarm systems can be deduced:

1. Since the smartphone applications share many similarities internally, there is a great need for a *common base* in order to avoid unnecessary redevelopment.

2. Since the solution will be applied to multiple alarm systems, the amount of resources that goes into developing the GUIs should be kept as low as possible.

It is the aim of this master's thesis to address the problems discussed above.

Chapter 3

Background

3.1 Introduction

This chapter will provide background information and introduce concepts that are needed to understand the rest of this thesis. The work on which this thesis builds upon is discussed, followed by a rundown of the PalCom framework. Lastly, the GUI builder of NetBeans and the concept of GUI languages are introduced and considered for the purpose of this thesis.

3.2 Ground work

In order to further specify the target problem of this thesis, as well as narrow down its scope, a study of a related master's thesis is appropriate. The work that will be discussed in this chapter section was carried out by Erik Johansson and Thomas Persson at Lunds University in 2010 [3].

In the report, the authors take on the task of introducing a new device to distribute alarm messages to the staff at hospitals in Region Skåne. As in this thesis, the new device in question was a smartphone. However, unlike this thesis the authors were only interested in a single source of alarms, namely a newly developed, wireless alarm button. To connect the alarm buttons with the smartphones they had to develop an entire new patient alarm system. This system also had to be connected to several medical databases to be able to get information on patients, so that relevant information could be included in the alarm messages. The system was built upon a framework called PalCom (see chapter 3.3), and a major part of the work was to connect all the different parts of the system to this framework. Once the system was in place, a GUI was developed for the smartphone in order to display the data sent in the alarm messages. However, this GUI was tailor built for this system alone, making it useless in the context of a different, although perhaps similar, medical alarm system.

The thesis proves that a viable alarm system can be developed using PalCom to connect some alarm source (alarm button, heart rate monitor, etc.), one or more medical databases and smartphones to display the alarm messages produced by the system. The problem is that the thesis solves the problem for a specific case, and if a similar system was to be developed for another case there would be a lot to redo, from two perspectives:

1. System connectivity
2. Smartphone GUI

Perspective 1 refers to the aspect that in a different system, their is likely to be different alarm sources, databases and different alarm message formatting. While much of the work of the authors can be reused in a new system, there is bound to be key differences that require major redesign of the system.

Perspective 2 refers to the fact that in a new system, the alarm messages will certainly contain different kinds of data and different formatting, i.e. were and how information is displayed in the GUI. Since the included data and formatting of the alarm messages are tightly bound to the producing alarm system, no two GUIs are likely to be the same, and from this perspective, non of the authors work is reusable. New system, new GUI.

Above we have seen two perspective were the work of Johansson and Persson would need to be reworked to make it dynamic enough to be reused in a new system. To narrow the scope of this thesis, only the second perspective will be treated. That implies that for all intents and purposes, the actual system producing the services can be substituted by the simulated system for the duration of this thesis. It also implies that the product of this thesis will be a customizable GUI that can be used for most, if not all, future systems of the kind described in [3].

3.3 PalCom

In the work of Johansson and Persson [3], which future systems will be based on, the components of the system are connected using a framework called PalCom. Since the customizable GUI proposed in this thesis will be tightly bound with PalCom, a basic understanding of the purpose of PalCom as well as its primary components is essential for this thesis, and will be covered in this chapter section.

3.3.1 Palpable computing

Ubiquitous computing [4] refers to a new type of computing in which computers (in different forms) completely penetrates all aspects of the life of the users. Instead of just having multi-purpose personal computers in the form of desktop computers and laptops, a seemingly endless array of other computing devices are emerging. Examples of such ubiquitous devices are smartphones, digital cameras, GPS-devices, etc. What distinguishes this new type of computers from the traditional ones is that these devices are usually built to solve a limited set of problems, and therefore only offer one or a few specific services. In the case of a GPS-device, its sole purpose is to calculate the global position of the user, and will most likely offer this position as a service to the user. An issue that arises with these new devices is that to fully take advantage of their power, communication amongst one or more devices is often necessary. Since most devices offer only one or a few predefined methods of communication, connecting any number of devices with each other is guarantied to be a challenging and time consuming task.

The problem of having different devices communicate with each other was addressed by the PalCom (Palpable Computing) project, which ran from 2004 through 2007. The project aimed to solve this problem by introducing the concept of *palpability* to the emerging computing devices. The term ”palpable” denotes systems (here: devices) that can be both noticed and logically understood [5]. This means that palpable systems should support the user in understanding and controlling the services provided by their devices, by offering a coherent interface. The project extended to how services on any type of device may be logically interlinked with each other, providing not only the means to control individual devices, but actually combining several devices to provide entirely new possibilities [6].

3.3.2 Architecture

The fruit of the PalCom project was the PalCom framework — the implementation of the open PalCom architecture. The framework is implemented in Java and consists of a set of communication protocols and logical structures representing the various parts of a palpable system. The most central of these are *devices*, *services* and *assemblies*. The interaction between these three is achieved through *discovery*, *connections* and *tunnels*. These concepts are discussed in the subsequent chapter sections.

3.3.3 Devices

A PalCom device represents any kind of device. A devices can be a piece of hardware, like a digital camera, or a software simulation. Ideally, the PalCom devices should run on the physical devices' own system and communicate directly with its interface. However, this is not always possible, and therefore it is sometimes useful to run the PalCom device on a simulated device (on a computer). The simulated device then serves as a layer between the PalCom environment and the system of the psychical device.

The purpose of the device is to host services and manage their identity in the PalCom environment. The devices also run several managers to function in the environment.

3.3.4 Services

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 services can have a direct link to something in the physical world, such as displaying a text on the screen of the device, but may also be strictly confined to the software environment, such as updating an internal counter in the device. A service is defined by its *service description*, which defines the interface towards which the user will operate. The description specifies a list of *commands* that the service provides for the user. It is through sending (and receiving) commands that the user manipulates a given service.

A **command** is a message that can be passed to, or received from, a service. They are identified by a name (ID), a direction which specifies whether the service wants to receive or send the command, and an optional list of *parameters*. It is in **parameters** (params) that the data being passed to or from a service is actually stored. A param is identified by its name (ID), and may contain different kinds of data, such as plain text, images, etc.

3.3.5 Assemblies

The logic of interaction between services is handled by assemblies. Assemblies are configurations that describe how and when communication between services should happen. The configuration consists of references to devices and their services, how these should be connected, and definitions (scripts) describing when and what the services should communicate. Aside from simply passing data between services at the right time, assemblies can also form new services, so-called *synthesized services*.

3.3.6 Communication

The **discovery** protocol is what makes the PalCom devices aware of each other. This is achieved through heartbeats; periodically sent broadcast messages that urges all devices to identify themselves as ”alive”. Whenever communication between two devices is needed, a **connection** is created based on the discovery data. The connection is cleaned up by the discovery protocol when it is determined to be ”dead”. Discovery is limited to finding devices that reside within the local area network. However, a **tunnel** can be used as a simple means of extending the PalCom environment to include a remote location. These tunnels are simple connections between two networks that basically forward PalCom traffic, like heartbeats, from one network to another, making it appear as if devices on different networks are actually on the same.

3.3.7 Example scenario

Presented below is a complete example scenario to demonstrate the power of PalCom, as well as to further clarify the purpose of the different components. In the example, three services on three different physical devices cooperate to fulfill a greater goal.

Surveillance is important in today's society. In this fictive scenario, there is some predefined immobile object that is to be photographed with a constant frequency, say once every hour on the hour. The image is to be printed and handed to qualified personnel for closer inspection. The obvious way to solve this problem is for some one person to physically be in the the vicinity of the object in question, closely monitor his/her watch and every hour snap a picture of the object. The picture is then to be printed and handed to a courier who rushes it to the right people for inspection.

It should be obvious to the reader that the problem in the scenario can be solved a lot easier than described above using current technology, but for the sake of demonstrating how PalCom can be used to combine different devices to solve a task, let's assume that it can't be solved any easier.

One way to solve the problem is to mount a digital camera pointed at the object, and have some kind of timing device (e.g. a computer) tell the camera to snap a photo every hour on the hour. The photo can then be sent over some network (e.g. WIFI, 3G, . . .) to a remote site, where a printer is instructed to print the photo. The printed photo can then be inspected by the right people.

The solution is illustrated in figure 3.1. The digital camera provides services both for snapping a picture as well as fetching taken photos. The timer device provides a service that signals with a predefined frequency. The printer offers a service that takes a picture

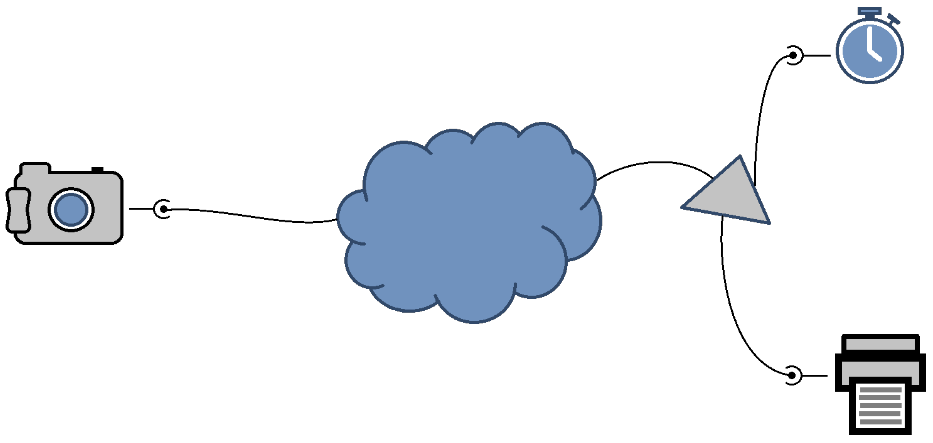


Figure 3.1: Scenario layout; devices and connections

as input, and prints it on paper. Note that in the figure, the camera is on a different physical location than the other devices. These two locations need to be connected, but that part of the solution is not relevant to this example, and is therefore illustrated with a cloud in the figure. The intelligence of the system is provided by the assembly (triangle in figure 3.1). When the timer device signals that it is time to snap a photo, the assembly contacts the camera and makes it snap a photo. The assembly then fetches the photo from the camera and forwards it to the printer service, which in turn prints the photo.

This simple example showcases how PalCom can be used to combine otherwise incompatible devices. While developing and implementing services similar to the ones described above might require a quite high level of competence, once developed they are incredibly flexible and can be reused in any number of different solutions.

3.4 NetBeans

NetBeans is an integrated development environment that, among other things, allows the developers to rapidly create Java applications. The environment offers a wide variety of features, but the most relevant for this thesis is the *Swing GUI Builder* [2]. This is a graphical editor that makes it possible to create GUIs by dragging and positioning GUI components from a palette into the main frame of the GUI. The builder automatically takes care of the spacing and alignment of the components, and the developer has constant visual confirmation of what the final GUI will look like.

The NetBeans Swing GUI Builder simplifies the process of creating a new GUI for a Java application. As mentioned above, the developer can at all time see what the final GUI will look like. This constant visual feedback is sure to decrease the amount of time that is required to get the desired look for the GUI. Furthermore, the amount of code that has to be produced by the developer is significantly reduced. This is due to the fact that whenever the developer adds a new GUI component, the code needed to create said component is automatically added to the underlying Java file. The same is true for when the developer changes the properties of existing components. The only code that the developer has to write manually is the code that links the components of the GUI to the actual application. This is sometimes referred to as *glue code*, and serves no purpose other than to link together parts of code. In this case it links the code describing the GUI with the code describing the functionality of the application.

The GUI builder described in this chapter section undoubtedly makes the process of producing a GUI simpler. However, the GUIs are not independent: for every new GUI that needs to be created, a new underlying basis, i.e. application, has to be created. Using the NetBeans IDE, this basis is created effortlessly, but for other platform the situation might be different. Another issue that stands unsolved is that when the GUIs should be used to control systems that are service based, the functionality of the applications (service communication) will be very similar. Even so, the functionality has to be rebuilt for each solution, since it has to be tailor made to suit the target service(s) of the system.

As mentioned earlier, the GUI components are linked to the functionality of the application using glue code. Furthermore, the functionality itself can be seen as glue code in that it only links the aforementioned glue code to the actual functionality that resides within the services of the target system. The use of glue code upon glue code indicates that the use of traditional graphical GUI editors like the one discussed in this chapter section is not optimal for the purpose of this thesis.

3.5 GUI languages

When creating GUIs for one specific target platform, the process is usually pretty straight forward. The developer is typically offered one or more graphical modules, or packages, that can be used to programmatically create the GUI using the programming language of the target platform. Another common method is that the programmer is presented with a graphical editor. In such editors, the developer can specify the appearance of the GUI without the need to write any code.

When creating *generic GUIs*, i.e. GUIs that should be usable on multiple platforms, the process is more complex. Different platforms differ not only in what can be described, but usually also in how the description should be structured. One way to get around these problems is to use a GUI language that was created to describe GUIs in a generic manner. There are plenty of alternatives for this purpose. Some studied languages that fit the description are *UsiXML* [7], *XUL* [8] and *UIML* [9]. All of these were developed to define GUIs that should be usable in different contexts, such as on different devices and/or platforms.

Traditionally when defining generic GUIs, or GUIs in general really, the formula is often the same: define what components should be included, where they should be positioned, what they should look like, and in this case most importantly, what they should do. Any interactive graphical component, e.g. a button, can be linked to some predefined behavioral script that describes what should happen when the user interacts with the component. In a truly generic GUI description, such scripts can and must be extremely complex to provide the means for complex functionality. Any component can be linked to an infinite number of functionality descriptions.

For the purpose of this thesis, the behavioral expressiveness of the generic GUI language will be adjusted. The GUI language will remain generic from a platform standpoint in that the same language can be used to specify GUIs for multiple target devices, such as computers, television sets, or smartphones. However, the GUI language will be strictly limited to providing the means of defining behavior for PalCom entities (devices, services, etc.) only. While this is a severe limitation on the behavioral power of the language, the gain is that the GUI definitions will be much simpler and shorter. Any interactive graphical component can only be linked to a limited number of PalCom entities, instead of an infinite number of functionality scripts. Since the link itself implicitly defines the desired behavior, no further behavioral description is needed. More on this later.

The use of GUI languages will prove to be a key aspect of producing a GUI that can be reused for more than one alarm system. Even if the languages mentioned in this chapter section mightn't fit the purpose of this thesis on the most fine grained level, they will at the very least provide a good basis for an extended language.

Chapter 4

Objective

4.1 Introduction

Building on the fact that the alarm system created by Johansson and Persson [3] is built upon the PalCom framework, this chapter will discuss the traditional methods of controlling such systems. Furthermore, a new alternative method will be introduced, along with a set of requirements that define the new method.

4.2 System control

The way to control a system that is built upon the PalCom framework is to control some specific PalCom services within the system. It is through interacting with such services that the user gets the means to influence the entire system. To control a service, or a set of services, there are today two main alternatives:

1. Use the *BrowserGUI* tool
2. Build a custom GUI from scratch

As will be shown in the next two chapter sections, both alternative 1 and alternative 2 can be considered to be extremes. For the purpose of this thesis, a new alternative method of controlling services will be introduced:

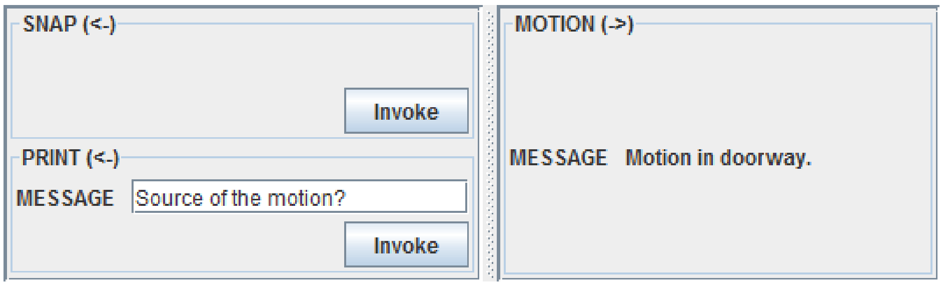
3. Build a custom PalCom GUI using descriptions

These three alternatives will be presented and discussed in the upcoming chapter sections. In order to clarify these and future concepts, the scenario from chapter 3.3.7 will be modified to serve as a general example for the rest of this thesis. In the example scenario, a photo snapped by a camera is printed on a printer every time a clock device ticks, i.e. with a certain frequency. In this modified version the clock device will be replaced by some sort of GUI, so that the user decides when the process should be triggered. To elaborate the example the camera should be able to notify the observer if it detects motion in the image. Also, when the photo is to be printed, the user should be able to provide a text that will be printed in the lower right hand of the photo. If all this functionality was to be collected into a synthesized assembly, the service description would have the following commands:

- **MOTION : output**
  The command signals that the camera has detected motion in the image.
  - `MESSAGE : text`
    A message explaining the motion.
- **SNAP : input**
  The command is used to make the camera snap and store a photo.
- **PRINT : input**
  The command is used to print the most recent photo taken by the camera.
  - `MESSAGE : text`
    A message to be printed along with the photo.

4.2.1 BrowserGUI tool

The BrowserGUI tool is a tool native to the PalCom framework. It enables the user to connect to one or more PalCom network of their choosing. Once connected, all of the available PalCom devices on the network(s) are accessible to the user. To control any given device, the user simply selects it from the list. The user is now presented with all of the PalCom services that the chosen device offers. If the user selects one of these services, s/he will be presented with a simple GUI to *control* said service. The level of control that is offered is however at its most basic. The PalCom commands of the service are found through its service description, and are displayed graphically as bordered areas with the command ID as the title. Inside these areas, the PalCom parameters are listed one by one in the order they appear in the service description. They are displayed as the parameter ID coupled with a control to edit or display the parameter value. At the time of writing, the BrowserGUI tool supports only two parameter data types: *image* and *text*. As an example, figure 4.1 illustrates what the GUI generated by the BrowserGUI tool looks like when controlling the service described above.



MESSAGE Source of the motion?

MESSAGE Motion in doorway.

Figure 4.1: Example GUI generated by the BrowserGUI tool

Acquiring control of a service using the BrowserGUI tool requires virtually no work at all from the user. The tool is started, and when the correct device/service has been located, it's immediately ready to be controlled. However, this accessibility comes at a cost: The user is offered no way to control what is displayed. All commands and parameters for any given service are shown, and there is no way to reorganize or change the look of the components. Another issue associated with the BrowserGUI tool is its indirect approach of acquiring control of services. After starting the tool, the user has to connect to the appropriate networks and then, using the browser provided by the tool, navigate to the desired service. It's only after doing all this that the user is presented with a GUI to control the service in question. For an inexperienced end user, this process might be too technically challenging.

Figure 4.2 illustrates what the example scenario system described earlier would look like if controlled by a the BrowserGUI tool. Notice that the tool connects directly to the synthesized service in order to control it. This is made possible by the *discovery mechanism* that is built into the tool, which allows the tool to find and influence the synthesized service. A benefit of controlling a service in this manner is that multiple GUIs can connect to the same service.

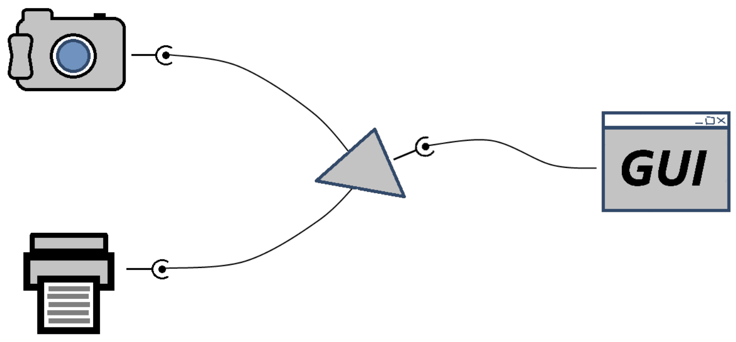


Figure 4.2: Example scenario controlled with BrowserGUI/ad hoc GUI (2)

4.2.2 Custom GUI

An alternative way of controlling a service is to build a custom GUI from scratch. In relation to alternative 1, this involves a lot of work. Aside from the task of designing and implementing the actual GUI, the developer must also link the GUI to the PalCom universe. This can be done in one of three ways.

The first possible way is illustrated in figure 4.3. Notice that it is the inverse of how the service is controlled by the BrowserGUI tool (figure 4.2). Instead of the GUI finding the assembly (service), the assembly finds the GUI. Since the assembly must recognize all GUIs involved in the system, a drawback of this approach is that in order to connect another GUI to the system, the assembly has to be updated to include the new GUI.

The second possible way is to mimic the method used by the BrowserGUI tool. Because of this, the system looks the same as for the BrowserGUI case, i.e. figure 4.2. To get this effect, the developer has to, not only develop the custom GUI itself, but also a custom discovery mechanism. The gain is, as with the BrowserGUI tool, that multiple GUIs can connect to the same service without the need to update the system. The drawback is that for every new custom GUI, a new discovery mechanism has to be implemented.

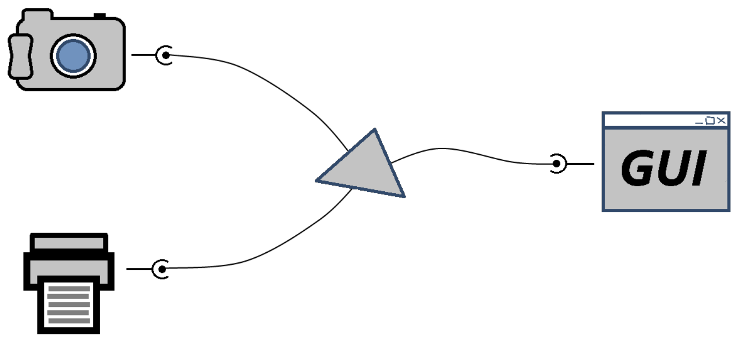


Figure 4.3: Example scenario controlled with ad hoc GUI (1)

The third possible way is illustrated in figure 4.4. The GUI is connected to the synthesized service via a 1:1 transit assembly. This assembly initiates the connections and forwards data from the service to the GUI, and vice versa. As before, this connection approach allows for several GUIs to connect to the same service, and while this method doesn't require a custom built discovery mechanism, the transit assembly has to be specially made for each GUI.

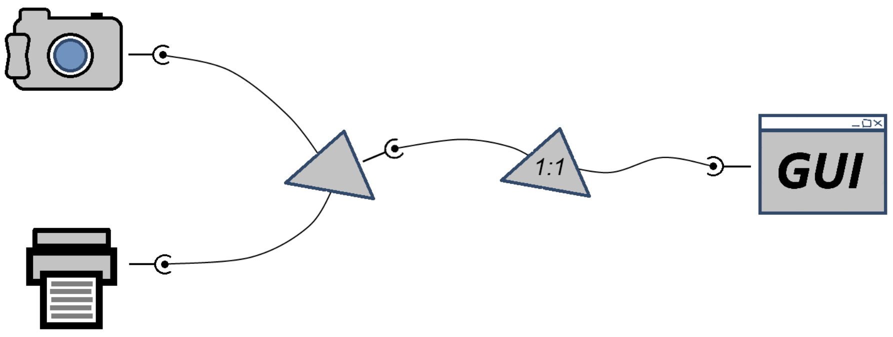


Figure 4.4: Example scenario controlled with ad hoc GUI (3)

When using this alternative method of controlling a service, every solution consists at the very least of one ad hoc GUI. Depending on which approach is selected to connect the GUI with the service, either significant system updating or ad hoc appendages are needed to get a fully functional system. While some of what needs to be done can surely be recycled from previous solutions, the vast amount of work needed to get a working system should nonetheless be considered a major drawback of this method. However, the gain of using this method is not insignificant. It effectively solves the issues associated with the BrowserGUI tool. Since the GUIs are custom built for each different scenario, their content, looks and behavior is extremely versatile. A direct point of entry will most likely be provided for the user by the developer.

4.2.3 Description based GUI

Both alternative 1 and alternative 2 can be considered to be extremes: alternative 1 requires virtually no effort to get started, and offers no way to customize the GUI. Alternative 2 on the other hand is very customizable, at the expense of the amount of effort to get it working properly. For the purpose of this thesis, a solution that combines the low development efforts of alternative 1 with the high customization power of alternative 2 is desirable. In view of this, a new alternative for controlling services is introduced as an objective for this thesis.

This new alternative should be more closely related to alternative 2 than to alternative 1 in that the user gets a custom GUI for each scenario. To facilitate the introduction of multiple GUIs for the same service, the GUIs should connect directly to the target service using a discovery mechanism. However, instead of having to rebuild the discovery mechanism and the GUI for every new scenario, they will instead be reused, and the only thing that needs to be rebuilt is a *description* file. The description file describes what the GUI should look like, and what its behavior should be. Figure 4.5 illustrates what the example scenario system would look like if controlled using this approach.

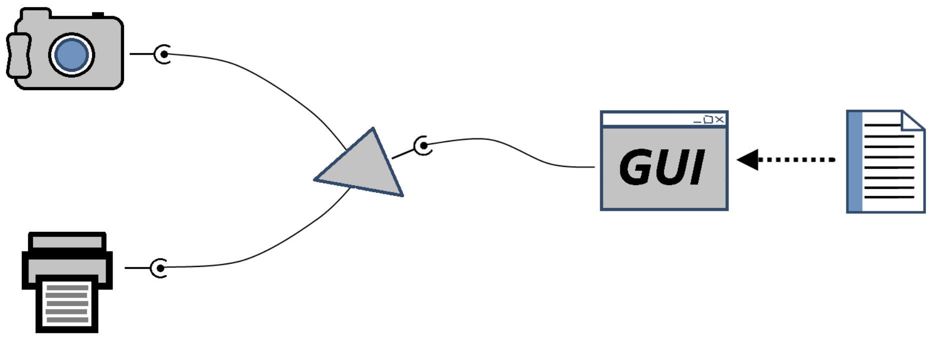


Figure 4.5: Example scenario controlled with description based GUI

The gain of this approach, in relation to alternative 2, is that instead of have to rebuild from scratch for each new scenario, only a single file has to be rebuilt. The compromise is that since the GUI language of the description file should be platform generic, the customization power can not be as high as when building natively for the platform, as with alternative 2. It is however much greater than the customization power of alternative 1 (none), and considering how much less effort has to be put into the development of the GUI, it should be considered as a fair trade-off.

4.3 Requirements

4.3.1 Description language

As mentioned above, the proposed method to reuse both discovery mechanism and GUI is controlled by a description file. The language of this file has to meet some minimum credentials. It is stated above that the new method should provide more customization power than the BrowserGUI tool. One implicit requirement for the description language would thus be:

- The language should provide the means to express a GUI that is comparable in terms of looks and functionality to that of a GUI generated by the BrowserGUI tool.

This is a minimum requirement, but simply by fulfilling this requirement the language should already be able to express more complex GUIs than the BrowserGUI tool can. More formally, the language should in terms of graphical expressiveness provide the following:

- The means to create the following Generic Graphical Components (GGCs):
  - Multi-lined text label.
  - Single-lined text input box.
  - Clickable button.
  - Scalable image viewer.
  - Area that can hold other components.
  - Tab control.
- The means to control how the GGCs are laid out in the GUI.

For each GGC the language should provide some graphical adjustment properties. These properties will be worked into the language as the need arises, and will therefore not be expressed as formal requirements.

In terms of behavioral expressiveness, the language should provide the following:

- The means to represent and reference PalCom entities; devices, services, commands and parameters.
- The means to use data provided by selected GGCs in the form of text and images as the value for parameters.
- The means to in selected GGCs display data, in the form of text and images, provided by parameter values.
- The means to invoke a command (i.e. sending the command along with the associated parameter values to the respective input command ”port” of the target service) using selected GGCs.

As already mentioned in chapter 3.5, the language will be strictly limited to providing the means of defining behavior for PalCom entities only. Due to this, the behavioral requirements listed above provide sufficient coverage.

4.3.2 Description interpreters

As discussed in chapter 4.2, the proposed way of lowering the amount of work that has to be done to get a working GUI is to introduce the concept of description files. With the description file describing the GUI, this file is the only thing that has to be replaced for different scenarios. This translates into the following requirements that must be fulfilled by description interpreter applications:

- The application must allow the user to load one (1) out of the available description files.
- The application must be able to create and display a GUI as defined by the description file.
- The application must be able to connect to PalCom entities and communicate with these as defined by the description file.

These simple yet comprehensive requirements will ensure that everything but the description files is reused for all scenarios.

The BrowserGUI tool requires at least five *selective* user operations (clicks) to get control of some service. A custom GUI would at best require only one (starting the application). Here, the term selective user operation refers to an operation where the user has to identify some target before acting, e.g. find the correct PalCom device and then clicking it. Since the proposed description based solution has to handle multiple descriptions, the following requirement is introduced:

- The application must allow the user to gain control over a predefined service in two (or less) selective user operations.

4.3.3 Example of use

In order to reconnect to the larger problem at hand, i.e. providing alarm devices for the medical staff at hospitals, the scenario from the work of Johansson and Persson [3] will be discussed. Details about the scenario might be added or removed in order to fit the purpose of this thesis.

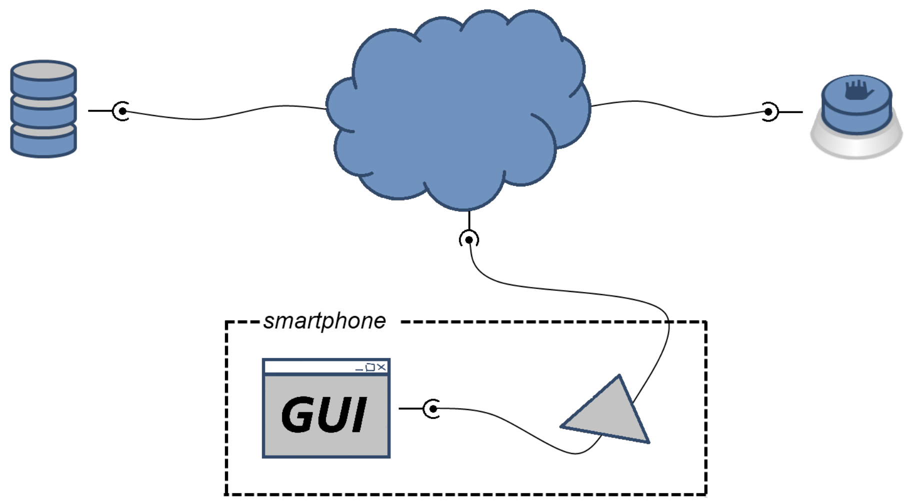
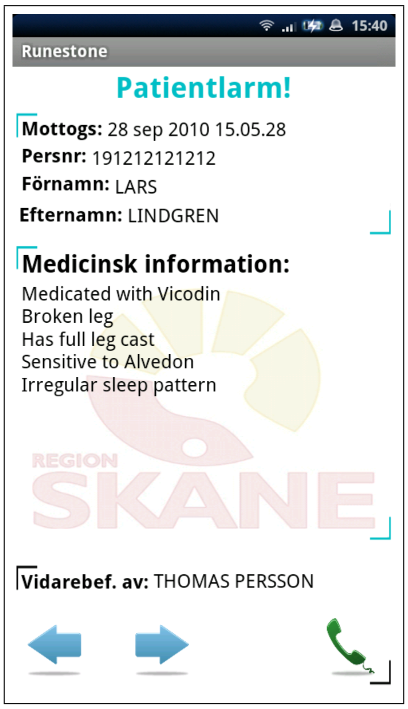


Figure 4.6: System setup of the patient alarm scenario

In the scenario, the authors have designed a complete alarm system using PalCom. The system involves, among other things, alarm buttons which are used to trigger an alarm and several medical databases to fetch information about the patient in distress. Whenever a patient presses his/her alarm button, a message with information about the patient is sent to a predefined physician. The alarm system converges into a single PalCom service. This service has an output command with one parameter for each piece of information about the patient: first name, last name, medical information, etc. It is through this command that the alarm system communicates with the receivers. The system setup is illustrated in figure 4.6.

The doctors use Android powered smartphones to receive alarm messages. The authors made this possible by developing a custom Android application specifically for this alarm system. When an alarm is received, even if the application is running in the background, a *notification* accompanied by sound and vibration is displayed to catch the attention of the user. When the user clicks it, the GUI appears and displays the alarm message as seen in figure 4.7. At the bottom of the GUI there are three buttons: a left arrow, a right arrow and a phone receiver. When the user presses the left arrow button, the previous (in chronological order) patient alarm message (if any) is displayed. Similarly, when the user presses the right arrow button, the next patient alarm message is displayed. Finally, when the user presses the phone receiver button, a phone number provided by the alarm message is dialed.



Patientlarm!

Mottogs: 28 sep 2010 15.05.28
Persnr: 191212121212
Förnamn: LARS
Efternamn: LINDGREN

Medicinsk information:
Medicated with Vicodin
Broken leg
Has full leg cast
Sensitive to Alvedon
Irregular sleep pattern

Vidarebef. av: THOMAS PERSSON

Figure 4.7: Alarm message GUI for button triggered patient alarms

The purpose of this example is to introduce a somewhat abstract requirement for the description interpreter application running on the Android platform:

- The language and the application running on the Android platform must be powerful enough to specify and run a GUI that is comparable in terms of looks and functionality to that of the example above.

To reiterate; this requirement is by choice abstract in its nature and is to be consider more as a general guide line than an official requirement. During the evaluation phase of the project, a replica of this scenario will be constructed to showcase the power and possibilities of the new application.

Chapter 5

PalCom User Interface Markup Language

5.1 Introduction

During the analysis of the languages used to specify GUIs in a generic fashion that were briefly introduced in chapter 3.5, it became apparent that any of the alternatives would provide more than enough expressive power for the purpose of this thesis. In fact, these languages provided, in more than one sense, too much power. To ensure that the GUI definitions are short and simple to understand, the proposed language has to be simple and should not provide redundant features. The researched languages provide too much expressive power in terms of graphics. The graphical requirements set up in chapter 4.3.1 can be met with a fraction of what the studied languages provide. Furthermore, the vast power to define behavior of these languages would only make the proposed language needlessly complicated, due to the fact that said language only needs to have a limited set of actions that all involve PalCom entities.

In view of the observations above, it was determined that the required language would have to be tailor suited for the purpose of this thesis. However, most of what is needed for this proposed new language is already present in the researched languages. By using one of these languages as a starting point, the proposed language would then be created by removing redundant features, modifying others, and adding completely new features that are needed for the new language. Since all of the studied languages could be used as the base for the new languages, it came down to personal preference when choosing the base. The relative simplicity of the GUI definitions provided by the language UIML [9] made it the most preferable base for the new language.

The language custom fitted from UIML for the purpose of this thesis is presented in this chapter. It is assigned the name *PalCom User Interface Markup Language* (PUIML).

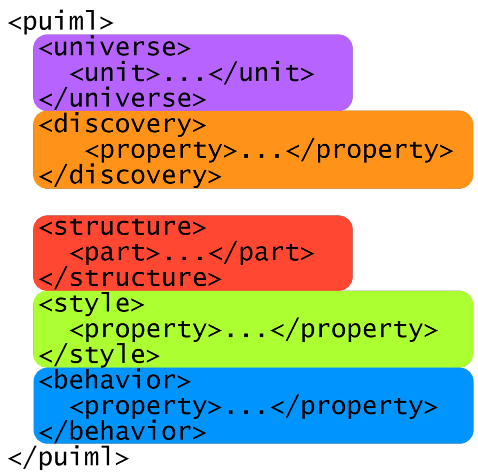
5.2 Overview of PUIML

5.2.1 Structure of PUI Descriptions

The PUI Markup Language is used to define PUIDs, or *PalCom User Interface Descriptions*. PUI Descriptions, in turn, are used to describe PUIs, or *PalCom User Interfaces*, which are GUIs created from a PUID, for the purpose of controlling PalCom services. PUIDs are structured using the eXtensible Markup Language (XML), and always contain two primary descriptive blocks: **universe**, which defines the *units* of the description, and **structure**, which defines the *parts* of the description. Units represent the PalCom entities that should be included in the system, while parts represent the graphical components that make up the PUI presented to the user.

The universe block is supplemented with the **discovery** block. This contains various properties for the units defined in the universe block. The main purpose of the discovery properties is to identify the PalCom entities.

The structure block is supplemented with the **style** and **behavior** blocks. Both of these contain various properties that affect different aspects of the parts defined in the structure block. The style properties specify the visual characteristics of the graphical components, while the behavior properties specify how they should behave. The described overview of a PUID is illustrated in figure 5.1.



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

  <structure>
    <part>...</part>
  </structure>
  <style>
    <property>...</property>
  </style>
  <behavior>
    <property>...</property>
  </behavior>
</puiml>
```

Figure 5.1: Structural overview of a PUI Description

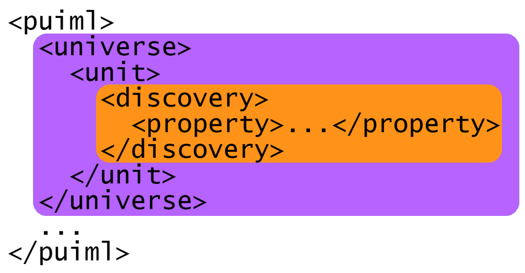
In PUIDs, units are represented by XML *elements* with the name ”unit”. These elements have two XML *attributes*: **id**, a unique ID string that identifies the unit within the description, and **class**, which describes what type (class) of unit is being represented (more on this later). Depending on the class of the unit, the unit element may contain other unit elements. Parts are represented in an analogous manner, using XML elements with the name ”part”.

Properties are represented by XML elements with the name ”property”, and these elements always have the **name** attribute, which specifies which property is being set (all available properties are listed in appendix A). For all properties, the property value is set by editing the *inner text* of the property element, i.e. the text between the properties' start tag, `<property>`, and its end tag, `</property>`. To link the property to an existing unit/part, one more attribute is required. If the property is meant for a unit (i.e. a discovery property), the **unit-name** attribute is used. If the property is meant for a part (i.e. a style or behavior property), the **part-name** attribute is used instead. In both cases, the attribute value must be the same as the value of the id attribute of the target unit/part.

```
1  <puiml>
2    <universe>
3      <unit id="device1" class="P:Device" />
4    </universe>
5    <discovery>
6      <property unit-name="device1" name="p:id">X:1234</property>
7    </discovery>
8    ...
9  </puiml>
```

Listing 5.1: Example of PUI Markup Language syntax

The PUIML syntax introduced in this chapter section is exemplified in listing 5.1. Please note that the only purpose of the example is to demonstrate the syntax of PUIML, and therefore the universe, style and behavior blocks have been omitted. This is represented by "..." on line 8, which in a real PUID would not be proper PUIML code. In the example, a unit of class "P:Device" with the ID "device1" is created on line 3. For this unit, the discovery property called "p:id" is set to "X:1234" on line 6. Notice that to link the property to the unit created on line 3, the unit-name attribute has the value "device1".



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

Figure 5.2: Structural overview of a PUI Description with local properties

PUIML provides an alternative way of structuring the descriptions. Instead of having one block containing all discovery properties of *all* units, as seen in figure 5.1, one can choose to specify a unit's properties directly in the definition. This is depicted in figure 5.2, and works analogously for the structure and the style/behavior blocks. As seen earlier, when properties that are created *globally* (i.e. figure 5.1, as opposed to figure 5.2), either the unit-name attribute or the part-name attribute has to be specified in order to link the property to the intended unit/part. However, when properties are created *locally* (i.e. figure 5.2, as opposed to figure 5.1), the property element is nested inside the unit/part element. The link is therefore implicit, and neither the unit-name nor the part-name attribute must be specified.

Both ways of structuring descriptions yield the same end result. It's a matter personal preference which should be used. Defining properties locally is useful for smaller descriptions in order to have all information regarding the same unit/part in one place. For larger descriptions, defining properties globally will allow for a quicker overview of the structure of the description, while keeping the details provided by the properties in one collective place.

5.2.2 The universe block

In the universe block of the PUI Description, the units of the description are declared. Units are used to represent PalCom entities in the PUID. Hence, units can belong to one of the following four classes:

- P:Device
- P:Service
- P:Command
- P:Param

The classes corresponds to the PalCom entities *device*, *service*, *command* and *parameter* (see chapter 3.3) respectively. As mentioned in the previous chapter section, unit elements can contain other unit elements — unit elements can be *nested*. How the elements can be nested depends on the class of the units. The universe element can contain $0 - \infty$ unit elements, but only unit elements of class ”P:Device”. This is logically sound since there is no entity that is higher in the PalCom hierarchy than the device. Units of class ”P:Device” can in turn contain a number of units of class ”P:Service”, since the device can be seen as being one level higher in the PalCom hierarchy than the service. For the same reason, units of class ”P:Service” can only contain a number of units of class ”P:Command”, and units of class ”P:Command” can only contain a number of units of class ”P:Param”. Since there is no level below the parameter in the PalCom hierarchy, units of class ”P:Param” can contain no nested units of any kind. Figure 5.3 presents a UML type diagram that illustrates the nesting possibilities for units as discussed above.

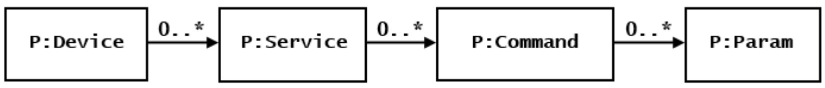
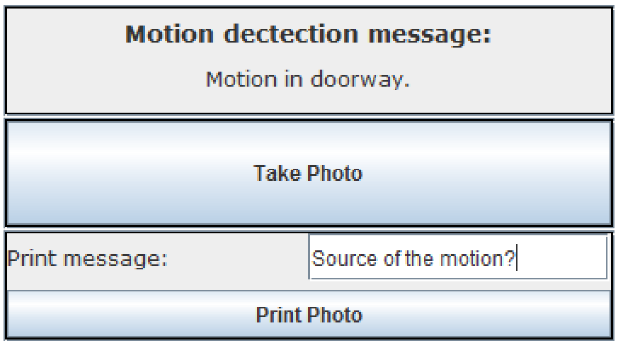


Figure 5.3: UML diagram describing the nesting of units

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 unit describing said service will be used. However, these alone are not enough to identify the service. The discovery properties of the parent unit describing a PalCom device must be used as well. This means that the same service (unit) element could be used to describe different PalCom services just by being nested under different device (unit) elements.

The PUI shown in figure 5.4 will be used as a running example for this and the upcoming chapter sections: 5.2.3, 5.2.4, 5.2.5, and 5.2.6. The PUI is based on the example scenario introduced in chapter 4.2, and controls the service specified in the same chapter. To demonstrate what each PUIML block might look like, the block of code that makes up the PUI in figure 5.4 will be presented in its respective chapter section. For example, this chapter section discusses the universe block. Hence, the universe block of the PUID that describes the PUI in figure 5.4 is presented in this chapter section (listing 5.2). If the code from all of the chapter sections are combined, they make up the PUI presented in figure 5.4.


Motion dectection message:

Motion in doorway.

Print message: Source of the motion?

Figure 5.4: PUI for chapter 4.2 example scenario

As mentioned above, listing 5.2 shows the universe block of the PUI Description that makes up the PUI in figure 5.4. Notice how all three of the commands of the target service, i.e. MOTION, SNAP and PRINT, are declared on lines 5, 8 and 9 respectively. Also notice that the IDs specified in this code sample will **not** be used to locate the PalCom entities. They are used to identify the units within the PUID, and the similarities to the actual PalCom IDs are to facilitate the understanding of the description.

```
1  <puiml>
2    <universe>
3      <unit id="device" class="P:Device">
4        <unit id="synth-assembly" class="P:Service">
5          <unit id="motion" class="P:Command">
6            <unit id="motion-message" class="P:Param" />
7          </unit>
8          <unit id="snap" class="P:Command" />
9          <unit id="print" class="P:Command">
10           <unit id="print-message" class="P:Param" />
11         </unit>
12       </unit>
13     </unit>
14   </universe>
15   ...
16 </puiml>
```

Listing 5.2: Universe block of example PUI (figure 5.4)

5.2.3 The discovery block

In the discovery block of the PUI Description, the discovery properties of the description are declared. Discovery properties apply to the units declared in the universe block, and depending on the class of the unit there is a number of discovery properties that defines it. Discovery properties are mainly used to define the identity of PalCom entities, i.e. how they can be located in the PalCom universe. Some of these properties are required and must be specified in the PUI Description, whilst others are optional and may be left out for an implicit default value. The available discovery properties are presented in appendix A.1.

The discovery block of the PUID that describes the PUI in figure 5.4 is presented in listing 5.3. Lines 4–10 list discovery properties to identify the device and service of the example scenario. The values of these properties are circumstantial, and vary depending on the system. The values of the ID properties for the commands and parameters are however correctly set according the service specification in chapter 4.2.

```
1  <puiml>
2    ...
3    <discovery>
4      <property unit-name="device" name="p:id">X:PP1001</property>
5
6      <property unit-name="synth-assembly" name="p:instance">1</property>
7      <property unit-name="synth-assembly" name="p:cdid">X:1scenarioBJ</
           property>
8      <property unit-name="synth-assembly" name="p:cn">BJscenario1</property>
9      <property unit-name="synth-assembly" name="p:udid">X:1scenarioBJ</
           property>
10     <property unit-name="synth-assembly" name="p:un">BJscenario1</property>
11
12     <property unit-name="motion" name="p:id">MOTION</property>
13     <property unit-name="motion" name="p:direction">out</property>
14     <property unit-name="motion" name="p:notifications">normal</property>
15
16     <property unit-name="motion-message" name="p:id">MESSAGE</property>
17
18     <property unit-name="snap" name="p:id">SNAP</property>
19     <property unit-name="snap" name="p:direction">in</property>
20
21     <property unit-name="print" name="p:id">PRINT</property>
22     <property unit-name="print" name="p:direction">in</property>
23
24     <property unit-name="print-message" name="p:id">MESSAGE</property>
25   </discovery>
26   ...
27 </puiml>
```

Listing 5.3: Discovery block of example PUI (figure 5.4)

5.2.4 The structure block

In the structure block of the PUI Description, the parts of the description are declared. Parts represent graphical components, such as buttons and text labels, in the PUID. They can belong to one of the following classes:

- G:TopContainer
- G:Area
- G:Tabbed
- G:Label
- G:TextArea
- G:TextField
- G:Image
- G:Button

What component each class corresponds to should be quite clear from the names, and is covered in detail in appendix A.2. It was mentioned in chapter 5.2.1 that part elements can be nested. The way in which they may be nested depends on the class of the part. The structure element can and must hold one, and only one, part element, and this part must be of the class ”G:TopContainer”. Parts of this class represent the topmost level of any PUI, e.g. a window frame, and must therefore be the first part defined in any PUI Description. For the same reason, this first part is also the only part that can be of the class ”G:TopContainer”. The nesting of part elements is used to logically structure the PUI. There are only three classes that allow nesting, i.e. can contain other parts: ”G:TopContainer”, ”G:Area” and ”G:Tabbed”. Within parts of these three classes, other parts of any class (except for ”G:TopContainer”) can be contained. For parts of all other classes, nesting is not possible. This is due to the fact that these three classes of parts are the only parts that can logically contain other parts. For example, a button (part of class ”G:Button”) can logically reside within a window frame (part of class ”G:TopContainer”), but a button can not logically reside within another button. A UML type diagram that illustrates the nesting possibilities discussed above is presented in figure 5.5.

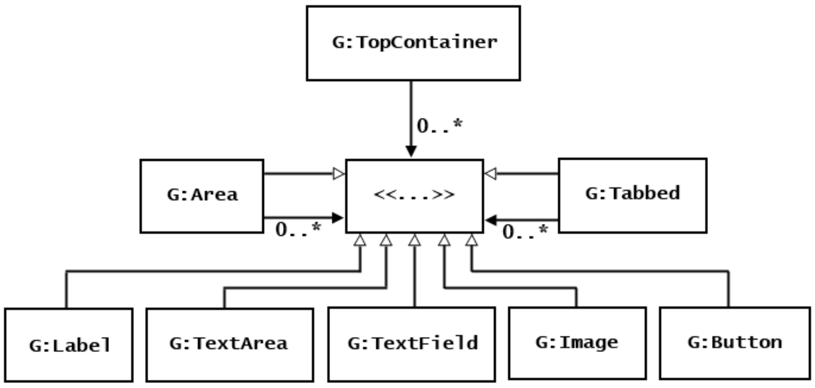


Figure 5.5: UML diagram describing the nesting of parts

It is through the logical nesting of parts, in conjunction with the concept of *layouts*, that the PUI gets its visual appearance. Layouts define how graphical components should be laid out in the PUI, e.g. from left to right, or in the form of a rectangular grid. In PUIML, layouts can be used for parts that can contain other parts by setting a number of style properties (appendix A.3). To properly structure the graphical components of a PUI using PUIML, there are three things to consider:

1. The nesting of parts decides in which parent component the resulting component should be placed.
2. The order in which the parts are declared decides the *order* in which the resulting graphical component will be laid out in the PUI.
3. The specified layout of a container part decides *how* the resulting components should be laid out.

In listing 5.3 the discovery block of the PUID that describes the PUI in figure 5.4 is presented. Notice how the parts of class ”G:TopContainer” and ”G:Area” contain other parts, whereas the parts of class ”G:Label”, ”G:Button” and ”G:TextField” don't.

```
<puiml>
  ...
  <structure>
    <part id="window" class="G:TopContainer">
      <part id="motion-area" class="G:Area">
        <part id="message1-label" class="G:Label" />
        <part id="message1-output" class="G:Label" />
      </part>
      <part id="snap-area" class="G:Area">
        <part id="snap-button" class="G:Button" />
      </part>
      <part id="print-area" class="G:Area">
        <part id="sub-area" class="G:Area">
          <part id="message2-label" class="G:Label" />
          <part id="message2-input" class="G:TextField" />
        </part>
        <part id="print-button" class="G:Button" />
      </part>
    </part>
  </structure>
  ...
</puiml>
```

Listing 5.4: Structure block of example PUI (figure 5.4)

5.2.5 The style block

In the style block, the style properties of the PUI Description are declared. These properties apply to the parts declared in the structure block, and depending on the class of the part there is a number of style properties that defines it. Style properties are used to define what a component should look like in the final PUI. Some of these properties are required, and therefore must be specified in the PUID, whilst others are optional and may be omitted. To get a sense for how the graphical components represented by the various part classes can be configured using style properties, table 5.1 shows a simplified list of what aspects of the components can be manipulated using style properties. For a complete list of the available style properties, please see appendix A.2.

```
<puiml>
  ...
  <style>
    <property part-name="window" name="g:title">Photo Print</property>
    <property part-name="window" name="g:resizable">true</property>
    <property part-name="window" name="g:layout">grid</property>
    <property part-name="window" name="g:layout-columns">1</property>
    <property part-name="window" name="g:size">440,310</property>

    <property part-name="motion-area" name="g:layout">grid</property>
    <property part-name="motion-area" name="g:layout-gap">5,5</property>
    <property part-name="motion-area" name="g:layout-columns">1</property>
    <property part-name="motion-area" name="g:scrollable">true</property>
    <property part-name="motion-area" name="g:border">line</property>

    <property part-name="message1-label" name="g:text">Motion dectection
        message:</property>
    <property part-name="message1-label" name="g:align-h">center</property>
    <property part-name="message1-label" name="g:font">Verdana</property>
    <property part-name="message1-label" name="g:font-size">14</property>
    <property part-name="message1-label" name="g:font-bold">true</property>

    <property part-name="message1-output" name="g:align-h">center</property>
    <property part-name="message1-output" name="g:align-v">top</property>
    <property part-name="message1-output" name="g:font">Verdana</property>

    <property part-name="snap-area" name="g:layout">grid</property>
    <property part-name="snap-area" name="g:layout-gap">5,5</property>
    <property part-name="snap-area" name="g:layout-columns">1</property>
    <property part-name="snap-area" name="g:scrollable">true</property>
    <property part-name="snap-area" name="g:border">line</property>

    <property part-name="snap-button" name="g:text">Take Photo</property>

    <property part-name="print-area" name="g:layout">grid</property>
    <property part-name="print-area" name="g:layout-gap">5,5</property>
    <property part-name="print-area" name="g:layout-columns">1</property>
    <property part-name="print-area" name="g:scrollable">true</property>
    <property part-name="print-area" name="g:border">line</property>

    <property part-name="sub-area" name="g:layout">grid</property>
    <property part-name="sub-area" name="g:layout-gap">0,5</property>
    <property part-name="sub-area" name="g:layout-columns">2</property>

    <property part-name="message2-label" name="g:text">Print message:</
        property>
    <property part-name="message2-label" name="g:font">Verdana</property>

    <property part-name="print-button" name="g:text">Print Photo</property>
  </style>
  ...
</puiml>
```

Listing 5.5: Style block of example PUI (figure 5.4)

Part class	**Settable properties**
G:TopContainer	Title, resizable?, layout, size
G:Area	Title, scrollable?, border, layout, size
G:Tabbed	Size
G:Label	Font properties, text, text alignment, size
G:TextArea	Font properties, text, size
G:TextField	Text, tool tip text, size
G:Image	Image source path, size
G:Button	Text, tool tip text, image source path, size

Table 5.1: Simplified list of settable style properties

The style block of the PUID that describes the PUI in figure 5.4 is presented in listing 5.5. Notice how the part-name attribute of each property element references one of the parts declared in the structure block (listing 5.4).

5.2.6 The behavior block

Behavior properties are declared in the behavior block of the PUI Description. These properties apply to the parts declared in the structure block, and are used to define how a graphical component should act in the PUI. As with all other properties, some of these are required, whilst others are optional. Since the behavioral scope of PUIML has been reduce to simply having to be able to specify PalCom specific functionality, all behavior is expressed using links. These links stretch from a single part to a single unit, and are specified as behavior properties for parts. The details of which parts can be linked to which units and how, i.e. which behavior properties do the different classes of parts have, is specified in detail in appendix A.2.

From a behavioral perspective, selected parts can take on one out of three roles (links): *viewer*, *provider* or *invoker*. A part can act as a viewer in respect to a PalCom parameter. What this means is that whenever the value of the parameter changes, the graphical component represented by the part in question will display the data to the user. In a similar manner, a part can also act as a provider in respect to a PalCom parameter. This means that whenever new data is entered by the user into the graphical component represented by the part in question, the value of the parameter will be updated with this data. Lastly, a part can act as an invoker in respect to a PalCom command. This entails that when the graphical component represented by the part in question is activated (e.g. clicked in the case of a button), the command will be invoked, i.e. sent along with its associated parameters' values.

These three simple roles enable the specification of quite powerful behavior. For example, to control some given service s, one would simply assign invoker parts for all input commands, and provider parts for all parameters of the input commands. Additionally, one could assign viewer parts for all parameters of the output commands, so as to monitor the output of s.

The behavior properties that define the behavior of the PUI in figure 5.4 are presented in listing 5.6. The behavior that is being described is that whenever the output command MOTION is invoked, the value of its text parameter MESSAGE will be displayed in the PUI where the text ”Motion in doorway.” is in figure 5.4. This is specified on line 4. Line 6 states that whenever the button labeled ”Take Photo” in the PUI is pressed, the SNAP command should be invoked. This triggers the camera of the system to take a photo. Line 9 states that the text entered in the text field of the PUI should be used as the value for the parameter MESSAGE of the input command PRINT. Lastly, line 8 states that when the button labeled ”Print Photo” in the PUI is pressed, the PRINT command should be invoked. This results in that the latest photo taken by the camera is printed along with the text in the text field of the PUI. In figure 5.4, this text would be ”Source of the motion?”.

```
1   <puiml>
2     ...
3     <behavior>
4       <property part-name="message1-output" name="p:viewer">motion-message</
            property>
5
6       <property part-name="snap-button" name="p:invoker">snap</property>
7
8       <property part-name="print-button" name="p:invoker">print</property>
9       <property part-name="message2-input" name="p:provider">print-message</
            property>
10    </behavior>
11  </puiml>
```

Listing 5.6: Behavior block of example PUI (figure 5.4)

5.3 Summary

As seen throughout this chapter, the produced language, PUIML, is by all means a complete language. The features of the language are plentiful and powerful enough to define a customized PUI, both in terms of visual and behavioral characteristics. The features that have been added to the language are in many cases related to what is needed for the purpose of this thesis. Therefore, some features that are important in other contexts might be missing. For this reason, the definition of PUIML presented in this chapter should not be viewed as final, but rather as starting point from where features can be added as the need arises.

Chapter 6

Interpreters for PUI Descriptions

6.1 Introduction

As mentioned earlier, one of the requirements for the PUI Markup Language is to be generic in respect to platform and/or device. With this requirement in mind, it goes without saying that the language can and should have PUI generators, or *interpreters*, for different programming languages, graphical libraries and devices. Such interpreters will as of now be referred to as PUIDI, or *PalCom User Interface Description Interpreter*. Because the PalCom framework is written in Java, and also to help limit the scope of this thesis, the only programming language that will be considered for these PUID Interpreters is Java. With regard to different graphical libraries and devices, this thesis will focus on two PUIDIs: The first is a simple Java application that uses the Swing/AWT libraries for graphics. The application is runnable on most computers. The second interpreter is an application for the Android OS, and is meant to be run on Android powered smartphones. In this chapter, both of these interpreters will be introduced and discussed.

6.2 PUID Interpreters

The language described in chapter 5 is a big part of the result of this thesis. The second big part of the result are the interpreters produced. The language is naturally of interest in itself, but it is with the aid of the interpreters that the actual power and the possibilities of the language are demonstrated. To properly connect the dots, and truly understand how the language and the interpreters are logically interlocked, the following terms are of the utmost importance:

PUIML *PalCom User Interface Markup Language*
The XML based language used to define PUIDs.

PUID *PalCom User Interface Description*
A document that describes the graphically generic look and PalCom based behavior of a PUI. Created using PUIML, and used as input for a PUIDI.

PUIDI *PalCom User Interface Description Interpreter*
An application that bridges the cap between PUID and PUI. Takes a PUID as input, parses it, and lastly interprets it in the form of a PUI.

PUI *PalCom User Interface*
A GUI that allow users to control PalCom entities in predefined ways. Generated by a PUIDI.

To reiterate: PUIML is used to define PUIDs. A PUID is to be used as input for a PUIDI, which will interpret it in the form of a PUI the user can interact with. The concept for a general example is illustrated in figure 6.1.

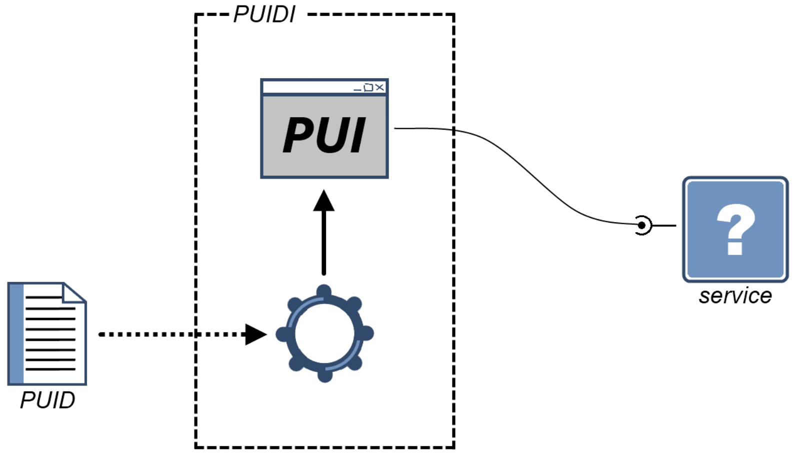


Figure 6.1: PUID Interpreter concept for the general case

What all PUIDIs have in common is that they use a PUID to create a PUI. The fact that PUIML is graphically generic means that PUIML can be used to describe PUIs for different target platforms. This is however not to say that all PUIDs generate into the same PUI, or even work on two different PUIDIs. If the same PUID is used as input for two different PUIDIs, the resulting PUI should be similar. However, due to the difference in how components are laid out and spaced, the PUIs will rarely be identical. For the same reason, it can not be guarantied that a PUID will generate into a working PUI, even if it can be parsed properly. One example of this is that the nesting of parts, while legal according to PUIML, might not work practically on the target platform. For these reasons, if the same PUID is to be used for different PUIDIs, minor tweaking might be needed to get the desired effect.

6.2.1 Swing/AWT PUID Interpreter

The first of the two developed PUID Interpreters is the Swing/AWT PUIDI. It is a Java application that can be run on computers where the Java Runtime Environment is present. There are several imaginable PUIDIs that could be built for this target platform. This one makes use of the graphical libraries *Swing* and *AWT*.

The PUIDI can be seen as a byproduct of this thesis, in that it wasn't a planned or even necessary result. It was developed mainly as a test platform during the development of the parser for the PUI Descriptions. This was much due to the fact the all-purpose computer is more flexible when debugging and testing, as compared to the alternative — a smartphone. The PUIDI was also to be used as a proof-of-concept for future interpreters. If a viable PUIDI could not be built under the optimal circumstances, i.e. the open and flexible environment of the all-purpose computer, then there would be no need to move on to more challenging platforms such as the Android smartphone. The proof-of-concept didn't just apply to the application itself, it also applied to the language. If any weak or missing points in the language were encountered, they could be addressed before moving on to the Android smartphone.

The Swing/AWT PUIDI is by all means a fully functioning interpreter, even though it is somewhat unfinished. PUIs are interpreted correctly according to PUIML, but there are some features of the language that are not supported. One such feature is *command notifications* (see appendix A.1.3, table A.4), i.e. how the user is notified when a PalCom command is received. There are more than one of these flaws that would need to be addressed before this application should be considered finished, but since this was meant as a proof-of-concept PUIDI, that is to be considered acceptable.

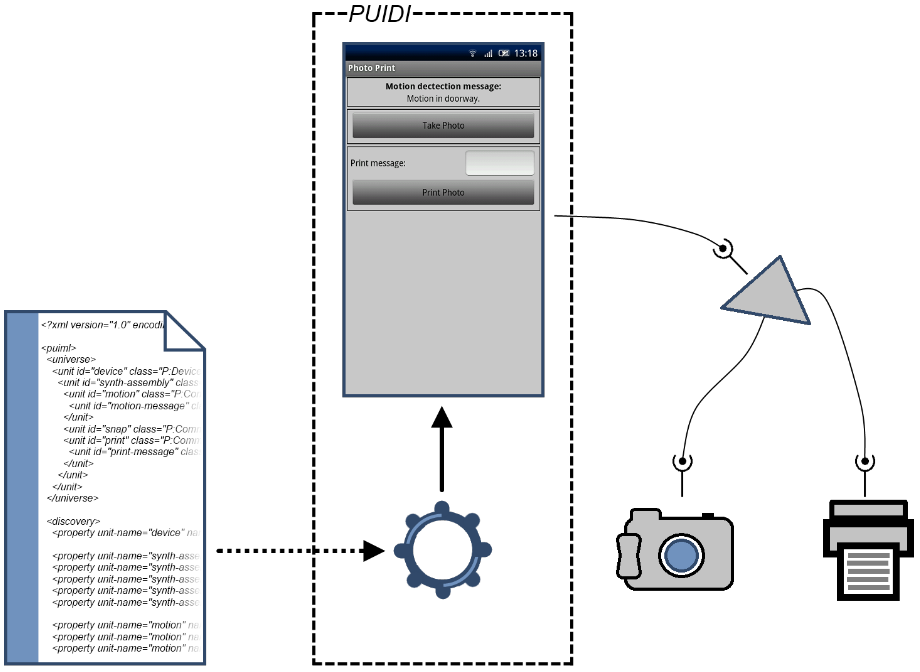
6.2.2 Android PUID Interpreter

The second of the two developed PUID Interpreters is the Android PUIDI. It is an Android app (application) that was developed to run on devices powered by the Android OS. It has however only been tested for a smartphone of the model and make *Sony Ericsson, Xperia X10*, and therefore there is no guarantee that it will run properly on other devices.

The Android PUIDI is the main result of this thesis, and even though it should still be considered to be in a prototype state it is quite complete as compared to the Swing/AWT PUIDI. The app allows the user to select one out of the installed PUI Descriptions on the device and generates a PUI based on the content. The app is connected to a predefined number of remote networks by the means of PalCom tunnels, either over WIFI or the 3G network. It is through these tunnels that the app then finds and communicates with the target PalCom devices. Due to the architecture of the app it can be left running in the background whilst the phone is used for other purposes. Whenever an attention requiring (as defined by the PUID) PalCom command is received, the user's attention is captured by throwing an Android notification message. This feature was critical in order to allow the medical staff to use their phones for calling and the like, and still get informed when urgent alarm message are received via PalCom commands.

To clarify the concept of PUIDIs as illustrated by figure 6.1, the example introduced in chapter 4.2 will be used once more. In figure 6.2, the synthesized assembly in the example scenario is to be controlled by the Android PUIDI. The complete PUID used in the figure is presented in appendix E.1.1.

Even though the Android PUIDI is only in the prototype state, it still provides a lot of functionality, while remaining simple to operate. The app is complete enough to be used on a smaller scale, but before being used in any serious system, an appropriate amount of polish is required.



```
<?xml version="1.0" encodi

<puiml>
  <universe>
    <unit id="device" class="P:Device
      <unit id="synth-assembly" class=
        <unit id="motion" class="P:Con
          <unit id="motion-message" cl
        </unit>
        <unit id="snap" class="P:Comn
        <unit id="print" class="P:Comm
          <unit id="print-message" class
        </unit>
      </unit>
    </unit>
  </universe>

  <discovery>
    <property unit-name="device" na

    <property unit-name="synth-asse
    <property unit-name="synth-asse
    <property unit-name="synth-asse
    <property unit-name="synth-asse
    <property unit-name="synth-asse

    <property unit-name="motion" na
    <property unit-name="motion" na
    <property unit-name="motion" na
```

13:18

Photo Print

Motion dectection message:
Motion in doorway.

Print message:

Figure 6.2: PUID Interpreter concept for chapter 4.2 example scenario

6.3 Details

In this chapter the notion of PUID Interpreters has been discussed, and the two developed PUIDIs have been introduced and described. The aim of this chapter was to introduce *what* the two applications do, not *how* they do it. To get more details on the actual implementation of the applications, please consult appendix B. To learn about the installation of the applications, see appendix C. Finally, to learn how to use the applications in a proper manner, consult the manuals in appendix D.

Chapter 7

Validation

7.1 Introduction

In this chapter, the resulting products of this thesis will be validated to ensure that they fulfill the brief requirement presented in chapter 4.3. The aspects that will be validated are the produced language, PUIML, and the developed description interpreters. To provide a practical validation, the system created by Johansson and Persson [3], as presented in chapter 4.3.3, will be reconstructed to validate the collective power of PUIML and the Android PUID Interpreter.

7.2 Example of PUI use

To validate the power of the developed language/interpreter cooperation, the example presented in chapter 4.3.3 will be replicated using the Android PUID Interpreter. This part of the validation serves to show that the developed Android application can be used practically in a medical context.

7.2.1 System configuration

As mentioned in chapter 4.3.3, the entire alarm system converges into a single PalCom services which outputs the alarms and patient data. Such alarm systems would in a real scenario be a combination of several alarm trigger, medical databases and other integral components, all connected using PalCom. However, how all of these components interact to output the alarms with the right medical information to the right person holds no relevance in this example. The important factor is the end service presented by the alarm system, and for the purpose of this example, said service will be simulated using a simple Java application.

The simulated alarm system stores all produced messages and lets the user access them at a later time. To keep track of which message is currently being accessed, the system has an internal counter, *index*, which ranges from 0 (the first produced message) to $length - 1$ (the newest message), where *length* is the total number of produced messages. The alarm service communicates using the following commands:

- **ALARM : output**
  The command signals that a new alarm message has just been produced.
- **DATA : output**
  The command is used to output the actual content of the currently selected alarm message, as indicated by *index*.
  - `DATE : text`
    The date and time when the message was produced.
  - `P_NUMBER : text`
    The civic number of the patient which the message refers to.
  - `F_NAME : text`
    The first name of the patient which the message refers to.
  - `L_NAME : text`
    The last name of the patient which the message refers to.
  - `MEDICAL : text`
    Medical information about the patient which the message refers to. May contain multiple pieces of information, in which case each piece should be separated with the character ';'. E.g. *Allergic to nuts;Infected wound;In a coma*
  - `REF_NAME : text`
    The name of the person that forwarded the message.
  - `REF_PHONE : text`
    The phone number of the person that forwarded the message.
- **PREV : input**
  The command is used to request the previous message. When invoked, *index* is decreased by 1 if $index > 0$. The DATA command is invoked to output the message.
- **NEXT : input**
  The command is used to request the next message. When invoked, *index* is increased by 1 if $index < length - 1$. The DATA command is invoked to output the message.
- **RET_PHONE : output**
  The command is used to output the reference phone number of the currently selected alarm message, as indicated by *index*.
  - `REF_PHONE : text`
    The phone number of the person that forwarded the message.
- **GET_PHONE : input**
  The command is used to request the current reference phone number. When invoked, the RET_PHONE command is invoked to output the phone number.

When the simulated alarm system produces a new alarm message, the message along with its data is stored internally, *length* is increased by 1 and *index* is set to $length - 1$ to highlight the newest message. The ALARM command is then invoked to notify the right (in this case: *all*) users that a new alarm message has been produced. Lastly, the DATA command is invoked to output the actual message data to the users.

Since behavior in PUIML is defined as actions on PalCom entities, even the task of performing a phone call has to be handled using PalCom. For this purpose, a simple phone service containing the following command is defined:

- **DIAL : input**
  The command is used to dial the provided phone number.
  - `PHONE_NBR : text`
    The phone number to dial.

This alone, however, is not enough. Since the language doesn't have support for internal variables, there is no way to temporarily store the phone number provided by the REF_PHONE parameter of the DATA command. Because of this, there is no way of passing the reference phone number of the currently selected alarm message to the DIAL command. To get around this problem, a synthesized assembly containing the following command is defined:

- **DIAL : input**
  The command is used to dial the reference phone number of the currently selected alarm message.

When the DIAL command is invoked, the GET_PHONE command of the alarm service is invoked. The target phone number is then returned in the REF_PHONE parameter of the RET_PHONE command (alarm service). The acquired phone number is then used for the PHONE_NBR parameter, as the DIAL command is invoked on the phone service.

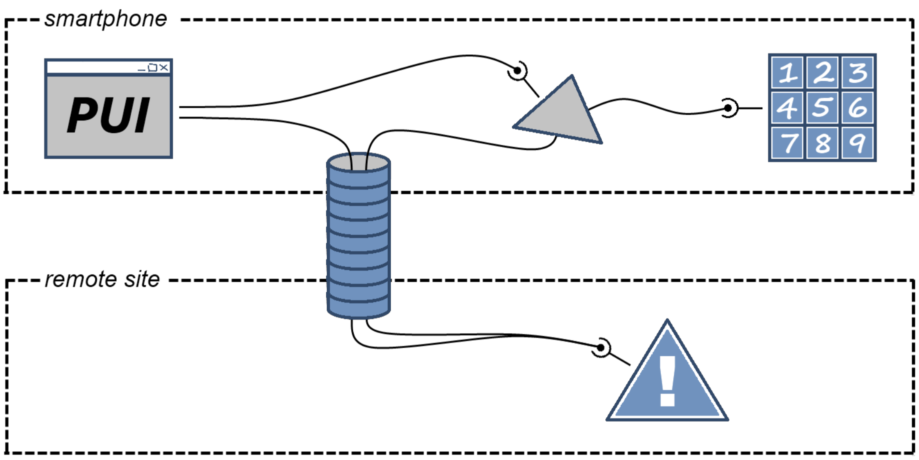


Figure 7.1: System setup of complete example

The complete system is depicted in figure 7.1. It is made up of five major components: the Android PUIDI, the dial assembly, the phone service, the simulated alarm service and a tunnel. The Android PUIDI (box labeled "PUI") will interpret the PUI Description and interact with the other components according to the description. The assembly (gray triangle), the phone service (numeric keypad) and the simulated alarm service (triangle labeled "!") have already been discussed above. Note that both the assembly and the phone service must run on the smartphone to access the native functionality of dialing phone numbers. However, the alarm service can be run on an arbitrary device, which might be connected to a different local network than the smartphone. This is illustrated by the dotted rectangles. The two networks are connected using a PalCom tunnel (blue tube).

Since there is currently no practical way of running custom services and assemblies on the Android platform, and since no additional functionality is showcased by including the dialing part of the demonstration, that part has been omitted from the actual validation system. The actual system setup used for the validation is displayed in figure 7.2. This is a justifiable simplification that will allow the demonstration to focus on the fundamental aspect of how the PUI communicates with the simulated alarm system, instead of how custom services and assemblies can be started on the Android platform.

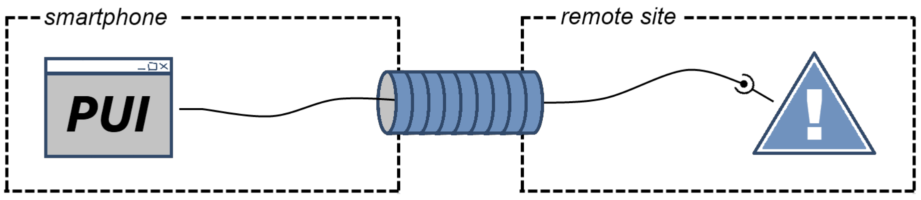


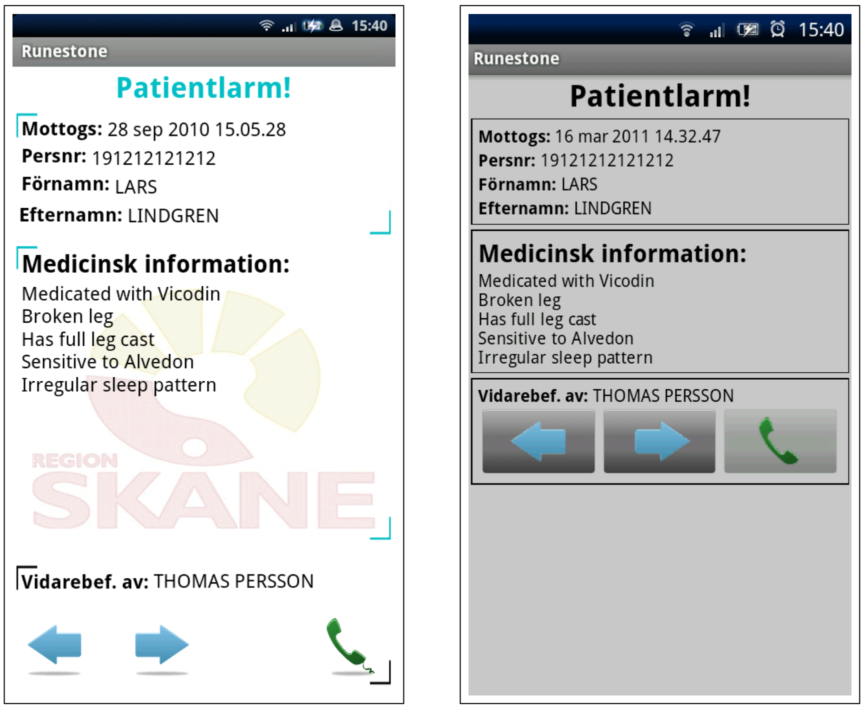
Figure 7.2: System setup of constructed example

7.2.2 Comparison

Having set up the system as described in the previous chapter section, the PUI Description that mimics the looks and behavior of the GUI described in chapter 4.3.3 is needed to complete the system. Said PUID is presented in its complete, unabbreviated form in appendix E.1.2, and the result of interpreting it with the Android PUID Interpreter is shown in figure 7.3(b). To validate the success of trying to replicate the *looks* of the original GUI, the two are presented side-by-side in figure 7.3.

Since the similarities by far outweigh the differences, let's focus on the latter. The first and perhaps most noticeable difference is that in figure 7.3(a) the background is white, where as in figure 7.3(b) it's gray. In figure 7.3(a) there is also a semi-opaque figure in the background. The color of the title text "Patientlarm!" is different between the two figures, and so are the borders of the different sections of the GUI. Lastly, in figure 7.3(b) the buttons have a classic button look, where as in figure 7.3(a) they are simply images. The common denominator for these differences is that they are all minor — it is a matter of personal preference and no alternative can be labeled as better than the other. However, there is one key difference that is of significance: in figure 7.3(a), the section with the three buttons is anchored to the button of the screen, whereas the corresponding section in figure 7.3(b) is anchored to the bottom of the middle section. The middle section displays the medical information about the patient as provided by the MEDICAL parameter of the DATA command of the simulated alarm service. The amount of medical information varies from one alarm message to the other, which in the case of the Android PUIDI produced PUI will result in different heights for the middle section. This in turn means that the section with the buttons won't have a fix position on the screen, as in the original GUI. One way of solving this problem would be to increase the expressive power of the language. A simpler work-around would be to simple put the button section under the top most section, which is of fixed size.

When comparing the two GUIs from a behavioral standpoint, the similarities are again predominant. Aside from not being able to call the staff member referenced in the alarm message (which was motivated in the previous chapter section), the two alternatives behave


15:40
Runestone
Patientlarm!
Mottogs: 28 sep 2010 15.05.28
Persnr: 191212121212
Förnamn: LARS
Efternamn: LINDGREN
Medicinsk information:
Medicated with Vicodin
Broken leg
Has full leg cast
Sensitive to Alvedon
Irregular sleep pattern
Vidarebef. av: THOMAS PERSSON

15:40
Runestone
Patientlarm!
Mottogs: 16 mar 2011 14.32.47
Persnr: 19121212121212
Förnamn: LARS
Efternamn: LINDGREN
Medicinsk information:
Medicated with Vicodin
Broken leg
Has full leg cast
Sensitive to Alvedon
Irregular sleep pattern
Vidarebef. av: THOMAS PERSSON

(a) Original GUI (b) Android PUIDI produced PUI

Figure 7.3: Visual comparison of GUIs

the same way. When an alarm is received, both applications notify the users with a Android notification, even if they are running in the background. Clicking said notification brings up the main GUI (figure 7.3) where the content of the alarm message is displayed in its proper place. Clicking the left-button loads the previous alarm message content into the proper place in the GUI, and clicking the right-button does the same for the next alarm message.

The comparison has shown that the GUIs are visually similar enough to support the statement that they are the same GUI in terms of looks alone. Even the somewhat limited functionality of the original GUI is mimicked as good as completely by the replica. To summarize, it stands clear that both of the presented GUIs are comparable not just terms of looks, but also in terms of functionality.

7.3 Validation of PUI Markup Language

To ensure the graphical expressiveness to be sufficient, in chapter 4.3.1 one requirement presented a list of Generic Graphical Components (GGC) that would have to be possible to represent using PUIML. In table 7.1 those GGCs are listed along with the class of the PUIML part that is used to represent them in PUIML. Another requirement for the language is that how the GGCs are laid out in the PUI must be controllable. As introduced in chapter 5, and as specified in appendix A.3, this is made possible by the means of nesting PUIML parts in conjunction with so-called layouts.

Part class	**Generic graphical component**
G:TextArea	Multi-lined text label
G:TextField	Single-lined text input box
G:Button	Clickable button
G:Image	Scalable image viewer
G:Area	Area that can hold other components
G:Tabbed	Tab control

Table 7.1: Part class equivalents of required GGCs

The first and most basic requirement on the behavioral expressiveness of the language was that all four PalCom entities must be representable using PUIML. Table 7.2 lists these entities along the class of the PUIML unit that represent the entity in PUIML.

Unit class	**PalCom entity**
P:Device	Device
P:Service	Service
P:Command	Command
P:Parameter	Parameter

Table 7.2: Unit class equivalents of required PalCom entities

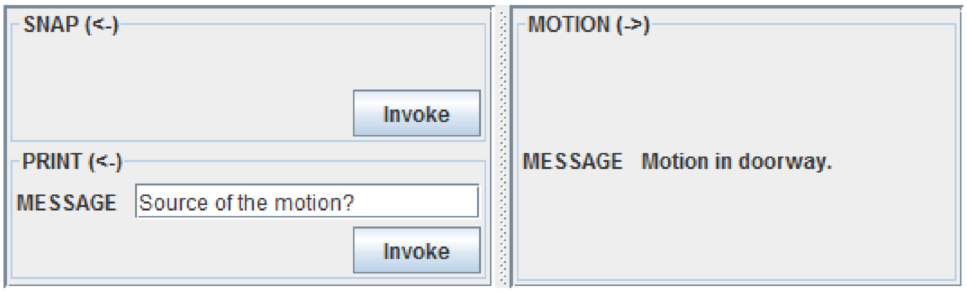
The other behavioral expressiveness requirements specify that the language should provide the means to set the value of PalCom parameters using selected GGCs, and reversely, display the value of PalCom parameters in selected GGCs. The language should also provide the means to invoke PalCom commands. These requirements are all met by PUIML, through the use of defining GGCs as either *provider*, *viewer* or *invoker*. These concepts were introduced and explained in chapter 5.2.6, and specifics for each PUIML part can be found in appendix A.2.

To ensure that the expressive power is at least on level with that of the BrowserGUI tool, the following requirement was specified:

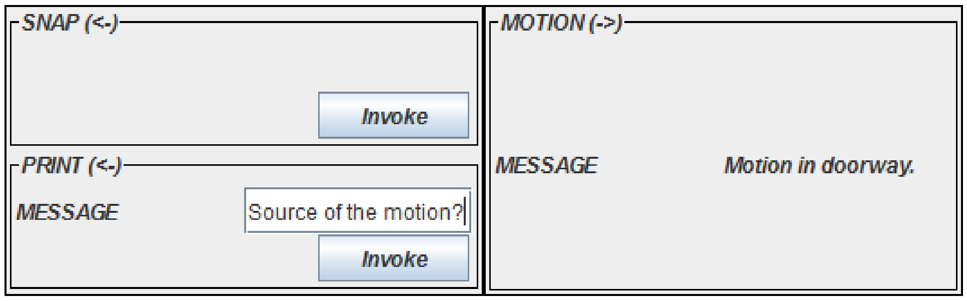
- The language should provide the means to express a GUI that is comparable in terms of looks and functionality to that of a GUI generated by the BrowserGUI tool.

To validate this abstract requirement, the example introduced in chapter 4.2 will once again be used. Figure 7.4(a) shows what the GUI provided by the BrowserGUI tool looks like when controlling the example service. As a comparison, figure 7.4(b) shows a PUI alternative, as defined by the PUI Description presented in appendix E.2.2, and as interpreted by the Swing/AWT PUIDI.

When comparing the two, one gets the overall impression that they are the same GUI. The differences are few, and what's more, the differences are minor, e.g. differing border



(a) Original BrowserGUI produced GUI



(b) Swing/AWT PUIDI produced PUI

Figure 7.4: Comparison of GUIs for chapter 4.2 example scenario

line color. Such differences could easily be addressed by expanding PUIML. The biggest difference, which is also of some importance, is how the components are spaced and sized. Notice, for example, that the (white) text box in figure 7.4(b) doesn't stretch all the way to the leading text "MESSAGE" as in figure 7.4(a). Such issues can usually be fixed by tweaking the PUID, but in some cases greater graphical expressiveness would have to be provided by the language.

The purpose of PUIML is not to describe PUIs that look the same as the GUIs produced by the BrowserGUI tool. The comparison in figure 7.4 is simply to illustrate that PUIML is indeed powerful enough to mimic the result of the BrowserGUI tool. One might argue that in the comparison of the two above, the BrowserGUI produced GUI is the superior one. However, this argument holds no merit, since if a GUI for the example scenario in chapter 4.2 was to be described using PUIML, the result would most likely look nothing like the PUI in figure 7.4(b). By taking advantage of the expressive power of PUIML, a more intuitive and aesthetically pleasing PUI could be described. Figure 7.5 shows an example of what such a PUI for the example scenario in chapter 4.2 could look like. The PUID used to produce this PUI is presented in appendix E.2.1. In a comparison between a properly described PUI (figure 7.5) and a BrowserGUI produced GUI (figure 7.4(a)), there should be no opposition to the statement that the PUI is the superior alternative.

7.4 Validation of PUID Interpreters

In chapter 4.3.2, requirements for the description interpreter applications, i.e. the PUID Interpreters, were presented. The first of these very basic requirements was that the

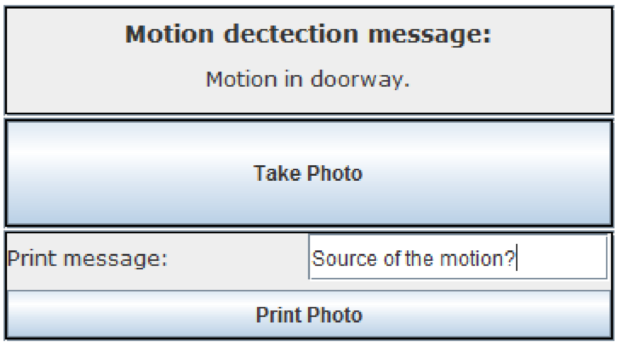


Figure 7.5: Intuitive PUI for chapter 4.2 example scenario

PUIDIs must allow the user to load one out of the available PUI Descriptions. As can be seen in appendix D, this is indeed very doable with both PUIDIs presented in chapter 6. The interpreters must also be able to interpret and display the PUIs described by the selected PUID. Again, very basic, and both PUIDIs meet this criteria. An example of a PUI created by the Swing/AWT PUIDI can be seen in figure 7.4(b), and a PUI created by the Android PUIDI is shown in figure 7.3(b). The produced PUIs must be able to connect to and communicate with PalCom entities, hence the name *PalCom* User Interface. This functionality is present in both PUIDIs, which can be verified by analyzing appendix D.

To ensure that an end user with a low level of expertise can operate the applications with ease, an accessibility requirement was introduced. It states that the user must be able to gain control over a predefined service, i.e. load a PUID, in less than two selective user operations (see chapter 4.3.2). As can be understood from studying chapter D, the Android PUIDI requires exactly two operations in both the best and the worst case scenario. The Swing/AWT PUIDI however require two operations only in the best case scenario. Depending on where the target PUID is located on the computers hard drive, the worst case scenario can require an undefined number of operations. However, since the Swing/AWT PUIDI is intended mostly for development and testing, this is acceptable.

Chapter 8

Future work

8.1 Introduction

Throughout the course of this thesis there have been instances where the limitations of the resulting solutions have been made clear. Most of these shortcomings are due to the time restriction on the master's thesis, and in this chapter they will be reviewed and presented as work that can be done in a possible follow-up to this thesis. Moreover, some additional aspects so far not discussed will also be introduced as possible future work.

8.2 PUI Markup Language extensions

In the case of the expressive power of the PUI Markup Language, there have been occurrences in this thesis where the power just wasn't enough, or where a PUI could have been improved in some way if the language was more powerful. One such example was seen in chapter 7.2.2, where the lower panel of an Android PUI (figure 7.3(b)) could not be anchored to the bottom of the screen as in the target GUI (figure 7.3(a)) — there is no way to express that layout in PUIML. The management of the balance between expressive power and simplicity in the language is no easy task, and the possible ways to extend the visual expressiveness of PUIML are numerous. Such extensions will be ignored in this discussion, which will instead concentrate on three possible improvements to the language itself.

8.2.1 Variables in PUIML

The need for internal variables in PUIML was identified during the validation in chapter 7.2, when trying to call the reference phone number provided by an alarm message. Since there currently is no way to temporarily store values in PUIML, there was no way to at a later point in time pass the same value (phone number) on the the dial service of the phone device. The problem was solved by adjusting the PalCom entities of the system, but introducing PUIML variables would allow for easier and more intuitive solutions.

8.2.2 Translatable constants in PUIML

This idea stems from the need to use the same PUI in multiple national regions, i.e. have the same PUI translated to multiple languages. Using current PUIML, each region would get its own PUI Description. These descriptions would be identical, except for the constant text strings that are displayed in the PUI. Since the same PUI is described in multiple files, this would result in *the double maintenance problem* [10]. If the PUI is to be changed in any way, the exact same changes have to be made in all PUID files. Except for the obvious hassle of having to apply the same changes to multiple files, there is also the problem that given enough time, the descriptions will start to differ. This is due to the fact that it is not feasible to apply exactly the same changes, and over time, many small differences will add up to big differences.

One possible solution for this problem is to introduce *translatable constants*. Instead of defining the constant text strings directly in the PUID file, the description would use references to constants that are specified in a separate file. Using this approach, the same PUID could be used for all national regions, and only the file containing the constants would have to be translated and maintained to create the PUI in a different language.

8.2.3 Property templates in PUIML

When describing PUIs it is not uncommon that groups of parts, i.e. graphical components, have many shared properties. It might for example be the case of a PUI that has several text labels that should have the same font properties, but display unique text strings. This would result in the same properties and property values being repeated multiple times in the PUI Description.

By introducing the concept of *property templates*, this issue could be resolved. The concept entails that one should be able to declare property templates and set properties for those templates, in the same way as one would with PUIML parts. The templates should then be usable as any other property for both parts and other templates. In the case of the PUI with the text labels, a property template specifying all font properties could be declared and used as a property for all the text labels in the PUI.

Introducing property templates would bring about two major advantages. The first is that by collecting shared properties in templates, the amount of PUIML code that has to be written will be decreased. This will result in shorter and more easily understandable PUIDs. The second advantage is that by collecting properties shared by multiple parts in a template, all parts affected can be updated by simply changing the values in one place.

The same could be applied to PUIML units.

8.3 Android PUID Interpreter

To make use of the native functionality of the Android OS in a PUI Description, the functionality has to be presented as a PalCom service. An example of this was seen in chapter 7.2. In order for the PUI to be able to use the phone calling functionality of the Android OS, it had to be represented as a PalCom service that took the phone number as a parameter. Unfortunately, there is currently no sustainable method for loading custom services (or assemblies) on the Android OS. Hence, one possible point of interest for future development would be the ability to:

- Run custom PalCom services/assemblies on Android OS

The Android PUID Interpreter currently only lets the user load one PUID at a time. Depending on how the PUIDs are designed, the need to load more than one PUID might arise. It might be the case of a physician that moves between several hospital departments, all of which have their own alarm system. In such a case, the physician would most likely have one PUID installed for each department's alarm system. Currently, s/he would have to load the corresponding PUID when switching department, and would for the duration not be able to receive alarms from the alarm systems of the other departments. To address this issue, the following would be desirable:

- Allow multiple PUIs to run simultaneously

8.4 Graphical PUI Description editor

Currently, the only method of creating and editing PUI Descriptions, i.e. designing PUIs, is to manually edit the XML file using a text editor. This method is manageable for people with a higher level of technical expertise. The person needs to have a good understanding of both XML and PUIML. Unfortunately, the ones that will be designing the PUIs will most likely be people with a high expertise in design, but with low technical expertise, i.e. *designers*. During the development of PUIML and the associated PUID Interpreters, the text editor method of designing PUIs was sufficient, since only the developer (the present author) created PUIs. However, if the project was to be scaled up, one would have to consider the creation of a graphical editor to create/edit PUIDs as a possible continuation.

8.5 Real-life application

A natural continuation on the work established in this thesis would be to use the outcome in real situations. In chapter 7.2, the Android PUID Interpreter was used to create a PUI for the alarm system developed by Johansson and Persson [3]. The PUIDI was hence used in a real context, but with a simulated environment. It would be of great interest to use PUIML and the Android PUIDI to interact with the actual PalCom alarm system, not just a simulated version. It may also be of interest to develop PUIs for currently unknown PalCom systems. This would not only further acknowledge the power of PUIML, but also provide new ideas of how PUIML can be improved in the future.

8.6 PalCom integration

Currently, the two developed PUID Interpreters are separate from the PalCom framework introduced in chapter 3.3. Both are stand-alone application which are started explicitly. A PUI Description is then chosen and the resulting PUI is displayed. A possibility for future work is to integrate the concept of PUIs into the framework itself. The simplest way of doing this is to extend the PalCom native tool *TheThing*. TheThing is used to host services, assemblies and tunnels. By extending TheThing to also being able to host one or more PUIDs, the corresponding PUIs could be started much like the other PalCom entities of TheThing.

So far, creating and using PUIs has been a *linear activity*: The designer creates the PUID, and the user uses it to get a PUI with which s/he can interact. One possible way of integrating the concept of PUIs into the PalCom framework is by making the creation and usage of PUIs a *spontaneous activity*. The idea is that using a PalCom native tool, the user can browse through the available service. When the desired service has been located, the user is presented with all PUIDs that control the service. By choosing a PUID, the user is not only presented with a PUI with which to interact, but can also spontaneously change the PUID to their liking. In a similar manner, if no PUID is present for a given service, the user can spontaneously create a new one. By using this approach, the line between creator and user of PUIs is made blurry.

The two ways of integrating the concept of PUIs into the PalCom framework presented above are just ideas on how the integration can be done. However, it stands clear that the integration is of great interest for future work.

8.7 Summary

There are plenty of possible ways to extend the outcome of this thesis. The ones presented in this chapter are just a few of the ideas that emerged during the course of the project, and there are certainly many more that remain undetected for the time being.

Chapter 9

Evaluation

The aim of this thesis is to resolve the problems associated with creating GUIs for smartphones, as presented in chapter 2. The primary outcome of addressing these problems is the Android PUID Interpreter, which is an Android application that creates custom PUIs based on descriptions. Such PUI Descriptions are specified using the developed language (PUIML). The application, in combination with the developed language, provides simplicity for three different categories of users:

PUI designers The designers of the PUIs that will be used by the end user are presented with a short and concise language that enables them to describe simple, yet powerful PUIs. The language alone is sure to speed up the design process. Furthermore, the big gain is that all the effort that goes into developing a new Android app from scratch is reused for each new scenario. This is possible because the Android PUIDI interprets PUIDs that are specific to a certain scenario, hence creating a scenario specific Android PUI. This way, the designer needs not concern himself with platform specifics, only the PUID.

System administrators The Android PUIDI is self-identifying, which means that the administrator doesn't have to manually "name" each device that will be running the interpreter. The ID of the device is based on the serial number of the phone's SIM card. This allows the user to keep his/her identity in the PalCom universe, even if s/he for some reason has to switch to another phone. Since both the tunnel configuration and the PUI Descriptions are kept in regular files, installation of a certain PUI on several devices is as simple as copying the same files to all devices. These small but important perks will most likely save more than a little time when installing large systems.

End users The end user, which might be part of the medical staff at a hospital, should be considered a technically inexperienced user. Therefore, the interface of the Android PUIDI is kept simple to avoid confusion. All the end user has to do is start the Android app, and select the desired PUI Description. This act can be made really simple (from the system administrator's side) by providing just one description. After the PUID selection, the user friendliness of the generated PUI all depends on how well the PUI designer did his/her job.

One of the problems addressed in this thesis is that the amount of resources that goes into creating GUIs for smartphones should be kept as low as possible. As mentioned above, this is aided by the fact that by using the Android PUIDI, the same common base is reused for all created PUIs. To evaluate how well the system performs at keeping the amount of resources needed at a minimum, the Android application developed by Johansson and Persson [3] will once again be used.

The GUI for the application is presented in figure 7.3(a). Next to it, in figure 7.3(b), the Android PUIDI created PUI equivalent is presented. It is not possible to compare the amount of time that went into creating the two, since the data is not available for either case. However, the amount of code that makes up the two solutions can be compared. Based on information provided by the creators, the code used to create the GUI in figure 7.3(a) is made up of 1482 words. The PUID that describes the PUI alternative in figure 7.3(b) is listed in appendix E.1.2, and the PUIML code consists of 442 words. Given that this is only a comparison for one case, it is still clear that the the amount code needed to create a practically usable GUI is significantly smaller when using PUIML and the Android PUIDI, as opposed to creating an ad-hoc GUI from scratch. Naturally, the amount of code could be reduced even further, but the balance between expressiveness and simplicity has to be kept at a practical level.

Another interesting point of evaluation is the comparison between how GUIs are created using the developed language, and how they are created using the GUI builder tool of the NetBeans IDE. As mentioned in chapter 8, an intuitive continuation for this thesis would be to create a graphical editor for PUI Descriptions. Taking this into consideration, creating PUIs using the proposed PUID editor would be just as simple as using the NetBeans builder to create GUIs.

One advantage of using PUIML is that nothing other than what the PUI should look like and how it should behave has to be specified. The same PUIDI, which provides the application basis and means for PalCom communication, can be reused for all PUI Descriptions. When using a tool like NetBeans, not only does the developer have to design the GUI for each new scenario, s/he must also rebuild the application basis and PalCom communication module to fit the new scenario. While some of the code of previous solutions can surely be reused, the amount of work that goes into tweaking and integrating the old code into the new solution should still be considered significant. Another advantage of using PUIML is that the means for interacting with PalCom services is built right into the language. This has the effect that no unnecessary glue code has to be produced when specifying the behavior of components. When using tools like NetBeans, the developer will most likely have to produce glue code simply to link the GUI component to the code of a PalCom communication module. This kind of code is usually simple to produce, but also very time consuming and tedious.

Because of the generic nature of PUIML, the language can be used to specify PUIs for many different target platform, provided that a PUIDI for said platform is available. When using tools like NetBeans, the developer is usually limited to one or a few platforms. Furthermore, the different platforms have different characteristics, and may require that the developer learns a new language, or how to use a specific set of packages provided for the platform.

As has been shown throughout the course of this thesis, the result of the project is an application that produces PalCom based Android GUIs in an accessible and sustainable way, using the developed language PUIML. The issues presented in chapter 2 are addressed, and the future possibilities are plentiful.

Chapter 10

Conclusion

The need for new and better alarm devices in the health care sector has lead to the need to create many similar smartphone applications using a limited amount of resources. One of the goals of this thesis was to achieve this effect by using a common service based system as a foundation. By using the PalCom framework, all medical equipment and the respective functionality can be represented as self-describing devices and services within the system. Even though different hospitals and even different departments in the same hospital needs to combine different services for different effects, they can all be based on PalCom. This ensures that the core of all smartphone applications that seeks to connect to the system are essentially the same. By using a common application core, the only thing that has to be recreated between different scenarios is the GUI.

To further reduce the amount of resources required to create smartphone applications, the GUI language PalCom User Interface Markup Language (PUIML) was developed. PUIML is platform generic, meaning that the GUIs specified for an Android smartphone should be portable to other platforms in the future. By using a generic language, the complexity of the GUI descriptions are kept low. Building on the fact that the target systems of this thesis were based on PalCom, the behavioral possibilities of the PUIML described GUIs were limited to simply being able to influence PalCom entities. While this limits the power of PUIML, the gain is that the specialized behavior is simple to express, and the GUI description are therefore kept short.

By combining the fact that the target systems are PalCom based with the language developed for the purpose of this thesis, a prototype Android application was developed as the primary result of this thesis. This application allows the user to choose one file (from the phones SD-card) that describes a GUI using PUIML. The description describes what the GUI should look like, and which services in the system should make up the functionality of the application. After a description file is chosen, the GUI is generated, and the user can start the interaction.

From the user's point of view, this is an application like any other. However, for a GUI developer, the advantages are significant. Instead of having to specialize in platform specific languages and modules, the developer is presented with a short and concise language — PUIML. Since the target system is PalCom based, defining the functionality of the GUI is as simple as hooking a graphical component, e.g. a button, to the desired PalCom service. This is made possible by the unique features of PUIML. Perhaps the biggest gain is that while a new GUI has to be defined using PUIML for every new scenario, the core of the Android application is reused every time. This significantly reduces the amount of code that has to be written, and therefore, the amount of resources that is needed to create a custom GUI.

By combining the concept of service based systems with a platform generic GUI language, this thesis has resulted in an Android applications with a context adaptable GUI. It stands clear that the amount of resources required to create a custom Android GUI has been significantly reduced, and in the future the system could easily be ported to other platforms.

Bibliography

[1] Interface Builder: Create your user interface. http://www.apple.com/macosx/developers/#builder, 2011. [Online; accessed 27-April-2011].

[2] NetBeans IDE: Swing GUI builder. http://netbeans.org/features/java/swing.html, 2011. [Online; accessed 27-April-2011].

[3] Erik Johansson and Thomas Persson. Flexible integration of medical systems using PalCom. Master's thesis, Lund University, 2010.

[4] R. Jason Weiss and J. Philip Craiger. Ubiquitous computing. *The Industrial-Organizational Psychologist*, 39(4), April 2002.

[5] Giovanni Rimassa, Dominic Greenwood, and Monique Calisti. Palpable computing and the role of agent technology. In *Proceedings of the 4th International Central and Eastern European Conference on Multi-Agent Systems*, Budapest, Hungary, September 2005.

[6] P. Andersen and S. Bo Larsen. PalCom external report 70: Developer's companion. March 2009.

[7] What is UsiXML ? http://www.usixml.org/index.php?mod=pages&id=2, 2011. [Online; accessed 07-March-2011].

[8] Peter Bojanic. The joy of XUL. https://developer.mozilla.org/en/The_Joy_of_XUL, 2011. [Online; accessed 07-March-2011].

[9] Marc Abrams and James Helms. User interface markup language (UIML) specification. March 2004.

[10] Wayne A. Babich. *Software Configuration Management*. Addison-Wesley, 1986.

Appendix A

PUIML specification

A.1 Units

A.1.1 P:Device

Units of class ”P:Device” represent a PalCom device (see chapter 3.3.3). The PalCom device is identified in the PalCom universe using the discovery properties listed in table A.1. An example on how to use the class is presented in listing A.1.

Property	**Value type**	**Default value**	**Description**
p:id	`String`	–	Sets the ID of the device.

Table A.1: Required discovery properties for the P:Device class

```
<unit id="my_id" class="P:Device">
  <discovery>
    <property name="p:id">X:test</property>
  </discovery>
</unit>
```

Listing A.1: Example use of the P:Device class

A.1.2 P:Service

Units of class ”P:Service” represent a PalCom service (see chapter 3.3.4). The PalCom service is identified in the PalCom universe using the discovery properties listed in table A.2 and A.3, in conjunction with the discovery properties found in the units parent device. An example on how to use the class is presented in listing A.2.

Property	**Value type**	**Default value**	**Description**
p:pdid	`String`	–	Sets the previous device ID of the service.
p:mdid	`String`	–	Sets the merged-from device ID of the service. **Note:** only applicable if the p:pdid property is set.

Table A.2: Optional discovery properties for the P:Service class

Property	**Value type**	**Default value**	**Description**
p:instance	`String`	–	Sets the instance number of the service.
p:cdid	`String`	–	Sets the creating device ID of the service.
p:cn	`String`	–	Sets the creation number of the service.
p:udid	`String`	–	Sets the updating device ID of the service.
p:un	`String`	–	Sets the update number of the service.
p:pn	`String`	–	Sets the previous number of the service. **Note:** only applicable if the p:pdid property is set.
p:mn	`String`	–	Sets the merged-from number of the service. **Note:** only applicable if the p:mdid property is set.

Table A.3: Required discovery properties for the P:Service class

```
<unit id="my_id" class="P:Service">
  <discovery>
    <property name="p:instance">1</property>
    <property name="p:cdid">X:test</property>
    <property name="p:cn">0</property>
    <property name="p:udid">X:test</property>
    <property name="p:un">1</property>
  </discovery>
</unit>
```

Listing A.2: Example use of the P:Service class

A.1.3 P:Command

Units of class ”P:Command” represent a PalCom command (see chapter 3.3.4). The PalCom command is identified in the PalCom universe using the discovery properties listed in table A.4 and A.5, in conjunction with the discovery properties found in the units ancestors. An example on how to use the class is presented in listing A.3.

Property	**Value type**	**Default value**	**Description**
p:direction	{ in | out }	in	Sets the direction of the command.
p:notifications	{ off, xlow, low, normal, high, xhigh }	off	Sets the level at which the user will be notified when the command is invoked.

Table A.4: Optional discovery properties for the P:Command class

Property	**Value type**	**Default value**	**Description**
p:id	`String`	–	Sets the ID of the command.

Table A.5: Required discovery properties for the P:Command class

```
<unit id="my_id" class="P:Command">
  <discovery>
    <property name="p:id">out_com_1</property>
    <property name="p:direction">out</property>
    <property name="p:notifications">low</property>
  </discovery>
</unit>
```

Listing A.3: Example use of the P:Command class

A.1.4 P:Param

Units of class ”P:Param” represent a PalCom parameter (see chapter 3.3.4). The PalCom parameter is identified in the PalCom universe using the discovery properties listed in table A.6 and A.7, in conjunction with the discovery properties found in the units ancestors. An example on how to use the class is presented in listing A.4.

Property	**Value type**	**Default value**	**Description**
p:data-type	{ text | image }	text	Sets the data type of the command.

Table A.6: Optional discovery properties for the P:Param class

Property	**Value type**	**Default value**	**Description**
p:id	`String`	–	Sets the ID of the command.

Table A.7: Required discovery properties for the P:Param class

```
<unit id="my_id" class="P:Param">
  <discovery>
    <property name="p:id">param_1</property>
    <property name="p:data-type">image</property>
  </discovery>
</unit>
```

Listing A.4: Example use of the P:Param class

A.2 Parts

A.2.1 G:TopContainer

The generic top container represents the root container on the target platform, in which all other parts will be housed. Consequently, the first part of any PUI Description must be of class ”G:TopContainer”. The available style properties for parts of this class are listed in table A.8. For an example on how to use the class and its style properties, refer to listing A.5.

```
<part id="my_id" class="G:TopContainer">
  <style>
    <property name="g:title">Test title</property>
    <property name="g:resizable">false</property>
    <property name="g:size">320,240</property>
  </style>
</part>
```

Listing A.5: Example use of the G:TopContainer class

Property	**Value type**	**Default value**	**Description**
g:title	`String`	–	Sets the text that is displayed as the title of the top container.
g:resizable	{ true | false }	true	Sets whether or not the top container should be allowed to be resized.
g:layout	{ linear | grid }	linear	Sets the layout of the container part. See appendix A.3.
g:size	width:`Integer`, height:`Integer`	–	Set the **preferred** size (in pixels) of the part.

Table A.8: Available style properties for the G:TopContainer class

A.2.2 G:Area

The generic area container represents some area on the target platform in which other parts can be contained. Area containers can contain parts of any class (with the exception of ”G:TopContainer”), including other parts of class ”G:Area”, making them perfect not only for grouping together associated parts of the interface, but also for structuring complex interfaces. The available style properties for area containers are listed in table A.9, and an example of how they can be used is represented by listing A.6.

Property	**Value type**	**Default value**	**Description**
g:title	`String`	–	Sets the text that is displayed in the border (if any) of the area.
g:scrollable	{ true | false }	false	Sets whether or not the area should be scrollable if too big.
g:border	{ empty | line | raised | lowered }	empty	Sets the type of border that should surround the area.
g:layout	{ linear | grid }	linear	Sets the layout of the container part. See appendix A.3.
g:size	width:`Integer`, height:`Integer`	–	Set the **preferred** size (in pixels) of the part.
g:tab-text	`String`	–	Sets the text on the tab for this part, given that it is a sub-part of a part of class ”G:Tabbed” (see appendix A.2.3).

Table A.9: Available style properties for the G:Area class

```
<part id="my_id" class="G:Area">
  <style>
    <property name="g:title">Border title</property>
    <property name="g:scrollable">true</property>
    <property name="g:border">line</property>
    <property name="g:size">160,120</property>
    <property name="g:tab-text">Tab title</property>
  </style>
</part>
```

Listing A.6: Example use of the G:Area class

A.2.3 G:Tabbed

The generic tab container represents some component on the target platform in which other parts can be contained. Each part (which may be of any class except for ”G:TopContainer”) contained is representing as a tab in the tab container. The available style properties for tab containers are listed in table A.10. For an example on how to use these properties, refer to listing A.7.

Property	**Value type**	**Default value**	**Description**
g:size	width:`Integer`, height:`Integer`	–	Set the **preferred** size (in pixels) of the part.
g:tab-text	`String`	–	Sets the text on the tab for this part, given that it is a sub-part of a part of class ”G:Tabbed” (see appendix A.2.3).

Table A.10: Available style properties for the G:Tabbed class

```
<part id="my_id" class="G:Tabbed">
  <style>
    <property name="g:size">160,120</property>
    <property name="g:tab-text">Tab title</property>
  </style>
</part>
```

Listing A.7: Example use of the G:Tabbed class

A.2.4 G:Label

The generic label represents a simple single line label. These labels can be used either as static information sources, or as viewers for units of class ”P:Param” that have the p:data-type property set to ”text” (see appendix A.1.4). The visual appearance of the label can be altered using the properties in table A.12, and the behavior using the properties displayed in table A.11. For an example use, please refer to listing A.8.

Property	**Value type**	**Default value**	**Description**
p:viewer	Unit ID	–	Sets the unit ID for which this part will be a viewer.

Table A.11: Available behavior properties for the G:Label class

Property	**Value type**	**Default value**	**Description**
g:font	`String`	–	Sets the font for this part. Valid font names may vary on different platforms. Invalid font names will have no effect. See section A.4.
g:text	`String`	–	Sets the initial text string of the part.
g:align-h	{ left | center | right }	left	Sets the horizontal alignment of the text.
g:align-v	{ top | center | bottom }	center	Sets the vertical alignment of the text.
g:size	width:`Integer`, height:`Integer`	–	Set the **preferred** size (in pixels) of the part.
g:tab-text	`String`	–	Sets the text on the tab for this part, given that it is a sub-part of a part of class ”G:Tabbed” (see appendix A.2.3).

Table A.12: Available style properties for the G:Label class

```
<part id="my_id" class="G:Label">
  <style>
    <property name="g:text">top-center</property>
    <property name="g:align-h">center</property>
    <property name="g:align-v">top</property>
    <property name="g:size">120,50</property>
    <property name="g:tab-text">Tab title</property>
  </style>
  <behavior>
    <property name="p:viewer">unit_id</property>
  </behavior>
</part>
```

Listing A.8: Example use of the G:Label class

A.2.5 G:TextArea

The generic text area represents a multi line text area. As the parts of class "G:Label", the parts of class "G:TextArea" can also be used either as static information sources, or as viewers for units of class "P:Param" that have the p:data-type property set to "text" (see appendix A.1.4). The available style properties for text areas are listed in table A.14, and the available behavior properties are listed in table A.13. An example use of these properties can be studied in listing A.9.

Property	**Value type**	**Default value**	**Description**
p:viewer	Unit ID	–	Sets the unit ID for which this part will be a viewer.
p:delimiter	`String`	–	Sets the delimiter characters used to separate text into separate lines. **Note:** only applicable if the p:viewer property is set.

Table A.13: Available behavior properties for the G:TextArea class

```
<part id="my_id" class="G:TextArea">
  <style>
    <property name="g:text">Some text</property>
    <property name="g:size">120,200</property>
    <property name="g:tab-text">Tab title</property>
  </style>
  <behavior>
    <property name="p:viewer">unit_id</property>
    <property name="p:delimiter">;:</property>
  </behavior>
</part>
```

Listing A.9: Example use of the G:TextArea class

Property	**Value type**	**Default value**	**Description**
g:font	`String`	–	Sets the font for this part. Valid font names may vary on different platforms. Invalid font names will have no effect. See section A.4.
g:text	`String`	–	Sets the initial text string of the part.
g:size	width:`Integer`, height:`Integer`	–	Set the **preferred** size (in pixels) of the part.
g:tab-text	`String`	–	Sets the text on the tab for this part, given that it is a sub-part of a part of class ”G:Tabbed” (see appendix A.2.3).

Table A.14: Available style properties for the G:TextArea class

A.2.6 G:TextField

The generic text field represents a simple single line text field to enter textual data. Parts of class ”G:TextField” can be used either as providers for units of class ”P:Param” that have the p:data-type property set to ”text” (see appendix A.1.4), or as non editable viewers for the same. The class' style properties are listed in table A.16, and its behavior properties in table A.15. For an example use, please refer to listing A.10.

Property	**Value type**	**Default value**	**Description**
p:provider	Unit ID	–	Sets the unit ID for which this part will be a provider.
p:viewer	Unit ID	–	Sets the unit ID for which this part will be a viewer. **Note:** only applicable if the p:provider property is not set.

Table A.15: Available behavior properties for the G:TextField class

A.2.7 G:Image

The generic image represents any component that displays images on the target platform. Images can be used either for static image displaying, or as viewers for units of class ”P:Param” that have the p:data-type property set to ”image” (see appendix A.1.4). The visual properties of this class are listed in table A.18, while the behavioral properties are listed in table A.17. For an example on how to use the G:Image class, consult listing A.11.

Property	**Value type**	**Default value**	**Description**
g:text	`String`	–	Sets the initial text string of the part.
g:alternate-text	`String`	–	Sets the alternate text (usually a tool tip text) string for the part.
g:align-h	{ left | center | right }	left	Sets the horizontal alignment of the text.
g:size	width:`Integer`, height:`Integer`	–	Set the **preferred** size (in pixels) of the part.
g:tab-text	`String`	–	Sets the text on the tab for this part, given that it is a sub-part of a part of class ”G:Tabbed” (see appendix A.2.3).

Table A.16: Available style properties for the G:TextField class

```
<part id="my_id" class="G:TextField">
  <style>
    <property name="g:text">initial txt</property>
    <property name="g:alternate-text">tooltip text</property>
    <property name="g:align-h">center</property>
    <property name="g:size">120,50</property>
    <property name="g:tab-text">Tab title</property>
  </style>
  <behavior>
    <property name="p:provider">unit_id</property>
  </behavior>
</part>
```

Listing A.10: Example use of the G:TextField class

Property	**Value type**	**Default value**	**Description**
p:viewer	Unit ID	–	Sets the unit ID for which this part will be a viewer.

Table A.17: Available behavior properties for the G:Image class

Property	**Value type**	**Default value**	**Description**
g:image-src	`String`	–	Sets the image source of this part. Must be a valid filename. See appendix A.5.
g:size	width:`Integer`, height:`Integer`	–	Set the **preferred** size (in pixels) of the part. Any of `width` and `height` may be set to a negative number in order to resize proportionally.
g:tab-text	`String`	–	Sets the text on the tab for this part, given that it is a sub-part of a part of class "G:Tabbed" (see appendix A.2.3).

Table A.18: Available style properties for the G:Image class

```
<part id="my_id" class="G:Image">
  <style>
    <property name="g:image-src">C:/images/img.png</property>
    <property name="g:size">180,-1</property>
    <property name="g:tab-text">Tab title</property>
  </style>
  <behavior>
    <property name="p:viewer">unit_id</property>
  </behavior>
</part>
```

Listing A.11: Example use of the G:Image class

A.2.8 G:Button

The generic button represents a button on the target platform which accepts input from the user in the form of mouse clicks. Parts of class "G:Button" can be used either to invoke units of class "P:Command" that have the p:direction property set to "in" (see appendix A.1.3), or as providers for units of class "P:Param" that have the p:data-type property set to "image" (see appendix A.1.4). As provider, buttons have two different *targets*: **browse**, which uses a file browser to locate an image, and **camera**, which uses the camera (if available). The class' style properties are listed in table A.20, and its behavior properties in table A.19.

Listing A.12 demonstrates an example where a button displays an image instead of a normal text label and is used as an invoker. In listing A.13, a button with a normal text label uses the camera to provide images.

Property	**Value type**	**Default value**	**Description**
p:invoker	Unit ID	–	Sets the unit ID for which this part will be an invoker.
p:provider	Unit ID	–	Sets the unit ID for which this part will be a provider. **Note:** only applicable if the p:invoker property is not set.
p:target	{ browse | camera }	browse	Sets the invoker target. **Note:** only applicable if the p:provider property is set.

Table A.19: Available behavior properties for the G:Button class

Property	**Value type**	**Default value**	**Description**
g:text	`String`	–	Sets the text label for the button.
g:alternate-text	`String`	–	Sets the alternate text (usually a tool tip text) string for the part.
g:image-src	`String`	–	Sets the image source of this part. Must be a valid filename. See appendix A.5.
g:size	width:`Integer`, height:`Integer`	–	Set the **preferred** size (in pixels) of the part.
g:tab-text	`String`	–	Sets the text on the tab for this part, given that it is a sub-part of a part of class "G:Tabbed" (see appendix A.2.3).

Table A.20: Available style properties for the G:Button class

```
<part id="my_id" class="G:Button">
  <style>
    <property name="g:image-src">C:/images/img.png</property>
    <property name="g:alternate-text">tooltip text</property>
    <property name="g:size">120,50</property>
    <property name="g:tab-text">Tab title</property>
  </style>
  <behavior>
    <property name="p:invoker">unit_id</property>
  </behavior>
</part>
```

Listing A.12: Example use of the G:Button class (as invoker)

```
<part id="my_id" class="G:Button">
  <style>
    <property name="g:image-src">press me</property>
    <property name="g:size">120,50</property>
    <property name="g:tab-text">Tab title</property>
  </style>
  <behavior>
    <property name="p:provider">unit_id</property>
    <property name="p:target">camera<property>
  </behavior>
</part>
```

Listing A.13: Example use of the G:Button class (as provider)

A.3 Layout properties

The g:layout property is a style property that can be set for parts of class ”G:TopContainer” (appendix A.2.1) and parts of class ”G:Area” (appendix A.2.2). The property has two valid values: ”linear” and ”grid”. Depending on the value of g:layout, additional properties can, and even must, be set. These additional properties when g:layout is set to ”linear” are specified in table A.21, and when g:layout is set to ”grid” are specified in table A.22 and table A.23.

Property	**Value type**	**Default value**	**Description**
g:layout-gap	x:`Integer`, y:`Integer`	–	Sets the gap (in pixels) between the parts in this layout.
g:layout-orientation	{ horizontal | vertical }	horizontal	Sets the direction from which the parts in this layout are laid out.

Table A.21: Optional style properties when g:layout is set to ”linear”

Property	**Value type**	**Default value**	**Description**
g:layout-gap	x:`Integer`, y:`Integer`	–	Sets the gap (in pixels) between the parts in this layout.

Table A.22: Optional style properties when g:layout is set to ”grid”

Property	**Value type**	**Default value**	**Description**
g:layout-columns	`Integer`	–	Sets the number of columns per row for the grid.

Table A.23: Required style properties when g:layout is set to ”grid”

Setting the g:layout property to ”linear” for a container part means that all sub-parts will be positioned according to a *linear layout*. This means that all sub-parts will be laid out one after another according to the g:layout-orientation property, as either left-to-right (g:layout-orientation set to ”horizontal”), or top-to-bottom (g:layout-orientation set to ”vertical”). Setting the g:layout property to ”grid” means that a *grid layout* will be used to position the sub-parts. In a grid layout the sub-parts are laid out into a grid. The proportions of the grid cells are set automatically and are dependent on the implementing platform.

```
1  <part id="my_id" class="G:Area">
2    <style>
3      <property name="g:layout">grid</property>
4      <property name="g:layout-gap">5,5</property>
5      <property name="g:layout-columns">2</property>
6      ...
7    </style>
8  </part>
```

Listing A.14: Example use of a grid layout

Listing A.14 demonstrates the use of a grid layout. In the example, a grid with 2 columns per row will be created, and the gap between the added sub-parts will be 5 pixels horizontally and 5 pixels vertically. Notice that while line 4 i the example could be removed, line 5 is mandatory.

```
<part id="my_id" class="G:Area">
  <style>
    <property name="g:layout">linear</property>
    <property name="g:layout-gap">5,5</property>
    <property name="g:layout-align">vertical</property>
    ...
  </style>
</part>
```

Listing A.15: Example use of a linear layout

Listing A.15 demonstrates the use of a linear layout. In the example, sub-parts will be added to the layout from top to bottom, meaning that the first sub-part added will be the topmost one. As in the previous example, the gap between the added sub-parts will be 5 pixels horizontally and 5 pixels vertically.

A.4 Font properties

The g:font property is a style property that can be set both for parts of class ”G:Label” (appendix A.2.4) and parts of class ”G:TextArea” (appendix A.2.5). The value of the property should be a text string describing the name of the desired font. As valid font names may vary depending on the target platform, some tweaking might be necessary to get the desired effect. However, if the font name is invalid, no errors will occur. The default font will simply be used instead, as if no value was set for g:font.

In addition to the g:font property there are a few other font related properties that can be set for the above mentioned part classes. These properties are listed in table A.24. Listing A.16 demonstrates how to set the font to italic (but not bold) verdana of size 12.

Property	**Value type**	**Default value**	**Description**
g:font-bold	{ true | false }	false	Sets whether the font should be in bold.
g:font-italic	{ true | false }	false	Sets whether the font should be in italics.
g:font-size	`Integer`	–	Sets the size of the font.

Table A.24: Additional font related properties

```
<part id="my_id" class="G:Label">
  <style>
    <property name="g:font">Verdana</property>
    <property name="g:font-bold">false</property>
    <property name="g:font-italic">true</property>
    <property name="g:font-size">12</property>
    ...
  </style>
</part>
```

Listing A.16: Example use of font properties

A.5 Filename values

The g:image-src property is a style property that can be set for parts of class ”G:Button” (appendix A.2.8) and parts of class ”G:Image” (appendix A.2.7). The value of the property must be a text string that represents the filename of the image source. The filename may be absolute, e.g. ”c:/folder1/folder2/image.png”, or relative, e.g. ”folder2/image.png”. Worth noting is that when presented with a relative filename, the parser will actually perform two checks:

1. Using the path of the file being parsed as a starting-point, the parser appends the folder name ”resources” as well as the relative filename, and attempts to use the newly formed absolute filename to locate the file.
2. If the first check failed, the parser will attempt to use the default relative filename handler provided by the Java framework.

For example:

> The parser, running out of the folder ”c:/folder1”, is parsing the file with the absolute filename ”c:/folder2/ggui.xml”. In this file, the relative image source filename ”image.png” is referenced for one of the parts of class ”G:Image”. The

path of the file being parsed is ”c:/folder2”. The first file check will therefore be for the file with the absolute filename ”c:/folder2/resources/image.png”. If the file exists, it will be used. Otherwise, the parser will use the default Java framework to form the absolute filename ”c:/folder1/image.png”. If this file exists it will be used, and if it does not exist a parsing error will be reported.

The first check allows the user to better organize his/her interface definitions along with the image resources being referenced by them.

Appendix B

Implementation

B.1 Model

The two created PUID Interpreters, Android PUIDI and Swing/AWT PUIDI, actually consist of two major parts: a shared Java *front-end*, and a platform specific *back-end*. The front-end is implemented using Java, and provides an interface so that platform specific PUIDIs can be constructed by interacting with this front-end. Since both of the constructed PUIDIs are Java based, they both build upon the shared Java front-end. A similar model could be developed for different programming languages, but that's outside the scope of this thesis.

The front-end parses an input PUI Description file, and constructs an internal representation of the description. All parts and units specified in the PUID are represented by a specific class in this internal representation. These classes provide an array of methods and properties that are needed to build a proper GUI in the back-end. Most of the heavy lifting is done by the front-end, and it is mainly by implementing two Java interfaces, `AbstractBuilder` and `AbstractConnector`, that the different back-ends are created. The back-end implementer passes an instance of his/her implementation of this two interfaces along with the PUID filename to the front-end parser.

The `AbstractBuilder` interface provides one build method for each part class of PUIML, e.g. `onButton` for parts of class "G:Button". The purpose of these methods is to allow the back-end implementer to decide how parts of a certain class should be created on the target platform. By implementing the interface in different ways, different back-ends can be created. The methods all take one parameter: an instance of the Java class representing the specific part that should be built. For example, parts of class "G:Button" are internally represented by the Java class `LButton`. Hence, `onButton(LButton part)`. If a PUID has three different buttons, the method should be called three times, once for each button. The parameter should be the `LButton` instance that represents the corresponding "G:Button" part internally.

When the front-end parser has constructed the internal representation of the PUID, the build method for the top most part will be called, i.e. `onTopContainer`. By using the methods and properties provided by the parameter, the implementer can create a suitable top container for the target platform. All Java class representations of PUIML parts that can contain other parts, such as top containers, provide methods to get the Java class representation of the contained parts. In the build method of such container parts, the implementer must call the build method of the contained parts. The build methods return an object of type `Object`, e.g. `Object onButton(LButton part)`. The actual return type of the object depends on the implementation, but could be used to return a graphical component which can then be placed inside the graphical component of the container part.

The second interface used to implement custom back-ends is `AbstractConnector`. This interface contains methods associated with the PalCom universe. Its main purpose is to find and establish connections to PalCom services. By implementing the methods of this interface the back-end implementer can control how the PUIDI perform those tasks.

How data from the PalCom universe goes through the front-end and is presented via the back-end is worth noting. The front-end parses the PUID and creates the internal representation of parts and units. The Java part instances have references to the Java unit instances for which they are either provider, viewer or invoker. The front-end then uses the information in the Java unit instances along with the provided methods in the `AbstractConnector` interface to create connections to the specified PalCom services. These connections are then saved internally in the Java unit instances. When the graphical components are created in the build methods of the `AbstractBuilder`, the Java part instance provided as the parameter for the build method is used to get a reference to a Java unit instance. The graphical component can then be properly matched to the unit, and data can flow in both directions.

By implementing the two interfaces described above, a new Java back-end is created. Combining the Java front-end with a newly created back-end makes for a new Java based PUIDI that can be used on the target platform.

B.2 Java front-end

The Java PUID Interpreter front-end is made up of five Java packages: `*.builder`, `*.connector`, `*.parser`, `*.parser.parts` and `*.parser.units`, where `*` is spelled out as `se.lth.cs.xbjorn.java_fe_puidi`. The basic idea of the front-end was discussed in appendix B.1, and in this part of the appendix the classes in the packages of the front-end will be presented.

B.2.1 se.lth.cs.xbjorn.java_fe_puidi.parser.parts

This package contains classes that are used to represent PUI Markup Language parts.

LPart.java
: Class used to represent any part. Contains methods and properties that are shared by all parts.

LContainerPart.java
: Extension of `LPart`. Represents all parts that can contain other parts. Contains a `List` of `LParts`.

LTabbed.java
: Class that represent parts of class ”G:Tabbed”. Extension of `LContainerPart`.

LLayoutPart.java
Extension of `LContainerPart`. Contains methods and properties associated with the layout of a container part.

LTopContainer.java
Class that represents parts of class "G:TopContainer". Extension of `LLayoutPart`.

LArea.java
Class that represents parts of class "G:Area". Extension of `LLayoutPart`.

LFontPart.java
Extension of `LPart`. Represents all parts for which the font properties apply.

LLabel.java
Class that represents parts of class "G:Label". Extension of `LFontPart`.

LTextArea.java
Class that represents parts of class "G:TextArea". Extension of `LFontPart`.

LTextField.java
Class that represents parts of class "G:TextField". Extension of `LPart`.

LImage.java
Class that represents parts of class "G:Image". Extension of `LPart`.

LButton.java
Class that represents parts of class "G:Button". Extension of `LPart`.

B.2.2 se.lth.cs.xbjorn.java_fe_puidi.parser.units

This package contains classes that are used to represent PUI Markup Language units.

LUnit.java
Class used to represent any unit. Contains methods and properties that are shared by all units, i.a. a `List` of `LUnits`.

LDevice.java
Class that represents units of class "P:Device". Extension of `LUnit`.

LService.java
Class that represents units of class "P:Service". Extension of `LUnit`.

LCommand.java
Class that represents units of class "P:Command". Extension of `LUnit`.

LParam.java
Class that represents units of class "P:Param". Extension of `LUnit`.

B.2.3 se.lth.cs.xbjorn.java_fe_puidi.builder

This package contains a single class.

AbstractBuilder.java
This interface, along with `AbstractConnector`, is what the back-end implementer needs to implement, instantiate and provide to the Java front-end in order to create a fully functional PUIDI. Contains methods that should be called in order to create the graphical components of the PUI.

B.2.4 se.lth.cs.xbjorn.java_fe_puidi.connector

This package contains classes that the front-end uses to find and connect to PalCom services.

AbstractConnector.java
This interface, along with `AbstractBuilder`, is what the back-end implementer needs to implement, instantiate and provide to the Java front-end in order to create a fully functional PUIDI. Contains methods that the front-end uses to find and connect to PalCom services.

Invokable.java
Simple interface used to ”link” a connection established by the `AbstractConnector` to an instance of `LService`.

DefaultConnector.java
Implementation of the `AbstractConnector` interface. Can be used as a default implementation when no custom implementation is needed.

MasterDevice.java
Helper class used for implementation of `DefaultConnect`.

MasterService.java
Helper class used for implementation of `DefaultConnect`.

B.2.5 se.lth.cs.xbjorn.java_fe_puidi.parser

This is the package that ties together all packages of the Java front-end.

PUIDParser.java
This is the main class of the package. The back-end inputs a PUID filename, an `AbstractConnector` and an `AbstractBuilder`. The parser will parse the file, create an internal representation of the description and with the aid of the provided `AbstractConnector` and `AbstractBuilder` create a PUI for the user.

PropertyParser.java
Class contain methods to parse the various data types that can occur in a PUID.

Constants.java
Class that contains static constants that are used heavily in the implementation of the front-end.

Logger.java
: Class that provides methods for logging errors to a file defined by the implementer.

B.3 Android back-end

This is the most complete of the two developed back-ends, and consist of five classes. The classes and a short description of them follow below.

BuilderActivity.java
: This is the main `Activity` (Android term) of the solution. It is what starts up the app, and is also the implementation of the `AbstractBuilder` interface. Hence, this is the class that is passed to the front-end.

ConnectionService.java
: This is a `Service` (Android term, runs in the background), which is used to ensure that the communication with the PalCom entities works even though the app is running in the background.

LitePUIDParser.java
: Simple extension of `PUIDParser` that adjusts the parser to work on the Android OS.

TunnelMasterDevice.java
: Simple extension of `MasterDevice`, i.e. small change in `DefaultConnector`. Allow for the connection of configuration defined PalCom tunnels.

Tunnel.java
: Class that represents a PalCom tunnel as defined in the configuration file.

B.4 Swing/AWT back-end

The Swing/AWT back-end is made up of only two classes: `SwingBuilder.java` and `Main.java`. The later is simply a startup class that contains the `main` method, which creates a `SwingBuilder` and calls its parse method. The `SwingBuilder` is the implementation of `AbstractBuilder` that is passed to the front-end. `DefaultConnector` is used.

Appendix C

Installation

C.1 Android PUID Interpreter

C.1.1 Application

1. Power up the Android smartphone and connect it to the computer using a USB cable.

2. Install `android_puidi.apk` on the device. Instructions on how to do this will not be included here, but are widely available on the Internet.

3. Using the computer, navigate to the root directory on the phones SD card.

4. Create a new directory called ”PUIDI”. Navigate to the new directory.

5. Create a new directory called ”descriptions”.

6. Create an empty text document called ”device_config.xml”.

7. Edit `device_config.xml` using any text editor so that it contains the following text: ”<configuration></configuration>”

C.1.2 PUI Descriptions

1. Power up the Android smartphone and connect it to the computer using a USB cable.

2. Using the computer, copy/move one or more PUID files to the directory ”<sd root>/PUIDI/descriptions”, where ”<sd root>” is the path to the phones SD card on the computer. **Note:** All PUID files must have the file extension ”xml”, i.e. have a filename that ends in ”`.`xml”.

C.1.3 Tunnels

1. Power up the Android smartphone and connect it to the computer using a USB cable.
2. Using the computer, navigate to the directory ”<sd root>/PUIDI”, where ”<sd root>” is the path to the phones SD card on the computer.
3. Open `device_config.xml` using any text editor.
4. Add an XML element called ”tunnel” between the configuration’s start tag, `<configuration>`, and its end tag, `</configuration>`.
5. Add an XML attribute called ”host” to the new tunnel element. The value of this attribute should be the host name of the device hosting the PalCom tunnel server.
6. Add an XML attribute called ”port” to the new tunnel element. The value of this attribute should be the port used by the PalCom tunnel server.

Optional:

7. Add an XML attribute called ”username” to the new tunnel element. The value of this attribute should be a valid user name as defined by the PalCom tunnel server.
8. Add an XML attribute called ”password” to the new tunnel element. The value of this attribute should be the password associated with the user name as defined by the PalCom tunnel server.

```
<configuration>
  <tunnel host="192.168.0.6" port="8030" />
  <tunnel host="192.168.0.7" port="8030" username="abc" password="123" />
</configuration>
```

Listing C.1: Example of tunnel configuration file content

C.2 Swing/AWT PUID Interpreter

1. Download and install the *Java Runtime Environment*.
   `http://www.java.com/getjava/`
2. Put `swing_puidi.jar` in any directory on the computer.

Appendix D

User's manual

D.1 Android PUID Interpreter

To start using the application, first boot up your Android powered smartphone. When booted and in its idle mode, the screen should display something similar to figure D.1. Locate the application called "PUIDI" on your device and start it by pressing its icon

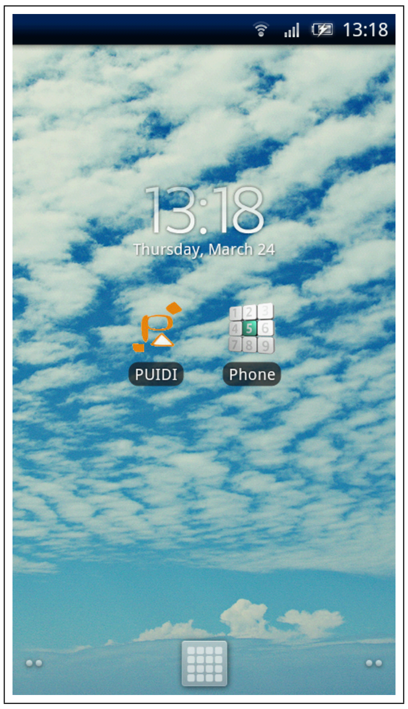


Figure D.1: Android device after startup

(figure D.2). The application will start, and the user will be presented with a list containing all installed PUI Descriptions (figure D.3). From this screen there are three possible outcomes (figure D.4):

Figure D.2: Icon for "PUIDI" app

(a) You, the user, choose no PUID by pressing the back button on the device, resulting in a blank screen. From here, there is no other option than to quit the application (or leave it running in the background to no end).

(b) You, the user, choose a PUID to generate by pressing the screen. The PUID is parsed, and the generated PUI is displayed. Naturally, the looks of this screen depends on the content of the PUID. Figure D.4(b) shows the result of generating the PUID listed in appendix E.1.1. Note that since no data has arrive yet all fields are empty.

(c) You, the user, chooses a PUID to generate by pressing the screen, but while parsing the PUID an error is encountered. Terminate the application and consult the log file to find and resolve the problem.

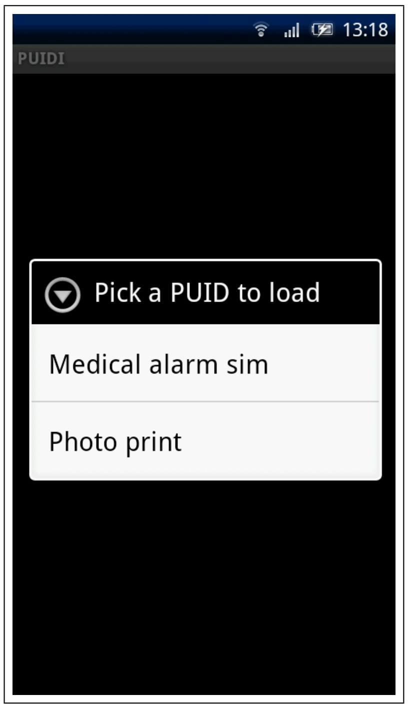
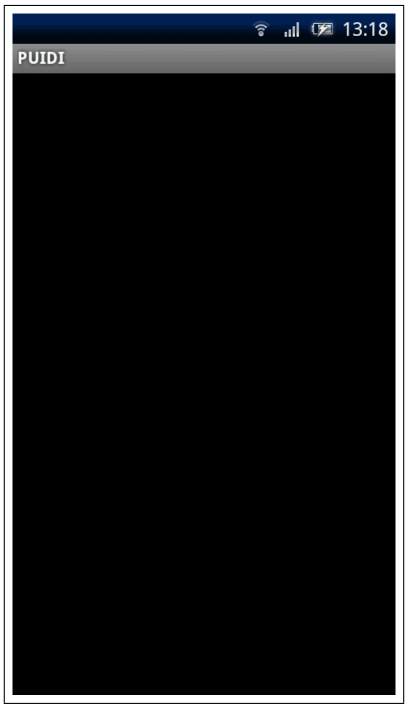
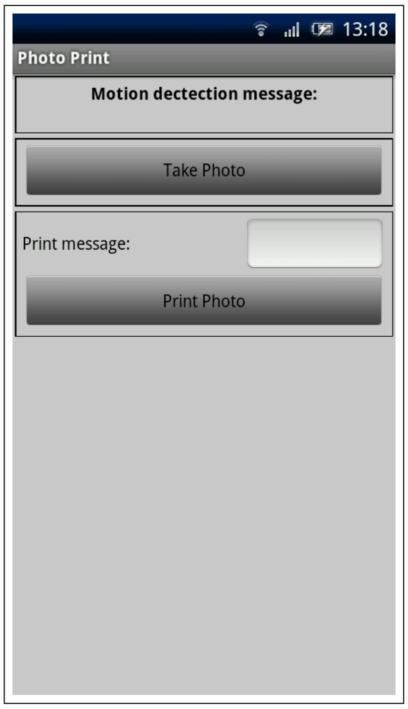


Figure D.3: PUI Description selection screen

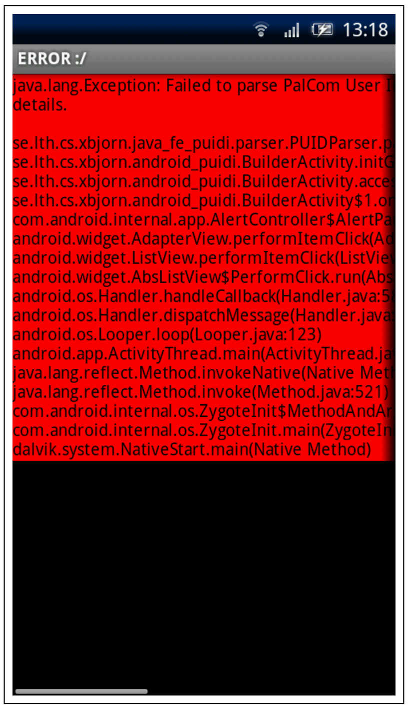
Continuing with alternative b, once the PUI has been generated, you may now interact with it at your leisure. If any of the graphical components are disabled, i.e. grayed out and non-responsive, this means that the PalCom entity linked to this component isn't available. You can at any time choose to terminate the application by pressing the back



(a) No selection (back)



(b) PUI generated



```
java.lang.Exception: Failed to parse PalCom User I
details.

se.lth.cs.xbjorn.java_fe_puidi.parser.PUIDParser.p
se.lth.cs.xbjorn.android_puidi.BuilderActivity.init(
se.lth.cs.xbjorn.android_puidi.BuilderActivity.acce
se.lth.cs.xbjorn.android_puidi.BuilderActivity$1.o
com.android.internal.app.AlertController$AlertPa
android.widget.AdapterView.performItemClick(Ad
android.widget.ListView.performItemClick(ListVie
android.widget.AbsListView$PerformClick.run(Abs
android.os.Handler.handleCallback(Handler.java:5
android.os.Handler.dispatchMessage(Handler.java
android.os.Looper.loop(Looper.java:123)
android.app.ActivityThread.main(ActivityThread.ja
java.lang.reflect.Method.invokeNative(Native Met
java.lang.reflect.Method.invoke(Method.java:521)
com.android.internal.os.ZygoteInit$MethodAndAr
com.android.internal.os.ZygoteInit.main(ZygoteIn
dalvik.system.NativeStart.main(Native Method)
```

(c) PUI Description parsing error

Figure D.4: Possible outcomes from PUI Description selection screen

button until you reach the home screen of the device. When the application is terminated, a message box will be displayed to inform how the application termination was handled (figure D.5). If all connections where closed properly the text "PUIDI shutdown OK" will be displayed, otherwise the text ">> Network shutdown failed <<" will be displayed instead.

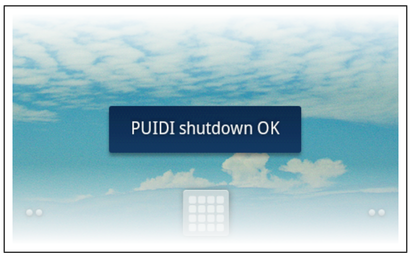
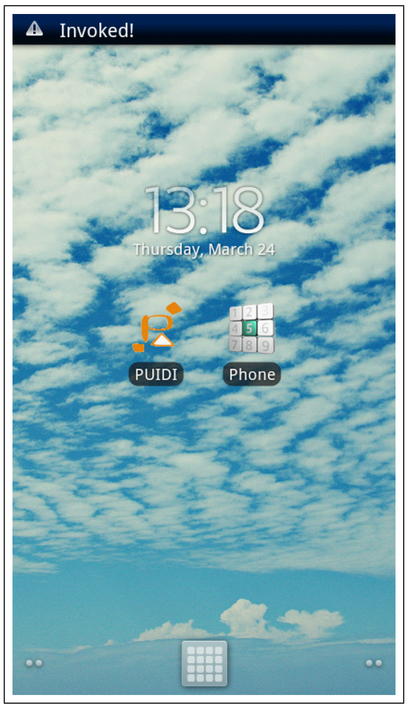
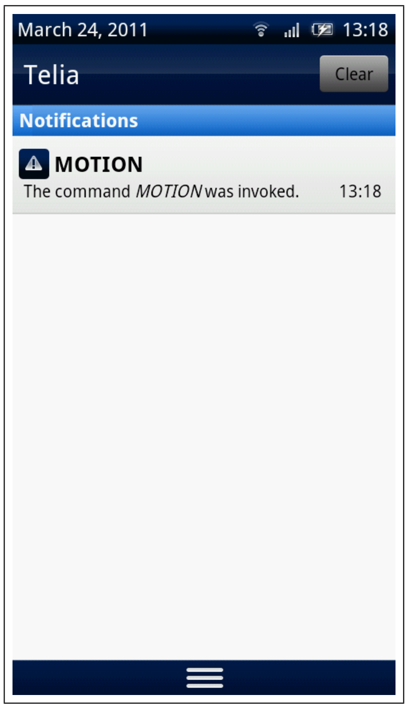


Figure D.5: "PUIDI" shutdown message

You may also choose to let the application run in the background. To do this, simply press the home button, which will return you to the home screen. When a message arrives that demands your attention (as defined by the selected PUID), the application will inform you using an Android notification message. What this might look like is shown in figure D.6.



(a) Home screen when message arrives



The command *MOTION* was invoked. 13:18

(b) Expanded notification bar

Figure D.6: Example of attention demanding message

In figure D.6(a), the critical message arrives, triggering the Android notification which can be seen in the upper left corner of the figure. By expanding the notification bar (figure D.6(b)) you can see all unattended message notification. If you click one, the PUI will appear, and the arrived message can be studied. If the application is already running in the foreground, the PUI will automatically be updated and the message can be studied directly without having to click the notification. For an example on what a PUI might look like after a message has arrived, compare figure D.7 to figure D.4(b).

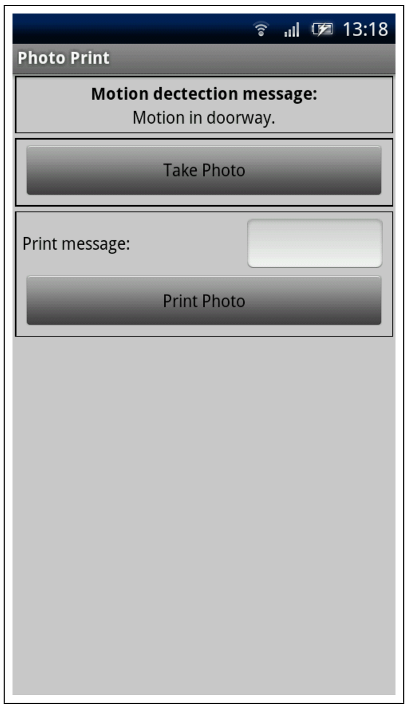


Figure D.7: PUI with message content

Troubleshoot: If you after starting the "PUIDI" application get stuck with an empty screen (figure D.4(a)), i.e. you don't get to the PUID selection screen (figure D.3), there is most likely some problem with the PalCom tunnel connectivity. Please see appendix C.1.3 on the topic of how to properly install PalCom tunnels.

D.2 Swing/AWT PUID Interpreter

To start the application, located the "swing_puidi.jar" file and run it, e.g. by double-clicking its icon. The first thing you will be presented with is a file chooser dialog window, as seen in figure D.8. Using this dialog, browse the computer's file system to locate a PUI Description file of your choice. Select a PUID to load either by double-clicking it or by highlighting it and pressing the "Open" button. After a PUID has been selected, the PUID Interpreter will attempt to parse it and create a PUI. From here, there are two possible outcomes (figure D.9):

(a) The PUID is parsed without errors, and the generated PUI is displayed. The looks of the PUI depends on the content of the PUID, and figure D.9(a) shows the result of generating the PUID listed in appendix E.2.1.

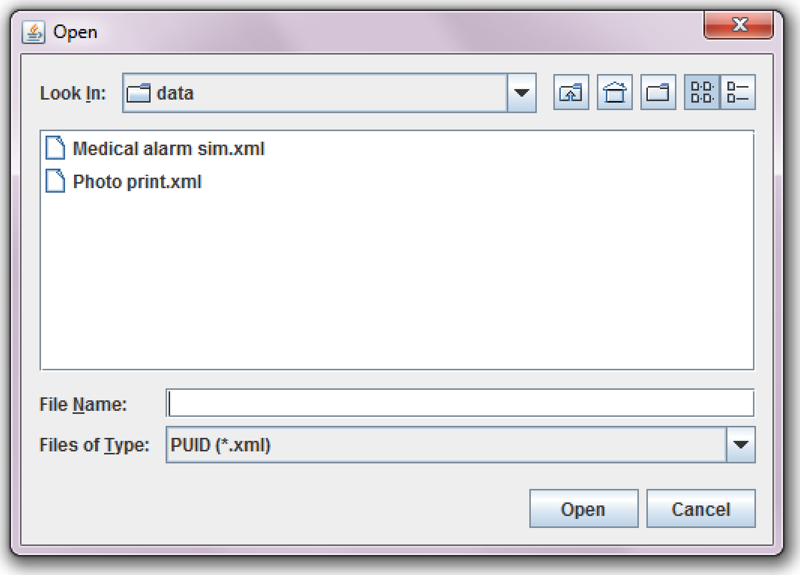
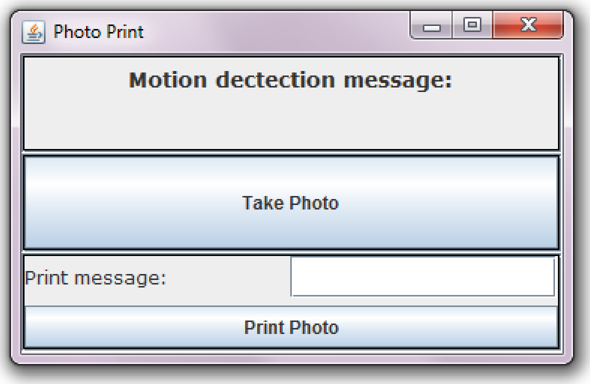


Figure D.8: PUI Description file chooser dialog

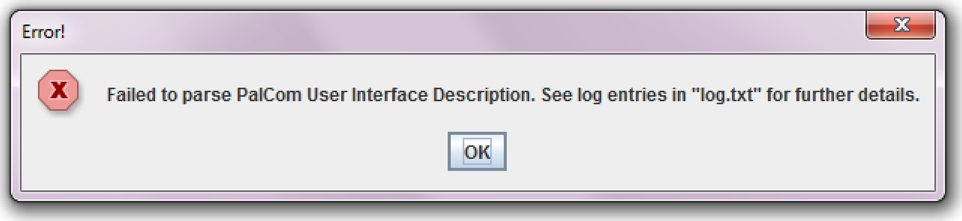
(b) While parsing the PUID an error is encountered. An error message similar to the one shown in figure D.9(b) is displayed to explain the error. Clicking the "OK" button terminates the application.

Continuing with alternative a, you may now use the generated PUI for its intended purposes. Note that if any of the graphical components are disabled, i.e. grayed out and non-responsive, this means that the PalCom entity linked to this component isn't available. For example, if the PalCom command a button acts as invoker for is unavailable, the button will not be clickable. When an incoming message arrives, the content of the message is presented in the PUI as specified by the PUID. For an example of what this might look like, see figure D.10 in contrast to figure D.9(a). Because of the limitations of the Swing/AWT PUIDI, messages that demand your attention (as defined by the PUID) will have no special effect on the PUI when arriving.

To terminate the application, simply close the window in which the PUI resides.



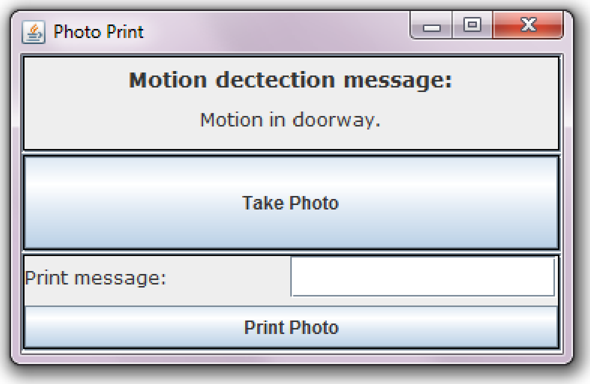
(a) PUI generated



Failed to parse PalCom User Interface Description. See log entries in "log.txt" for further details.

(b) PUI Description parsing error

Figure D.9: Possible outcomes of the file chooser dialog



Motion in doorway.

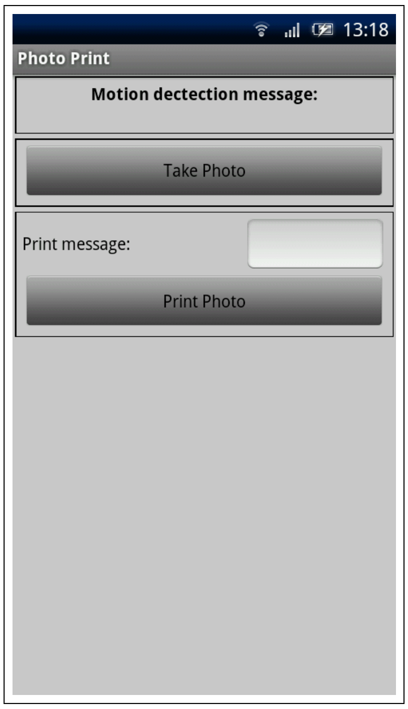
Figure D.10: PUI after message arrival

Appendix E

PUIML code samples

E.1 Android PUID Interpreter

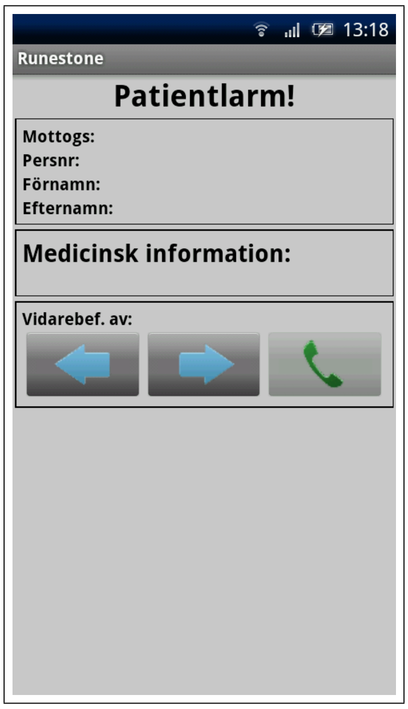
E.1.1 Photo Printer



```
<?xml version="1.0" encoding="UTF-8"?>

<puiml>
  <universe>
    <unit id="device" class="P:Device">
```

```
      <unit id="synth-assembly" class="P:Service">
        <unit id="motion" class="P:Command">
          <unit id="motion-message" class="P:Param" />
        </unit>
        <unit id="snap" class="P:Command" />
        <unit id="print" class="P:Command">
          <unit id="print-message" class="P:Param" />
        </unit>
      </unit>
    </unit>
  </universe>

  <discovery>
    <property unit-name="device" name="p:id">X:PP1001</property>

    <property unit-name="synth-assembly" name="p:instance">1</property>
    <property unit-name="synth-assembly" name="p:cdid">X:1scenarioBJ</
        property>
    <property unit-name="synth-assembly" name="p:cn">BJscenario1</property>
    <property unit-name="synth-assembly" name="p:udid">X:1scenarioBJ</
        property>
    <property unit-name="synth-assembly" name="p:un">BJscenario1</property>

    <property unit-name="motion" name="p:id">MOTION</property>
    <property unit-name="motion" name="p:direction">out</property>
    <property unit-name="motion" name="p:notifications">normal</property>

    <property unit-name="motion-message" name="p:id">MESSAGE</property>

    <property unit-name="snap" name="p:id">SNAP</property>
    <property unit-name="snap" name="p:direction">in</property>

    <property unit-name="print" name="p:id">PRINT</property>
    <property unit-name="print" name="p:direction">in</property>

    <property unit-name="print-message" name="p:id">MESSAGE</property>
  </discovery>

  <structure>
    <part id="window" class="G:TopContainer">
      <part id="scroll-area" class="G:Area">
        <part id="motion-area" class="G:Area">
          <part id="message1-label" class="G:Label" />
          <part id="message1-output" class="G:Label" />
        </part>
        <part id="snap-area" class="G:Area">
          <part id="snap-button" class="G:Button" />
        </part>
        <part id="print-area" class="G:Area">
          <part id="sub-area" class="G:Area">
            <part id="message2-label" class="G:Label" />
            <part id="message2-input" class="G:TextField" />
          </part>
          <part id="print-button" class="G:Button" />
        </part>
      </part>
    </part>
  </structure>

  <style>
    <property part-name="window" name="g:title">Photo Print</property>
    <property part-name="window" name="g:layout">linear</property>
    <property part-name="window" name="g:layout-orientation">vertical</property>

    <property part-name="scroll-area" name="g:layout">linear</property>
    <property part-name="scroll-area" name="g:layout-orientation">vertical</property>
    <property part-name="scroll-area" name="g:scrollable">true</property>

    <property part-name="motion-area" name="g:layout">linear</property>
    <property part-name="motion-area" name="g:layout-orientation">vertical</property>
    <property part-name="motion-area" name="g:border">line</property>

    <property part-name="message1-label" name="g:text">Motion dectection message:</property>
    <property part-name="message1-label" name="g:align-h">center</property>
    <property part-name="message1-label" name="g:font">Verdana</property>
    <property part-name="message1-label" name="g:font-bold">true</property>

    <property part-name="message1-output" name="g:align-h">center</property>
    <property part-name="message1-output" name="g:align-v">top</property>
    <property part-name="message1-output" name="g:font">Verdana</property>

    <property part-name="snap-area" name="g:layout">linear</property>
    <property part-name="snap-area" name="g:layout-orientation">vertical</property>
    <property part-name="snap-area" name="g:border">line</property>

    <property part-name="snap-button" name="g:text">Take Photo</property>

    <property part-name="print-area" name="g:layout">linear</property>
    <property part-name="print-area" name="g:layout-orientation">vertical</property>
    <property part-name="print-area" name="g:border">line</property>

    <property part-name="sub-area" name="g:layout">grid</property>
    <property part-name="sub-area" name="g:layout-columns">2</property>

    <property part-name="message2-label" name="g:text">Print message:</property>
    <property part-name="message2-label" name="g:font">Verdana</property>

    <property part-name="print-button" name="g:text">Print Photo</property>
  </style>

  <behavior>
    <property part-name="message1-output" name="p:viewer">motion-message</property>

    <property part-name="snap-button" name="p:invoker">snap</property>

    <property part-name="print-button" name="p:invoker">print</property>
    <property part-name="message2-input" name="p:provider">print-message</property>
  </behavior>
</puiml>
```

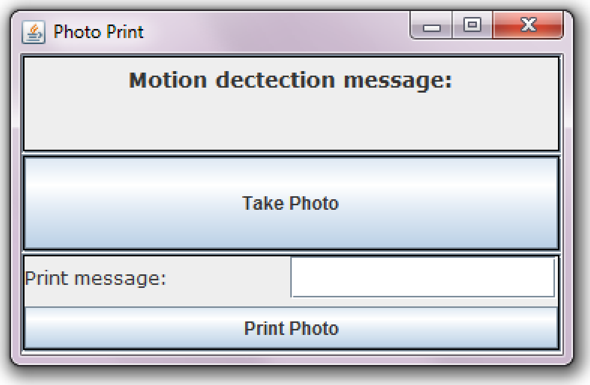
E.1.2 Alarm Receiver



```
<?xml version="1.0" encoding="UTF-8"?>

<puiml>
  <universe>
    <unit id="alarm_device" class="P:Device">
      <unit id="alarm_service" class="P:Service">
        <unit id="prev" class="P:Command" />
        <unit id="next" class="P:Command" />
        <unit id="alarm" class="P:Command" />
        <unit id="data" class="P:Command">
          <unit id="date" class="P:Param" />
          <unit id="pnbr" class="P:Param" />
          <unit id="fname" class="P:Param" />
          <unit id="lname" class="P:Param" />
          <unit id="info" class="P:Param" />
          <unit id="refn" class="P:Param" />
          <unit id="reft" class="P:Param" />
        </unit>
      </unit>
    </unit>
    <unit id="phone_device" class="P:Device">
      <unit id="phone_service" class="P:Service">
        <unit id="call" class="P:Command" />
      </unit>
    </unit>
  </universe>

  <discovery>
    <property unit-name="alarm_device" name="p:id">X:ac3000</property>

    <property unit-name="alarm_service" name="p:instance">1</property>
    <property unit-name="alarm_service" name="p:cdid">X:1tfalarmBJ</property
        >
    <property unit-name="alarm_service" name="p:cn">BJtfalarm1</property>
    <property unit-name="alarm_service" name="p:udid">X:1tfalarmBJ</property
        >
    <property unit-name="alarm_service" name="p:un">BJtfalarm1</property>

    <property unit-name="prev" name="p:id">PREV</property>
    <property unit-name="prev" name="p:direction">in</property>

    <property unit-name="next" name="p:id">NEXT</property>
    <property unit-name="next" name="p:direction">in</property>

    <property unit-name="alarm" name="p:id">ALARM</property>
    <property unit-name="alarm" name="p:direction">out</property>
    <property unit-name="alarm" name="p:notifications">normal</property>

    <property unit-name="data" name="p:id">DATA</property>
    <property unit-name="data" name="p:direction">out</property>
    <property unit-name="data" name="p:notifications">off</property>

    <property unit-name="date" name="p:id">DATE</property>
    <property unit-name="pnbr" name="p:id">P_NUMBER</property>
    <property unit-name="fname" name="p:id">F_NAME</property>
    <property unit-name="lname" name="p:id">L_NAME</property>
    <property unit-name="info" name="p:id">MEDICAL</property>
    <property unit-name="refn" name="p:id">REF_NAME</property>
    <property unit-name="reft" name="p:id">REF_PHONE</property>


    <property unit-name="phone_device" name="p:id">X:phone000</property>

    <property unit-name="phone_service" name="p:instance">1</property>
    <property unit-name="phone_service" name="p:cdid">X:1phoneXY</property>
    <property unit-name="phone_service" name="p:cn">XYphone1</property>
    <property unit-name="phone_service" name="p:udid">X:1phoneXY</property>
    <property unit-name="phone_service" name="p:un">XYphone1</property>

    <property unit-name="call" name="p:id">CALL</property>
  </discovery>

  <structure>
    <part id="frame" class="G:TopContainer">
      <part id="label1" class="G:Label"/>

      <part id="area1" class="G:Area">
        <part id="area1a" class="G:Area">
          <part id="label2a" class="G:Label"/>
          <part id="label2b" class="G:Label"/>
        </part>
        <part id="area1b" class="G:Area">
          <part id="label3a" class="G:Label"/>
          <part id="label3b" class="G:Label"/>
        </part>
        <part id="area1c" class="G:Area">
          <part id="label4a" class="G:Label"/>
          <part id="label4b" class="G:Label"/>
        </part>
        <part id="area1d" class="G:Area">
          <part id="label5a" class="G:Label"/>
          <part id="label5b" class="G:Label"/>
        </part>
      </part>

      <part id="area2" class="G:Area">
        <part id="label6" class="G:Label"/>
        <part id="tarea1" class="G:TextArea"/>
      </part>

      <part id="area3" class="G:Area">
        <part id="area3a" class="G:Area">
          <part id="label7a" class="G:Label"/>
          <part id="label7b" class="G:Label"/>
        </part>
        <part id="area3b" class="G:Area">
          <part id="button1" class="G:Button"/>
          <part id="button2" class="G:Button"/>
          <part id="button3" class="G:Button"/>
        </part>
      </part>
    </part>
  </structure>

  <style>
    <property part-name="frame" name="g:title">Runestone</property>
    <property part-name="frame" name="g:layout">linear</property>
    <property part-name="frame" name="g:layout-orientation">vertical</
        property>


    <property part-name="label1" name="g:text">Patientlarm!</property>
    <property part-name="label1" name="g:align-h">center</property>
    <property part-name="label1" name="g:font-size">25</property>
    <property part-name="label1" name="g:font-bold">true</property>


    <property part-name="area1" name="g:scrollable">true</property>
    <property part-name="area1" name="g:border">line</property>
    <property part-name="area1" name="g:layout">linear</property>
    <property part-name="area1" name="g:layout-orientation">vertical</
        property>

    <property part-name="label2a" name="g:text">Mottogs: </property>
    <property part-name="label2a" name="g:align-h">right</property>
    <property part-name="label2a" name="g:font-bold">true</property>
    <property part-name="label2b" name="g:align-h">left</property>

    <property part-name="label3a" name="g:text">Persnr: </property>
    <property part-name="label3a" name="g:align-h">right</property>
    <property part-name="label3a" name="g:font-bold">true</property>
    <property part-name="label3b" name="g:align-h">left</property>

    <property part-name="label4a" name="g:text">Förnamn: </property>
    <property part-name="label4a" name="g:align-h">right</property>
    <property part-name="label4a" name="g:font-bold">true</property>
    <property part-name="label4b" name="g:align-h">left</property>

    <property part-name="label5a" name="g:text">Efternamn: </property>
    <property part-name="label5a" name="g:align-h">right</property>
    <property part-name="label5a" name="g:font-bold">true</property>
    <property part-name="label5b" name="g:align-h">left</property>

    <property part-name="area2" name="g:scrollable">true</property>
    <property part-name="area2" name="g:layout">linear</property>
    <property part-name="area2" name="g:layout-orientation">vertical</
        property>
    <property part-name="area2" name="g:border">line</property>

    <property part-name="label6" name="g:text">Medicinsk information:</
        property>
    <property part-name="label6" name="g:font-size">20</property>
    <property part-name="label6" name="g:font-bold">true</property>


    <property part-name="area3" name="g:layout">linear</property>
    <property part-name="area3" name="g:layout-orientation">vertical</
        property>
    <property part-name="area3" name="g:border">line</property>

    <property part-name="label7a" name="g:text">Vidarebef. av: </property>
    <property part-name="label7a" name="g:align-h">right</property>
    <property part-name="label7a" name="g:font-bold">true</property>
    <property part-name="label7b" name="g:align-h">left</property>

    <property part-name="area3b" name="g:layout">grid</property>
    <property part-name="area3b" name="g:layout-columns">3</property>

    <property part-name="button1" name="g:image-src">left.png</property>
    <property part-name="button1" name="g:size">55,42</property>
    <property part-name="button2" name="g:image-src">right.png</property>
    <property part-name="button2" name="g:size">55,42</property>
    <property part-name="button3" name="g:image-src">phone.png</property>
    <property part-name="button3" name="g:size">55,42</property>
  </style>

  <behavior>
    <property part-name="label2b" name="p:viewer">date</property>
    <property part-name="label3b" name="p:viewer">pnbr</property>
    <property part-name="label4b" name="p:viewer">fname</property>
    <property part-name="label5b" name="p:viewer">lname</property>
    <property part-name="tarea1" name="p:viewer">info</property>
    <property part-name="tarea1" name="p:delimiter">;:</property>
    <property part-name="label7b" name="p:viewer">refn</property>

    <property part-name="button1" name="p:invoker">prev</property>
    <property part-name="button2" name="p:invoker">next</property>
    <property part-name="button3" name="p:invoker">call</property>
  </behavior>
  </behavior>
</puiml>
```

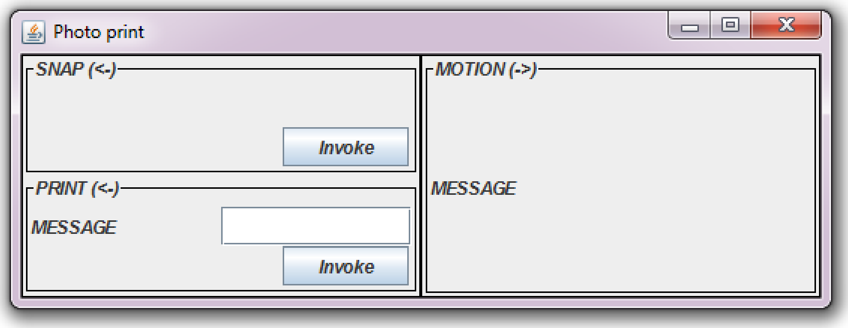
E.2 Swing/AWT PUID Interpreter

E.2.1 Photo Printer



```
<?xml version="1.0" encoding="UTF-8"?>

<puiml>
  <universe>
    <unit id="device" class="P:Device">
      <unit id="synth-assembly" class="P:Service">
        <unit id="motion" class="P:Command">
          <unit id="motion-message" class="P:Param" />
        </unit>
        <unit id="snap" class="P:Command" />
        <unit id="print" class="P:Command">
          <unit id="print-message" class="P:Param" />
        </unit>
      </unit>
    </unit>
  </universe>

  <discovery>
    <property unit-name="device" name="p:id">X:PP1001</property>

    <property unit-name="synth-assembly" name="p:instance">1</property>
    <property unit-name="synth-assembly" name="p:cdid">X:1scenarioBJ</
        property>
    <property unit-name="synth-assembly" name="p:cn">BJscenario1</property>
    <property unit-name="synth-assembly" name="p:udid">X:1scenarioBJ</
        property>
    <property unit-name="synth-assembly" name="p:un">BJscenario1</property>

    <property unit-name="motion" name="p:id">MOTION</property>
    <property unit-name="motion" name="p:direction">out</property>
    <property unit-name="motion" name="p:notifications">normal</property>

    <property unit-name="motion-message" name="p:id">MESSAGE</property>

    <property unit-name="snap" name="p:id">SNAP</property>
    <property unit-name="snap" name="p:direction">in</property>

    <property unit-name="print" name="p:id">PRINT</property>
    <property unit-name="print" name="p:direction">in</property>

    <property unit-name="print-message" name="p:id">MESSAGE</property>
  </discovery>

  <structure>
    <part id="window" class="G:TopContainer">
      <part id="motion-area" class="G:Area">
        <part id="message1-label" class="G:Label" />
        <part id="message1-output" class="G:Label" />
      </part>
      <part id="snap-area" class="G:Area">
        <part id="snap-button" class="G:Button" />
      </part>
      <part id="print-area" class="G:Area">
        <part id="sub-area" class="G:Area">
          <part id="message2-label" class="G:Label" />
          <part id="message2-input" class="G:TextField" />
        </part>
        <part id="print-button" class="G:Button" />
      </part>
    </part>
  </structure>

  <style>
    <property part-name="window" name="g:title">Photo Print</property>
    <property part-name="window" name="g:resizable">true</property>
    <property part-name="window" name="g:layout">grid</property>
    <property part-name="window" name="g:layout-columns">1</property>
    <property part-name="window" name="g:size">440,310</property>

    <property part-name="motion-area" name="g:layout">grid</property>
    <property part-name="motion-area" name="g:layout-gap">5,5</property>
    <property part-name="motion-area" name="g:layout-columns">1</property>
    <property part-name="motion-area" name="g:scrollable">true</property>
    <property part-name="motion-area" name="g:border">line</property>

    <property part-name="message1-label" name="g:text">Motion dectection
        message:</property>
    <property part-name="message1-label" name="g:align-h">center</property>
    <property part-name="message1-label" name="g:font">Verdana</property>
    <property part-name="message1-label" name="g:font-size">14</property>
    <property part-name="message1-label" name="g:font-bold">true</property>

    <property part-name="message1-output" name="g:align-h">center</property>
    <property part-name="message1-output" name="g:align-v">top</property>
    <property part-name="message1-output" name="g:font">Verdana</property>

    <property part-name="snap-area" name="g:layout">grid</property>
    <property part-name="snap-area" name="g:layout-gap">5,5</property>
    <property part-name="snap-area" name="g:layout-columns">1</property>
    <property part-name="snap-area" name="g:scrollable">true</property>
    <property part-name="snap-area" name="g:border">line</property>

    <property part-name="snap-button" name="g:text">Take Photo</property>

    <property part-name="print-area" name="g:layout">grid</property>
    <property part-name="print-area" name="g:layout-gap">5,5</property>
    <property part-name="print-area" name="g:layout-columns">1</property>
    <property part-name="print-area" name="g:scrollable">true</property>
    <property part-name="print-area" name="g:border">line</property>

    <property part-name="sub-area" name="g:layout">grid</property>
    <property part-name="sub-area" name="g:layout-gap">0,5</property>
    <property part-name="sub-area" name="g:layout-columns">2</property>

    <property part-name="message2-label" name="g:text">Print message:</property>
    <property part-name="message2-label" name="g:font">Verdana</property>

    <property part-name="print-button" name="g:text">Print Photo</property>
  </style>

  <behavior>
    <property part-name="message1-output" name="p:viewer">motion-message</property>

    <property part-name="snap-button" name="p:invoker">snap</property>

    <property part-name="print-button" name="p:invoker">print</property>
    <property part-name="message2-input" name="p:provider">print-message</property>
  </behavior>
</puiml>
```

E.2.2 Photo Printer (BrowserGUI imitation)



```
<?xml version="1.0" encoding="UTF-8"?>

<puiml>
  <universe>
    <unit id="device" class="P:Device">
      <unit id="synth-assembly" class="P:Service">
        <unit id="motion" class="P:Command">
          <unit id="motion-message" class="P:Param" />
        </unit>
        <unit id="snap" class="P:Command" />
        <unit id="print" class="P:Command">
          <unit id="print-message" class="P:Param" />
        </unit>
      </unit>
    </unit>
  </universe>

  <discovery>
    <property unit-name="device" name="p:id">X:PP1001</property>

    <property unit-name="synth-assembly" name="p:instance">1</property>
    <property unit-name="synth-assembly" name="p:cdid">X:1scenarioBJ</
        property>
    <property unit-name="synth-assembly" name="p:cn">BJscenario1</property>
    <property unit-name="synth-assembly" name="p:udid">X:1scenarioBJ</
        property>
    <property unit-name="synth-assembly" name="p:un">BJscenario1</property>

    <property unit-name="motion" name="p:id">MOTION</property>
    <property unit-name="motion" name="p:direction">out</property>
    <property unit-name="motion" name="p:notifications">normal</property>

    <property unit-name="motion-message" name="p:id">MESSAGE</property>

    <property unit-name="snap" name="p:id">SNAP</property>
    <property unit-name="snap" name="p:direction">in</property>

    <property unit-name="print" name="p:id">PRINT</property>
    <property unit-name="print" name="p:direction">in</property>

    <property unit-name="print-message" name="p:id">MESSAGE</property>
  </discovery>

  <structure>
```

```
    <part id="frame" class="G:TopContainer">
      <part id="area-1" class="G:Area">
        <part id="area-1-1" class="G:Area">
          <part id="area-1-1-1" class="G:Area" />
          <part id="area-1-1-2" class="G:Area">
            <part id="lbl-1-1-2-a" class="G:Label" />
            <part id="lbl-1-1-2-b" class="G:Label" />
            <part id="btn-1-1-2-a" class="G:Button" />
          </part>
        </part>
        <part id="area-1-2" class="G:Area">
          <part id="area-1-2-1" class="G:Area">
            <part id="lbl-1-2-1-a" class="G:Label" />
            <part id="txtf-1-2-1-a" class="G:TextField" />
          </part>
          <part id="area-1-2-2" class="G:Area">
            <part id="lbl-1-2-2-a" class="G:Label" />
            <part id="lbl-1-2-2-b" class="G:Label" />
            <part id="btn-1-2-2-a" class="G:Button" />
          </part>
        </part>
      </part>
      <part id="area-2" class="G:Area">
        <part id="area-2-1" class="G:Area">
          <part id="area-2-1-1" class="G:Area" />
          <part id="area-2-1-2" class="G:Area">
            <part id="lbl-2-1-2-a" class="G:Label" />
            <part id="lbl-2-1-2-b" class="G:Label" />
          </part>
          <part id="area-2-1-3" class="G:Area" />
        </part>
      </part>
    </part>
  </structure>

  <style>
    <property part-name="frame" name="g:title">Photo print</property>
    <property part-name="frame" name="g:resizable">true</property>
    <property part-name="frame" name="g:layout">grid</property>
    <property part-name="frame" name="g:layout-columns">2</property>
    <property part-name="frame" name="g:size">440,310</property>

    <property part-name="area-1" name="g:layout">grid</property>
    <property part-name="area-1" name="g:layout-columns">1</property>
    <property part-name="area-1" name="g:layout-gap">0,0</property>
    <property part-name="area-1" name="g:border">line</property>

    <property part-name="area-1-1" name="g:layout">grid</property>
    <property part-name="area-1-1" name="g:layout-columns">1</property>
    <property part-name="area-1-1" name="g:layout-gap">0,0</property>
    <property part-name="area-1-1" name="g:border">line</property>
    <property part-name="area-1-1" name="g:title">SNAP (&lt;-)</property>

    <property part-name="area-1-1-2" name="g:layout">grid</property>
    <property part-name="area-1-1-2" name="g:layout-columns">3</property>
    <property part-name="area-1-1-2" name="g:layout-gap">0,0</property>

    <property part-name="btn-1-1-2-a" name="g:text">Invoke</property>

    <property part-name="area-1-2" name="g:layout">grid</property>
    <property part-name="area-1-2" name="g:layout-columns">1</property>
    <property part-name="area-1-2" name="g:layout-gap">0,0</property>
    <property part-name="area-1-2" name="g:border">line</property>
    <property part-name="area-1-2" name="g:title">PRINT (&lt;-)</property>

    <property part-name="area-1-2-1" name="g:layout">grid</property>
    <property part-name="area-1-2-1" name="g:layout-columns">2</property>
    <property part-name="area-1-2-1" name="g:layout-gap">0,0</property>

    <property part-name="lbl-1-2-1-a" name="g:text">MESSAGE</property>

    <property part-name="area-1-2-2" name="g:layout">grid</property>
    <property part-name="area-1-2-2" name="g:layout-columns">3</property>
    <property part-name="area-1-2-2" name="g:layout-gap">0,0</property>

    <property part-name="btn-1-2-2-a" name="g:text">Invoke</property>

    <property part-name="area-2" name="g:layout">grid</property>
    <property part-name="area-2" name="g:layout-columns">1</property>
    <property part-name="area-2" name="g:layout-gap">0,0</property>
    <property part-name="area-2" name="g:border">line</property>

    <property part-name="area-2-1" name="g:layout">grid</property>
    <property part-name="area-2-1" name="g:layout-columns">1</property>
    <property part-name="area-2-1" name="g:layout-gap">0,0</property>
    <property part-name="area-2-1" name="g:border">line</property>
    <property part-name="area-2-1" name="g:title">MOTION (-&gt;)</property>

    <property part-name="area-2-1-2" name="g:layout">grid</property>
    <property part-name="area-2-1-2" name="g:layout-columns">2</property>
    <property part-name="area-2-1-2" name="g:layout-gap">0,0</property>

    <property part-name="lbl-2-1-2-a" name="g:text">MESSAGE</property>
  </style>

  <behavior>
    <property part-name="btn-1-1-2-a" name="p:invoker">snap</property>

    <property part-name="btn-1-2-2-a" name="p:invoker">print</property>
    <property part-name="txtf-1-2-1-a" name="p:provider">print-message</
        property>

    <property part-name="lbl-2-1-2-b" name="p:viewer">motion-message</
        property>
  </behavior>
</puiml>
```