Graphical Editor for Graphical User Interfaces for an "Internet of Things" System

Gunnar Weibull
Graphical Editor for Graphical User Interfaces for
an “Internet of Things” System

Gunnar Weibull, Lund University

Supervisors
Björn Johnsson
Boris Magnusson

Examiner
Ulf Asklund

June 27, 2015

Original illustration by Louis Rhead, 1897
Abstract

The aim for the graphical editor described in this paper is to simplify the creation of Graphical User Interfaces (GUIs) for PalCom (an “Internet of Things”-solution). Ideally, the people who are to use the GUIs will be able to create them themselves.

Most graphical editors focus on placing graphical components (buttons etc.), then assigning them a function. The graphical editor described in this paper aims to introduce the opposite workflow, i.e. identify functionality and then suggest graphical components to express this functionality in the Graphical User Interface (GUI) to be created.
# Contents

1 Problem ................................. 5  
  1.1 Introduction .......................... 5  
  1.2 Problem formulation .................. 6  
  1.3 Proposed solution .................... 6  
  1.4 Report structure ..................... 7  

2 Background ........................... 8  
  2.1 Introduction to PalCom ................. 8  
  2.2 Introduction to PML .................. 9  
    2.2.1 Code Overview ................... 10  
    2.2.2 The PML Interpreter .............. 11  
    2.2.3 PML Links ....................... 12  

3 Related work ......................... 14  
  3.1 Other graphical editors ............... 14  
  3.2 Conclusion .......................... 16  

4 Solution .............................. 16  
  4.1 Overview ............................ 16  
  4.2 Development Iterations ............... 16  
    4.2.1 Iterative development .......... 16  
    4.2.2 Iteration 1: A basic sketch .... 17  
    4.2.3 Iteration 2: Hello world! ...... 20  
    4.2.4 Iteration 3: Expanding the program 21  
    4.2.5 Iteration 4: Testing and polishing 23  
  4.3 Overview of the code ................ 27  
  4.4 Interaction with PML ................. 30  
  4.5 Suggestions for changes of PML ....... 31  
  4.6 Modifications of PML ............... 31  

5 Evaluation ............................ 31  
  5.1 Case studies ......................... 31  
    5.1.1 First testing session .......... 32  
    5.1.2 Second testing session ......... 34  
  5.2 Time efficiency ...................... 35  
  5.3 Flexibility .......................... 36  
  5.4 Usability of the editor .............. 37  
  5.5 Space efficiency ..................... 38  
  5.6 Conclusion .......................... 39  

6 Future work ......................... 39  
  6.1 Evaluation .......................... 40  
    6.1.1 Comparison to a specialized solution with the opposite work flow approach .................. 40  
    6.1.2 Testing with a non-programmer ........ 40  
  6.2 Improvements in usability .......... 40  
    6.2.1 Dynamic development tips ........ 40  
    6.2.2 Help texts ...................... 41
6.2.3 Undo/redo support .................................................. 41
6.2.4 Optimization ..................................................... 41
6.2.5 Code view .......................................................... 42
6.3 Missing functionality ................................................ 42
  6.3.1 Links between graphical components ....................... 42
  6.3.2 PML links to services. ......................................... 43
  6.3.3 More solid support for custom parts ......................... 44

7 Appendix 1: Known issues .......................................... 45

8 Appendix 2: Instructions manual .................................. 46
  8.1 Quick start ......................................................... 46
  8.2 A short introduction and some useful words ................. 47
  8.3 Overview .......................................................... 48
  8.4 Camera example .................................................. 49
  8.5 The network palette .............................................. 50
  8.6 The Command Palette ............................................ 52
  8.7 The Canvas and the Options Panel ............................ 57
  8.8 The Menu bar ..................................................... 60
  8.9 Running a PML GUI on Android ................................. 61

9 Appendix 3: Instructions given to Mia ........................... 62
1 Problem

1.1 Introduction

A hospital is an environment containing a multitude of advanced devices produced by various manufacturers. A common problem that arises is that the devices have no general way of communicating with each other and with the user. This leads to a lot of unnecessary time spent by medical personnel reading data from one machine, and inputting the data into another. Thus, significant efficiency gains are to be made if the communication to and between devices could be generalized and automatized. [1]

**PalCom**, short for *Palpable Computing*, aims to provide such a solution. Providing a general solution for communication between devices, it naturally has applications beyond the hospital example above, though health care is one of the current projects in which it is used. PalCom was originally developed as a part of an EU IST (European Union Information Society) project. It has subsequently been further developed in EASE and other VINNOVA and SSF projects.

In order for doctors, nurses and other personnel who can’t be expected to know how to code, *Graphical User Interfaces (“GUIs”)* are needed to use PalCom in a convenient way. A GUI is simply put the part of a computer program that a user sees and interacts with. Parts of a GUI might thus be buttons, text fields for input of text, or images. A program without a GUI might be controlled via the command prompt, or run with no user input at all.

Considering the wide variety of uses of PalCom, it would be hard and clunky to rely on a single software tool/GUI solution for all users of PalCom. A nurse visiting a patient on dialysis in their home, for example, might need a GUI with a button for commanding the dialysis machine to send him/her relevant data and display it on a tablet. On the other hand, another nurse might have need for a completely different GUI to do his/her job in the most efficient way.

In order to allow the creation of such specialized GUIs, a programming language was conceived. **PalCom Markup Language**, or **PML** for short, is an XML-based language (meaning information is stored in brackets. For an example of the concept check the source code of a web page.) A PML-file contains information such as what devices that a GUI should connect to, and how the Graphical Components (buttons etc.) should interact with the devices [2].

In the dialysis example, the PML file would contain information about the dialysis machine, what the command to make it send data is, and what commands and parameters it sends back once called. The PML file would also contain information about the button we want to use to make the machine send the data, and associate it with the appropriate command to the machine. Furthermore, it would describe how the data the machine sends back should be presented, e.g. as graphs, text fields and/or images.

The PML-file thus describes how the program should look and function in a compact way, and aims to separate this from the more complex parts of Pal-
Com that actually makes what the PML-file describes happen. If we want our dialysis GUI to run on an Android device for example, we can load it in an app on a tablet that will read the PML-file, do what is needed to make what the PML-file describes happen, and display the GUI for us. PML makes huge improvements with regards to how efficiently and easily a GUI for PalCom can be created, and our story might thus have ended here [2].

There is however still expected gains to be made with regards to efficiency if a more graphical solution was to be used, instead of writing the code directly in PML. A graphical solution would typically be a computer program featuring a palette of buttons, text boxes etc. that the user can drag to a canvas. When the user is finished designing her GUI, she could simply press a button to export it to a PML-file that would run on her tablet. What was proposed was thus a graphical editor for creating GUIs. Moreover, the people who can be expected to know the best how the GUIs should look and function are the users. A fully graphical solution for creating PalCom GUIs does not demand any programming knowledge, and would thus ideally be usable for the end users themselves to design their own personalized GUIs.

This is where this thesis comes into the picture.

1.2 Problem formulation

PML is a programming language for creating custom PalCom GUIs. Since a plethora of specialized PalCom GUIs are expected to be needed (depending on the use case and devices involved), increased efficiency in producing PML code is highly relevant to the overall efficiency of using PalCom. While coding PML from scratch is entirely possible, it is a tedious process which requires programming knowledge from the GUI designer. A faster way of producing PML code would thus be desirable.

Moreover, the users of the PML GUIs are expected to best know the problem and use case the PML GUI is attempted to solve. Thus, if a solution would be easy enough to use by the future GUI users themselves, efficiency could be increased since the users would likely end up with a solution that is closer to their expectations compared to if a programmer would do it for them.

The problem to be solved is thus the following: *We need a tool for creating PML files that is more efficient than coding, and that is usable to non-programmers.*

1.3 Proposed solution

The goal of this paper is to provide a graphical editor for PML, which will speed up the process of designing PalCom GUIs as well as enable non-programmers to design PalCom GUIs. The graphical approach is favored since it is typically considered more intuitive by casual users, and because it will enable users to focus more on function rather than glue code and syntax. The program will be called the *Graphical PML Editor.*
To make the usage of the Graphical PML Editor more intuitive, an unconventional approach to the work flow has been attempted. Rather than placing graphical components (buttons, text fields etc.) on a canvas and then deciding what they are to do (their functionality), the user will start by identifying what s/he wants to do (the functionality), and will then get suggestions for graphical components that can achieve that functionality. In practice, this has meant that the user will start by selecting commands and parameters, then match them with graphical components (figure 1). If for instance a device has a command to make it send data, the user may choose to have a button to prompt the device to send the data, and then choose to display the received data to the user in the form of a text field.

Figure 1: The conventional approach of GUI creation is to create graphical components and add functionality to them, e.g. by adding a listener (left). The approach tested in this paper is to identify functionality, and then suggest graphical components to express that functionality (right).

1.4 Report structure

The report begins with an introduction to PalCom and the PML language, which are the underlying technologies for the Graphical PML Editor (section 2). In the next section, a couple of other graphical editors are discussed, to put the Graphical PML Editor in a larger context (section 3). Section 4 gives an overview of the finished program, along with a summary of the development. This part also discusses how the program interacts with the PML interpreter, including suggestions for changes to PML for future versions of the Graphical PML Editor. Section 5 details the evaluation of the Graphical PML Editor, by discussing the testing of the editor and what conclusions that were made. Suggestions for future work on the Graphical PML Editor are detailed in section 6.

The report also includes appendices detailing the known issues with the current version (section 7), an instructions manual for using the Graphical PML Editor (section 8), as well as some material that was used during testing (section 9). The instructions manual was written with future users in mind, and reading the rest of the report is not needed to make use of it.
2 Background

2.1 Introduction to PalCom

PalCom is an “Internet of Things” (“IoT”) solution. The term Internet of Things refers to connecting small devices to the internet, and having them communicate with each other and the user. A common problem related to Internet of Things is that devices produced by different manufacturers can’t communicate with each other, typically because they are using different protocols. Another problem that often arises is to connect the devices if they are on different networks, or if they connect in different ways (e.g. LAN, Bluetooth etc) [3]. PalCom presents a solution to these problems.

In order to make devices accessible to each other, a PalCom network is created. Device 1 and device 3 in figure 2 want to connect but aren’t on the same network. However, both happen to be connected to device 2. With the help of a PalCom network, they can reach each other through device 2. Furthermore, the connection between devices 1 and 2 and the connection between devices 2 and 3 don’t have to use the same kind of connection (one connection can be a LAN connection and the other a Bluetooth connection, for example.) If a connection is lost, PalCom will automatically attempt to find another way for the devices to connect. [2]

![Figure 2: Example connections between three PalCom Devices.](image)

A PalCom device is a device on a PalCom network that supports the PalCom protocol. A PalCom device typically represents a piece of hardware, but it could also be a virtual device (a software-simulated device). The description for a PalCom device is ideally run on the physical device itself, but can also be run as a simulated device, e.g. by connecting the hardware to a computer on which the simulated device is run.

A PalCom device typically contains a number of PalCom Services. A PalCom Service represents an action or similar that can be performed by the device, and contains a number of commands with which the user can communicate with the service. Each of these PalCom Commands might have PalCom Parameters containing information. PalCom devices, services, commands and parameters are collectively referred to as PalCom Units.

As an example, let’s say we have a camera that we want to take a photo with, and then get the taken photo from the camera. The PalCom model for the camera can be seen in figure 3. The camera will be represented as a PalCom Device,
and it contains two services called “Photo” and “Storage”. Photo contains an in-going command (a command that is sent to the camera) called “takePhoto”. By calling takePhoto, we tell the camera to take a photograph. To get the camera to send the picture to us, we have a look at the service called Storage. By sending the in-going command “getPhoto”, we tell the camera to send the out-going command “photo”, which has a parameter containing the image. The distinction between commands and parameters will be important on the work with the graphical editor: The important difference is that a command alone can only trigger something or tell us that something has happened rather than contain information such as text or images (which is handled by the parameters of the command) [2].

![Diagram of a PalCom Device hierarchy](image)

**Figure 3: Example hierarchy of a PalCom Device.**

A couple of other notable programs in PalCom are:

**TheThing** is a Java application which acts as a device in the PalCom Universe. Services can be installed on the TheThing dynamically.

**TheAndroidThing** is a version of TheThing for the Android OS.

**The PalCom Browser** is an application for testing PalCom devices and setting up *PalCom Assemblies*. PalCom Assemblies can be used to coordinate services using a simple event-based scripting language (e.g. when *command 1* from *device 1* is received, send *command 2* to *device 2*). Assemblies can’t be directly interacted with by the graphical editor, but can be turned into *Synthesized Services* which the graphical editor can interact with in the same way as regular services if they are loaded to a TheThing.

### 2.2 Introduction to PML

PML (PalCom Markup Language) is an XML-based language that is designed specifically to describe a PalCom GUI. This means that a PML-file states information such as the PalCom units that is to be communicated with on the
network (devices, services, commands and parameters) as well as the graphical components of the GUI that associate with the commands and parameters (e.g. a button to send a command, or an image to show an image parameter).

2.2.1 Code Overview

PML contains a few basic tags as seen in listing 1. *Universe* is where the PalCom units (devices, commands etc.) used in the program will be listed, defined as *unit* tags nested in the hierarchy of the units (service tags within device tags, command tags within service tags and parameter tags within command tags, similarly to figure 3). *Discovery* lists more information about the units, such as different ids.

*Structure* is where the graphical components, windows etc. will be listed. Graphical components will be represented as *part* tags. The Structure tag furthermore defines the basic layout of the gui, in such a way that if a button tag is defined within a window tag, the button will appear in that window. *Style* contains more information about the appearance of the graphical components, such as what text is to be displayed on a button, and in what font.

*Behavior* lists the behavior properties of the graphical components, notably how they relate to the units listed in the Universe tag. A button might be associated to send a specific command upon clicking in the Behavior tag. *Logic* contains other information about the internal logic of the program, such as constants and variables. It is also worth noting that Discovery, Style and Behavior-properties can be either defined in their own main tag as seen in listing 1, or as a nested tag to the unit or part tags of Universe or Structure as seen in listing 2.

*Listing 1: The main tags of the PML code.*

```xml
1 <puiml>
2  <universe/>
3  <discovery/>
4  <structure/>
5  <style/>
6  <behavior/>
7  <logic/>
8 </puiml>
```

*Listing 2: Simple nesting example. Discovery properties can be defined within a unit tag instead of in its main tag.*

```xml
1 <universe>
2  <unit class="P:Device">
3     <discovery>
4       <property name="p:id">Camera</property>
5     </discovery>
6  </unit>
7 </universe>
```
2.2.2 The PML Interpreter

A simplified overview of the PML Interpreter can be seen in figure 4. The PML language is interpreted in two main steps. Firstly, the code is translated to an platform-independent internal model, meaning that it is the same regardless of what device the PML program is later going to be run on. This is the front-end of the PML interpreter.

The created internal model of the PML file contains classes for the different PalCom units (device/service/command/parameter) as well as the different parts (graphical components such as buttons, text fields, windows etc). The graphical editor described in this paper utilizes these classes for representing units and graphical components. Furthermore, these classes have a pre-defined ‘exportToPML’ method that makes it easy to export an internal model created in the graphical editor into a PML file. By utilizing the existing classes of the internal PML model, the work put into this thesis could be shifted from the basic requirement of outputting valid code towards making the program actually usable.

After the PML code has been interpreted into a platform-independent model in the front-end, this internal model is passed to the back-end which creates a GUI that can be displayed to the user. The back-end is platform-dependent, meaning that for each new platform a new back-end needs to be developed. By separating the back end from the other components of the interpreter, the work needed to create interpreters for new platforms is minimized. At the time of writing, a finished back-end exists for the Android platform, and development has been initiated for a back-end for Java Swing. The Android PML Interpreter
is called AndroidPUIDI, or APu for short. With APu installed on your Android device, you may load PML descriptions to your device with a PalCom Service of the APu (e.g. through the PalCom Browser) and run them with the APu app.

In addition to sending the internal model to the back end, it is also sent to a connector. The connector creates the PalCom connections between the units described in the PML code. The GUI created by the back end together with the connections created by the connector forms a connected model, which constitutes a fully functional program.

2.2.3 PML Links

An important concept of the PML language are PML Links, which describe how parts (graphical components) of the program relate to PalCom units on the network. Parts can be be compatible with any number of the following link types:

- Invoker
- Reactor
- Provider
- Viewer

Let’s take a Button as an example. A button can be clicked, and the action of clicking the button is expected to trigger some kind of action. This is expressed in PML by the button being an invoker, and having a list of invoker targets. The invoker targets are the things that should be affected by clicking the button (figure 5). Typically, the invoker targets are in-going commands (i.e. commands to a device). If we have a camera and we want a GUI with a button to take a photo, we would add the ‘take photo’-command of the camera to the button’s list of invoker targets.

![Figure 5: Example of an invoker. The “Send”-button sends the “Send”-command upon clicking.](image)

In addition to connecting parts with units, a command can also have another command as an invoker target (figure 6). This enables a light degree of scripting that is not directly related to the GUI of the program. However, the intention is that scripting that isn’t directly related to the GUI should be done in an assembly/synthesized service instead.
Figure 6: Example of a command invoking another command. When the “Photo taken”-command is received, the “Get picture”-command will be sent.

Reactors have the inverse functionality of invokers, and are thus parts that can react to an out-going command (i.e. a command from a device). An example of this is a System Notification that displays a predefined text (e.g. “System failure, system self-destruct initiated”) when a command is received (figure 7).

Figure 7: Example of a reactor. The “Whoops”-command will cause the warning message dialog to be displayed.

Providers provide data such as text or images to a parameter of a command. An example might be that we want to be able to write a text in a text field in our PML GUI, and then upload the text we wrote to a server. In this example, we would need a text field as the provider supplying the parameter with a value, and an invoker such as a ‘Send’-button to actually send the command with the parameter (figure 8).

Figure 8: Example of a provider. The text field supplies the “Text”-parameter with text. Note that the text will not be sent until we click the “Send”-button to send the “Send”-command (which contains the “Text”-parameter.)

Viewers have the inverse functionality of providers, and can thus display the values of parameters. This can for example be used to view logs from a server (figure 9. A part can have multiple link roles, and a text field might thus be able to supply information (provider) as well as display information (viewer). Some parts don’t have any link types, typically because their main purpose is to be containers for other parts (e.g. areas).
Another aspect of PML links is that a part might have different ways to link the same link type. For example, a standard OK/Cancel-dialog will have different lists of invoker targets for the two options available, e.g. send “Command 1” if the user clicks “OK” and “Command 2” if the user clicks “Cancel”. See figure 10 for an example of the link types in a messaging app.

Figure 10: An example of the link types that would’ve been used if the Android messaging app was implemented with PML.

3 Related work

As stated in previous sections, the Graphical PML Editor builds upon the PML language (e.g. by loading and saving PML code, as well as utilizing code from the front-end of the PML interpreter). When creating the graphical editor for PML, several other graphical editors were reviewed. The work flow of other editors was studied, in order to see if any editor had taken the same approach of work flow as the Graphical PML Editor (i.e. identifying functionality and then suggesting graphical components). The features of the other editors were also reviewed, in order to find features that could be integrated into the Graphical PML Editor.

3.1 Other graphical editors

The following solutions for graphical user interface editors were considered during the development of the editor described in this paper:

- WindowBuilder
- Netbeans
• Android

"WindowBuilder" is a plug-in for the Eclipse Integrated Development Environment (Eclipse IDE). Windowbuilder enables you to create your GUI using two different views, one graphical and one with the code. The user may switch between the graphical view and the code view at any point, and the tool thus combines the overview and simplicity of dragging and dropping graphical components with the power of editing the code directly. Graphical components are added by dragging them from a palette to the canvas [4]. WindowBuilder was used to design some of the main GUI parts of the Graphical PML Editor (the editor described in this paper).

A main disadvantage of WindowBuilder with regards to this project is that the functionality of a graphical component needs to be coded after the graphical component has been placed. This is contrary to one of the main goals of the editor, enabling a user with no programming knowledge to use the editor.

The Graphical PML Editor does however have some of the features of WindowBuilder. In the Graphical PML Editor you get the simplicity of placing graphical components on a canvas to auto-generate the GUI code (in addition to adding functionality with no need to code). You also have the ability to export the GUI at any time, work with the code directly in the coding software of your preference and then load the updated description back into the GUI editor. The Graphical PML Editor thus has functionality corresponding to the graphical view of WindowBuilder, and the code view can be simulated by working on the code directly in an external software. Admittedly, exporting and importing PML code is more clunky than simply switching tabs (as is the case of WindowBuilder), but the functionality is available for the advanced user.

"Netbeans" is an Integrated Development Environment similar to the Eclipse IDE. In Netbeans the user can code with the help of a handful of tools. These tools include a graphical environment for creating GUIs called Swing GUI Builder. Similarly to WindowBuilder, Netbeans’s solution aids in the creation of a GUI only to the extent of placing graphical components in the appropriate places (as well as adjusting the settings of the graphical components). In order to make a graphical component, e.g. a button, to actually do something an ActionListener needs to be added to the button and code needs to be written describing what the button should do. Swing GUI Builder thus does not enable linking graphical components to functionality without coding, and is not suitable for a user that is not familiar with programming.

Swing GUI Builder does however have a couple of noteworthy features, such as the ability to add custom components to the palette, and context-sensitive help texts[5].

Other tools for graphical GUI creation includes the graphical view of Android Studio, the official IDE for Android Application development. Android studio functions similarly to the other solutions described, with a graphical view for drag and drop and a text view for coding[6].
3.2 Conclusion

The Graphical PML Editor begins by identifying functionality and then suggests graphical components for that functionality. This sets it apart from other graphical editors, which have the workflow of first placing graphical components and then adding functionality to the graphical components.

This major difference aside, the other graphical editors considered contain features that were copied when implementing the Graphical PML Editor, such as adding graphical components by dragging them to a canvas (thus deciding the layout of the GUI to be created) and the ability of adding custom components. The other graphical editors do however require users to know how to code in order for the created GUIs to have any functionality, which is opposed to a main goal of the Graphical PML Editor. This is likely due to the other editors being more general in nature. Features that might be considered for future work on the Graphical PML Editor are the ability to switch from the standard graphical view to a code view similarly to the WindowBuilder plug-in, and the inclusion of context-sensitive help texts.

4 Solution

4.1 Overview

The functionality of the Graphical Editor consists of this basic functionality:

- The ability to import PalCom units (devices, services, commands and parameters) connected to a Palcom Network.
- The ability to load an existing PML description.
- The ability to build functional graphical user interfaces with no programming knowledge.
- The ability to export a built interface to a PML-file.

4.2 Development Iterations

4.2.1 Iterative development

The Graphical PML Editor was built through a series of iterations, which could each run as a stand-alone program. Each iteration was reviewed upon completion, in order to improve the program. The advantage of iterative development cycle such as this is that it deals with the problem that both user and developer typically have a rather vague idea of what the best way to realize the program is (both with regards to design and internal logic). The risk of not developing in iterations is thus that all of the development time may be spent on creating a full program which is then realized not to work well for the user or missing some important functionality. Mending problems that are spotted at the end of a fully developed programs might be hard, especially if the fix requires changes to the basic structure of the program. With an iterative development cycle overlooked functionality and flawed design can be spotted at an earlier stage when it’s easier to fix.
4.2.2 Iteration 1: A basic sketch

The first iteration was basically a sketch of the finished program with a focus on design rather than functionality. As can be seen in figure 11, the program layout consists of two basic parts: the palette (left part) and the canvas (right part). The palette is divided into three parts (sub-palettes).

The imagined usage scenario was the following:

1. Initially, all but the left-most palette are empty. The user starts by choosing a command in the left-most palette (helpfully labeled “1. Choose a command”), which shows the available commands and their respective services and devices.

2. Once a command is chosen, the second palette (labeled “2. Choose parameter”) updates to show the available parameters for that command. The user would then choose either the command or one of its parameters.

3. The third, right-most palette (labeled “3. Choose options”) is updated differently depending on whether a command or parameter was chosen, the direction of the command/parameter (in/out) and the type of information the parameter sends/receives (e.g. text/image):

Figure 11: The layout for the first iteration of the program. This version is a sketch for a possible layout of the finished program (and the author is aware of its ugliness).

The imagined usage scenario was the following:

1. Initially, all but the left-most palette are empty. The user starts by choosing a command in the left-most palette (helpfully labeled “1. Choose a command”), which shows the available commands and their respective services and devices.

2. Once a command is chosen, the second palette (labeled “2. Choose parameter”) updates to show the available parameters for that command. The user would then choose either the command or one of its parameters.

3. The third, right-most palette (labeled “3. Choose options”) is updated differently depending on whether a command or parameter was chosen, the direction of the command/parameter (in/out) and the type of information the parameter sends/receives (e.g. text/image):
(a) If an in-going command (i.e. a command to the device) was chosen, the options presented would include different ways of triggering (sending) the command, i.e. graphical components such as buttons. If an out-going command (i.e. a command from the device) was chosen, options for what the command could trigger would be presented, i.e. triggering other in-going commands. New graphical components would then have to be dragged to where the user want them on the canvas. In addition to suggestions for new or existing graphical components, the user is presented the option of having an out-going command trigger and in-going command.

(b) If an in-going parameter was chosen, it would show a list of options on how to supply the parameter with information (e.g. different graphical components such as text fields). Similarly to the case of commands: if the user would like to supply the information from a new graphical component, s/he could then drag the preferred component and drop it on the appropriate spot on the canvas. If the user prefers to supply the information from a graphical component already on the canvas, s/he could choose the graphical component in question from a list. Both the options for new graphical components and the list of existing graphical components presented in the palette would be filtered on the type of information the parameter needed (e.g. text/image). If on the other hand an out-going parameter was chosen, the palette would show options of where to send the information of the parameter, in a similar fashion to if it was an in-going parameter.

For an example of the graphical component suggestions, see figure 12.

Figure 12: Example suggestions for different PalCom unit/command direction combinations. Please note that the parameter suggestions would also depend on the parameter type (text/image). Painting by Claude Monet, 1871.
4. When the user has performed the above steps to add all the desired functionality of the program, s/he can press the “Generate PML code”-button to export the GUI to a PML file.

This design fulfills the two main design goals of the graphical editor, i.e. that the program doesn’t require any programming skills from the user and that graphical components have built-in functionality (the sending and receiving of commands/parameters). Contrary to other common GUI editors, it is thus impossible to have a graphical component without any functionality.

An advantage of this design is that it is based on a clear idea of what the work flow should be like for the user, and the work flow is emphasized by numbering the menus as well as organizing the menus so that the user should begin with the left-most menu and work his/her way to the right to complete a part of the functionality of the GUI [11]. The idea of this is to add to the intuitiveness of the program.

An important design decision when conceiving this design was that everything that can happen in the program should be visible or at least hinted at all times (the palettes are visible even though they are empty). There are thus no hidden menus or settings. The advantage with this approach is that the program is more predictable, and the inexperienced user will not have to search for as long to find what s/he is looking for and is less likely to miss out on important aspects of the program[7][8]. Ideally, it provides an overview of the structure of the program and what can happen. Moreover, since all but the left-most palettes are empty when starting the program, the user will be likely to begin by using that palette, thus introducing the intended work flow by the constraint of available options[8].

An obvious disadvantage of this design is that the program might look cluttered, especially if a lot of PalCom units are available on the network, or if a lot of options exist for a given command. This might be mitigated by presenting the contents differently, e.g. as expandable trees or expandable multiple-choice boxes. The palettes furthermore steal a lot of screen real estate, despite only one of them being used at a time.

A more subtle disadvantage of linking graphical components to functionality in such a direct manner might be that the program has to be updated each time functionality is added to the PML-language, or when different parameter types are added. Imagine for example that the parameter type “plain text” exists within the program, but not the parameter type “integer”. If the parameter type “integer” is added at a later stage, this editor will not know what options to provide the user in order to supply an integer to the parameter. On the other hand, if the program didn’t pre-link functionality to graphical components and the user was able to take the input from a text field and assign its value to an integer parameter, this would work if the user behaves as expected and inputs an integer into the text field. If on the other hand the user inputs any text, this might lead to unexpected results since no built-in check would exist to check that the input is an integer. Scenarios like these will likely be hard to overcome without enabling the user to code. Enabling the user to code would however
oppose the basic idea that the program shouldn’t require any programming knowledge. A compromise might be to include advanced settings enabling the user to input custom code, or to enable third-party add-ons.

4.2.3 Iteration 2: Hello world!

The focus of the second iteration was the inverse of that of the first iteration, i.e. focus on functionality rather than (visible) design considerations. The goal of the second iteration was to get the basic functionality in place in a depth-first manner, i.e. all of the core functionality of the final program should be available to a limited degree. The idea is to avoid certain functionality to be developed fully and then realized not to work well with the other parts of the program, thus wasting development time.

Specifically, the user should be able to implement a simple GUI for a basic “echo”-service. The echo service has two commands, one in-going with a parameter for the text to be echoed, and one out-going with the echoed text. A basic GUI for this service might consist of a text-field where the user can input the text to be echoed, a button to send the text, and another text-field to display the echoed text. An echo-service doesn’t provide any actually usable functionality other than indicating that the program works.

The core functionality of the editor at this point can be summarized as follows:

- The ability to import units on the network to the internal model of the editor (in order to use them in GUIs). This is needed in order to import the echo service from the network into the project.

- The ability to present options for linking commands and parameters to graphical components in a similar manner to iteration 1. We need to be able to choose to get the text from a text-field, trigger the echo service with a button and present the echoed text in a text field (for example).

- The ability to place graphical components on a canvas in order to design the GUI, i.e. click and drag the button and text-fields to where we want them in the GUI.

- The ability to export the created GUI to a PML-file, so that we can run the PML-code on a device.

- The ability to load an existing PML-description into the editor by converting it to the internal model of the editor (using the classes of the model that is created by the front-end a PML interpreter). This functionality is not strictly needed for the echo-service GUI, but will be integral to the finished graphical editor. Moreover, it is handy to be able to save and load a half-finished GUI during development.

A lot of the basic layout of the iteration 1 version of the editor remained in the second version. An options panel was added to the right of the canvas in order to display options for graphical components on the canvas (when a graphical component is placed on the canvas, default options are chosen). The three
palettes were compressed in order to save space into one “Command palette”, which lists PalCom units imported into the project and the available options for linking them to graphical components. A new palette called the “Network palette” was added to the left of the command palette in order to present items on the network that can be imported into the project.

### 4.2.4 Iteration 3: Expanding the program

With all the basic functionality in place, the code was expanded to include more graphical components and options. Furthermore, a PML-file that was written from scratch was loaded into the editor.

The addition of more graphical components made the layout of the first version inconvenient. The presentation of the graphical component suggestions were changed to a drop-down list next to the command/parameter (in the previous iterations they were presented as a bunch of draggable/clickable components added directly to the palette). Moreover, with more commands and parameters
being used in larger scale projects, the ability to expand/collapse units in the
command palette became integral. This was solved by adding a small “+/-
”-button at the top-left corner of each unit menu. For the same reasons, the
network palette (which used to view the units on the network in a long list) was
now presented as a collapsible tree-structure.

In addition to adding support for more graphical components to the editor,
some other changes had to be made. Notably, windows also need to be invoked
(in order to navigate to another window within the GUI). The windows thus
had to be introduced as items in the command palette, so that graphical com-
ponents could be added for switching between windows. It was also noted that
some graphical components can link to other graphical components (in addition
to unit/graphical component links). This functionality was postponed for now,
since it was not considered integral to creating GUIs with the editor.

Though most graphical components have a specific position in the GUI and
thus need to be placed on the canvas, some don’t (for example System notifi-
cations or pop-ups). These graphical components still need a representation in
the canvas part of the editor GUI, so that the user can click them to access their
options in the options panel. A “general options”-panel was thus created which
list these components. The “general options”-panel is displayed in the options
panel when no specific graphical component is selected.

Loading a PML-file written from scratch revealed two problems. Firstly, a cou-
pel of aspects of PML that isn’t needed for simple GUIs proved to be missing
(e.g. grid layout, tabbed windows). Secondly, the GUI loaded was a lot larger
than the example GUIs that the editor had been made in the editor so far. The
editor proved to slow down a lot while working with a large GUI.

Reloading the command palette too often was found to be a major cause for
the slow-down when working with large GUIs. The entire command palette
was recreated from the internal PML interpreter-model each time a change was
made that needed to be reflected in it, which works well for small GUIs, but took
about 4 seconds for the larger GUI. A simple fix mitigated the problem greatly.
Only the uncollapsed part of the command palette is now recreated when it is
refreshed, and when a collapsed part of the command palette is uncollapsed only
this part is now created (rather than refreshing the entire command palette).
This quick fix made working with the editor with a large file significantly nicer
(and the lag dropped from four seconds to below half a second for typical use
cases). A lot more could still be done to improve the performance, notably edit-
ing only the relevant parts of the command palette if a change is made. This
would however risk introducing new bugs since a lot more data would have to
be stored within the editor (except for the internal PML model), and has thus
been postponed for now.

22
4.2.5 Iteration 4: Testing and polishing

The Graphical PML Editor was now in a stage to be tested by other people than the developer. Testing revealed a couple of problems. The testing process is described in more detail in section 5. A large amount of bugs were fixed, and a couple of improvements were made to both the functionality and the appearance of the editor.

The ability to drag parts around on the canvas was implemented, which is useful if the user would change his/her mind about where a graphical component is placed. Before this change, if the user would like to move a part, s/he would have to delete the graphical component, recreate it in a new place, and then set all the options and PalCom unit associations from scratch. This change should thus be a big improvement in usability for the editor, especially for larger projects where a user might find new and better layouts for the GUI as development proceeds.

The tree structure for the network palette was found to be confusing in the previous version. Users would have trouble understanding that the sub-units for a unit were reachable through a folder below the unit in the tree, especially when the tree wasn’t fully uncollapsed (for example that the folder called “Commands” below the “Photo” Service in figure 14 contained the commands for the “Photo” service.) The tree structure was thus changed to have the sub-units directly below the parent units in the tree hierarchy.
Even though a manual is written for the GUI (see section 8), a quick overview of the functionality of the editor (the quick start guide) was created in order to mitigate user confusion upon starting to use the program. Furthermore, software users often do not consult a manual for figuring out how a software works[10], which emphasizes the need for quick instructions for programs with complex functionality. The quick start guide is displayed upon start-up of the editor (unless disabled) in a “welcome window”, or can be viewed at any time through the help menu. During testing (section 5), the quick start guide proved to provide enough information to create the example GUI described in the manual. The quick start guide might thus provide a good tool for people comfortable with using computers, who only need to be taught the basics of a program and tend to find the rest out on their own.

The save/load buttons were moved to a menu bar to free up space for more options. In addition to the already present options, some options were added, such as creating a new GUI, disabling the welcome window or viewing the quick start guide. Related to this, meta settings for the editor (such as the file path for the most recent file saved) were now saved upon quitting the editor.

In the previous iteration, any settings edited in the options panel had to be saved by clicking a “save”-button (figure 16). Having a “save” button in the options panel would make more sense if the options panel was presented as a pop-up window, because you would then have to make a conscious decision whether to save or not in order to proceed from editing the options. The user
would thus loose changes if s/he didn’t actively remember to save in the previous iteration of the editor. Having a “save” button for the options panel was thus deemed nonintuitive, and the button was replaced with an auto-save feature. In order for the user to be able to return to the general options without the save/cancel-buttons, two buttons were added: a small purple one with an arrow symbol in the top-left corner of the options panel and one at the bottom of the options panel with the text “Return to general options” (see figure 15).

![Image](image_url)

**Figure 16:** The save-button of the options panel (shown above) was deemed nonintuitive and scrapped for an auto-save solution in iteration 4.

When a graphical component is chosen for a unit in the command panel, a yellow label will appear for the unit indicating that the graphical component is now associated with that unit (see figure 17 for an example: a button triggers the takePhoto command.) In the previous iteration of the editor, clicking this label would remove the association between the unit and the graphical component. This is not obvious at first sight. It was deemed more obvious that clicking the label for the graphical component in the list would bring up the options for that component, similarly to clicking a graphical component in the canvas. It was also considered a better solution since showing the options is a harmless consequence as opposed to deleting something (which would more likely result in frustration if the user was just fooling around trying to work out how the program worked). In addition to changing this behavior, in order to facilitate finding the corresponding graphical component in the canvas, clicking the label would now also highlight the corresponding graphical component in the canvas. The functionality of removing the association between the graphical component and the unit was moved to a little “(x)”-symbol at the left part of
the label (figure 18).

Figure 17: Labels showing existing PML links would delete the link if clicked in iteration 3.

Figure 18: In iteration 4, the delete functionality had been moved to an X-symbol at the left part of the "existing link"-label, and clicking the rest of the label would now show options for the graphical component (in the options panel) as well as highlight the graphical component in the canvas.

A lot of confirmation dialogs (pop-ups) were added during this iteration. The confirmation dialogs were added for destructive actions in the editor, which the user might regret (e.g. deleting components or overwriting a file). If a user attempts to delete a component, they will be prompted if they are sure. The number of confirmation dialogs might be decreased once an undo/redo-feature is implemented (see section 6). Effort was put into making these dialogs “smart” whenever possible, e.g. if a user would try to save the file to a file name already present in the chosen directory, the user would not only be prompted to choose whether to overwrite the file or cancel the save, but also get an option to save the file as a similar but available file name (e.g. “test 2.xml” if the occupied file name was “test.xml”).

In the previous iteration, reference ids (which are shown in the canvas among other places) were automatically set to the class name of the graphical component with an index e.g. “LButton2”. This is obviously not an optimal solution, because the user would be unlikely to easily separate “LButton1” from “LButton2” from looking at the index. This was deemed to be a problem due to it making the editor confusing to use, unless the user put in active effort to name each component (which is just one of the many options available for a graphical component). A first solution was to make the user manually input the reference id for each new graphical component created, upon creating the graphical component (through a pop-up dialog). This was however deemed cumbersome. The second solution had the editor suggest a default reference name from the name of the PalCom unit the graphical component was for (e.g. the suggested name for a button triggering the “takePhoto” command would be “takePhoto”). This posed another problem, namely that the internals of PML requires all reference names to be unique. The third solution solved this by having the reference name consist of two parts (‘a’ and ‘b’) with the user
only being able to edit the first part (‘a’) and the second part (‘b’) being the
class name of the graphical component in parenthesis (the full reference name
would thus be “takephoto (LButton)”, where “takePhoto” corresponds to ‘a’
and “(LButton)” corresponds to ‘b’).

The basic assumption up to this point has been that a graphical component
needs an association with a PalCom unit in order to do anything meaningful.
This is a simplification, which became very apparent during testing. One of the
most important “static” graphical components are text labels. If you for exam-
ple want to supply a user name and a password on a log-in screen, you might do
this by having two text fields for the user to input that information. Without
text labels indicating what the text fields are for, it might be impossible for the
user of the GUI to know what to do (i.e. the user will just see two empty text
fields and nothing else). Not wanting to clutter the command palette further,
this functionality was added to the general options part of the options panel.
As seen in figure 19, the user can simply input text and drag the text field to
the canvas.

![Figure 19: Testing highlighted the need to add static graphical components such as
text labels.](image)

A visible change to the editor between this and the previous versions has
been the inclusion of icons. The two main reasons for adding icons were for
the editor to be easier to navigate quickly (since identifying an icon is typically
faster than reading and understanding a text label), and for the editor to appear
more welcoming to new users.

### 4.3 Overview of the code

The graphical editor consists of a couple of main classes, along with some helper
classes. Generally, any components of the program that rely on information of
the parts or units of the PML description are generated anew upon the user
updating PML units/parts relating to the component. For instance, if the user
adds a graphical component to the canvas, the entire canvas is rebuilt from
scratch. The component is generated from directly parsing the internal PML
model. The internal PML model thus represents the ‘true’ state of the GUI
created in the program, which is an attempt at keeping the program as simple
and robust as possible.
Figure 20 shows a summary of the main classes of the program and the relationship between them. The editor being the main class, contains classes for updating the GUI with different components of the GUI (the components being the canvas, the command palette and the network palette). Figures 21, 22 and 23 shows a summary of some of the main helper classes of some of the main classes, and the relations between them. Please note that none of the figures (20, 21, 22 or 23) are valid UML diagrams, and that only the relations and classes found to be of importance to the understanding of the program are presented. More classes exist in addition to the classes shown in the figures, for example the classes enabling the drag-and-drop functionality.

**Editor** (figure 20) is the main class, containing the main() method, and the setup of the program is thus handled in Editor. Editor furthermore handles tasks such as creating new GUIs, loading PML descriptions and saving (exporting) PML descriptions. In addition to this, Editor contains instances of classes for parsing a PML file (a parser, a builder and a caller) and thus acts as a nexus for parsing-related tasks such as adding, removing and getting units as well as removing and getting graphical components in the PML description. Related to this, Editor has methods for loading custom parts (custom graphical components). Calls to setup or update the command palette are relayed through the Editor class.

**GUI** is the nexus for the graphical parts of the program, and is the window that is shown to the user. During development it has been attempted to keep the functionality and the graphical aspects of the program separate (making it easier to port the program to other platforms as well as making it easier to change the layout of the program). This has been successful to a certain degree; however the palettes in the program are generated in a Swing format which unfortunately deviates from this design goal. It was decided that the amount of work to keep strictly to the design principle of separating functionality from appearance would be significant, especially considering that there are no current plans to port the editor to other platforms.
platforms. Updating the palettes (the network palette, the options palette and the command palette) as well as the canvas is handled by other classes, which update the GUI by calling the appropriate update-methods (figure 20). The GUI thus mainly acts as a layout class, which makes it easier to change the layout of the main parts of the editor GUI.

CommandPalette (figure 21) generates the command palette from the internal PML model. For each of the commands and parameters imported into the project from the network palette (figure 22), suggestions are generated for appropriate graphical components (existing or new) and shown in a drop-down menu. In the case of an in-going command, suggestions would include invoker parts such as buttons (new and existing). If a user chooses a new graphical component, a PaletteDraggable is shown which the user can drag to the canvas (graphical components with no specific place in the GUI layout are represented by PaletteNonDraggables which are added by clicking them in the list). Furthermore, any existing graphical components that are already related to the command/parameter (e.g. a button that already triggers a command) will be shown as clickable yellow labels (PaletteClickables).

EditorBuilder (figure 23) produces the finished canvas. The build-method of the EditorBuilder parses the different parts of the PML description model, and produces clickable labels that shows options for the part when clicked.
The options are produced by the class \texttt{GCOptions}. In the case of container parts (windows, areas etc), so called \texttt{EmptySlots} are produced to surround the parts (buttons etc) in the container part (figure 24). \texttt{EmptySlots} act as receivers for parts dragged to the canvas from the command palette. When the user wants some items with a different layout within the current layout (i.e. some parts arranged horizontally within a vertical layout), \texttt{EmptySlots} enable new container parts to be produced dynamically. It is worth noting that the canvas is generated from the internal PML model each time the canvas is updated, i.e. changes to the canvas are written directly to the PML model. This decision was made in order to keep the program as simple and robust as possible, with minimal data stored in two places at once (i.e. both in the PML model and the editor part of the program).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure23.png}
\caption{Summary of the helper classes of the EditorBuilder class.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure24.png}
\caption{A window containing a button, surrounded by four \texttt{EmptySlots}. The window has a horizontal layout, and placing a graphical component in the \texttt{EmptySlots} above or below the button will automatically create a new container with a vertical layout in the window (placing a graphical component in the \texttt{EmptySlots} to the left or right of the button doesn’t require a change of layout).}
\end{figure}

\section*{4.4 Interaction with PML}

The editor uses the internal front-end model of the PML-interpreter as the ‘true state’ of the GUI to be created, and thus minimal information is stored locally in the editor classes. The implementation of the PML front-end used is called \texttt{JavaFrontEndPUIDI}. The internal model is created at the start of the program by either loading an existing PML-file into a PML-parser or by creating a template PML file (consisting of a main window which is loaded upon the start of the PML application), and then letting the parser parse the description.

When a new unit or graphical component is created by the user, it is created in the form of an instance of the corresponding class of \texttt{JavaFrontEndPUIDI}. It is then added to its parent unit/graphical component. When a graphical component is changed, the changes are made to the \texttt{JavaFrontEndPUIDI}-class of the internal PML model.
Viewing the PML GUI to be created in the editor is achieved by a custom implementation of the AbstractBuilder-class of JavaFrontEndPUIDI.

XML-exporting for the units and graphical components of the internal PML-model, which was essential for the function of the editor, was already present in the classes of JavaFrontEndPUIDI.

4.5 Suggestions for changes of PML

More parameter types. As of the time of writing, only two parameter types are supported by PML: text and images. Other basic types such as integers and booleans will likely be needed in applications. Another idea that might prove useful are parameters that might have to be one of a determined set of values (such as italic/bold/underlined if the application is in text formatting). If custom parts were to be updated, they might be able to specify custom parameter types that can be matched with parameters on the network by the editor (e.g. a Java JAR-file parameter type).

Interfaces for graphical component PML links. During the implementation of setting links between units and graphical components, a lot of calls to get or set a specific link type had to be made. Due to the hash map of link targets being stored locally for each graphical component, the graphical components had to be separated in an extensive if/else-clause in these cases. If the graphical components would implement interfaces for the link types, this code could be shorter and expressed more clearly.

4.6 Modifications of PML

It has been attempted to make as few changes as possible to PML during the project, unless needed. No changes have been made to the PML language itself, but some changes have been made to the classes of JavaFrontEndPUIDI (especially to the graphical component and PalCom unit classes). The main changes made have been to add getters and setters to relevant fields if not present. JavaFrontEndPUIDI has for the most parts covered the needs of the editor in its current state. Building of the GUI (in order to present it to the user on the canvas) was made with a custom implementation of a builder-interface, thus not interfering with the PML interpreter internals.

5 Evaluation

Considering the two main goals of the editor was to create a tool to speed up the creation of PML GUIs as well as making it more available to non-coders, the most important aspects for evaluation are time efficiency and usability.

5.1 Case studies

In order to evaluate these factors, a couple of people with different backgrounds were asked to create a some simple sample GUIs. The people who were asked to evaluate were the following:
**Björn Johnsson:** Creator of the PML language and thus very proficient with the underlying structure of PML GUIs. However, due to a paternal leave, he hadn’t tried out the Graphical PML Editor before, and had mostly been involved in discussions concerning PML rather than the design considerations of the editor.

**Mia Måansson:** Fellow thesis author at the PalCom project, however with no experience of PML or creating Graphical User Interfaces.

**Me:** The author of the Graphical PML Editor.

Testing was done in two main parts. The goal of session one was to find out what worked well in the program, and what needed to be improved. The goal of session two was to test the program with regards to the important evaluation criteria (time efficiency and usability). In session two, some aspects of the program that needed further polishing were found as well.

### 5.1.1 First testing session

The first testing session took place at the end of iteration 3 when all of the basic functionality of the editor was implemented (see section 4.2.4), and thus provided feedback on what functioned well in the editor, functionality that might have been overlooked, as well as suggestions for improvements of the program. This testing session led to the changes and additions made during iteration 4 (section 4.2.5).

There are many different ways of conducting the evaluation. Considering the manual for the Graphical PML Editor is basically a step-by-step guide of how to create a simple example GUI, an idea was to first have the user try to finish the camera example GUI described in the manual by reading the manual, then try to make a different GUI utilizing what was learned. On the other hand, considering the program has a quick start guide, it would also be interesting to see whether the user could grasp enough of how the editor works to build a simple GUI by only reading the quick start guide (figure 25).
In order to decide on the best way to evaluate the editor, a quick test was made with Mia Månsson (who is also writing her Master’s thesis about PalCom, albeit is not involved in PML or the Graphical PML Editor). The goal of this test was to decide whether the quick start guide was sufficient for creating a simple GUI. She was first shown a list of commands relevant to the camera example in the manual, including a description on what each command did. She was then shown the quick start guide of the editor, as well as a sketch of what the finished GUI should look like. These instructions are presented in section 9.

Initially, she quickly grasped how to begin creating the camera GUI. After a quick confusion with regards to the indentation of the tree structure of the network palette (which was later solved), she imported the relevant commands, created a button for the takePhoto-command, and edited the button text (and saved the changes). She then made the photoTaken-command trigger the getPicture-command. After this, she seemed to be searching for a way to display the image. The crux was that she hadn’t imported the parameter of the photo-command, and thus couldn’t find a way to get an image to be displayed in the drop-down
list of the photo-command. She then seemed to search randomly for a while before she got confused and asked for help. After being told that she needed a parameter she finished the GUI without further issues. At one point she accidentally created a new window, but correctly identified what had happened, and managed to delete the new window when prompted. She needed a total of eight minutes to finish the GUI (from never having used the editor before).

As a result of this, it was deemed to be feasible for Björn to create a camera GUI without the manual. A slight change was made to the instructions, to emphasize the existence of an image-parameter (since parameters aren’t imported upon import of commands, Mia’s confusion was deemed to be partly due to the focus on commands in the instructions).

Coincidentally, Björn also took eight minutes to complete the camera example in the editor. Multiple suggestions for improvements were made, which are described in detail in the section 4.2.5. In conclusion, the first testing proved that the Graphical PML Editor achieved being easy to get started with even though the test subjects hadn’t used it before, with only a brief overlook of the functionality provided in terms of instructions.

5.1.2 Second testing session

The goal of the second testing session was to evaluate the program with regards to the main evaluation criteria (time efficiency and usability).

In the licentiate thesis by Björn Johnsson [2], which describes the creation of the PML language, evaluation was done by coding the GUI of a simple log-in screen example. The log-in GUI contained the following functionality:

- A login screen with two text fields for user name and password entries, accompanied by explanatory text labels, and a “Sign in”-button to send the command. The password text field displayed dots instead of letters upon input.
- A message dialog was displayed if the log-in was unsuccessful, letting the user know that the log-in had failed.
- If log-in was successful, a new window was opened. The new window contained a drop-down menu of patients and a text field displaying the heart rate of the currently selected patient. A refresh button updated the value.

Several evaluations were then done of the efficiency of PML and PalCom. Comparisons were made between coding with no PalCom-related software, with PalCom for handling network-related code but with custom solutions for functionality and graphics, and with a combination of PalCom and PML. By using the same example in the evaluation of the Graphical PML Editor, the results could be put into a larger context.

Björn managed to finish the simple log-in GUI in 40 minutes, while discussing the current state of the program and some polishing that needed to be done (which was later fixed). He also stated that he would be able to create the GUI
considerably faster if he were to do it again. Considering his past experience with the Graphical PML Editor was limited to finishing the camera example GUI described in the manual (section 8), this could be considered a significant improvement.

I developed the simple log-in GUI in 9 minutes.

5.2 Time efficiency

The time efficiency evaluation made in Björn’s paper on PML was made by comparing developers creating a simple Log-in App in different ways. Times were measured for three different solutions:

1. An ad hoc network protocol and a custom Android GUI.
2. PalCom and a custom Android GUI.
3. PalCom and manually coded PML.

The development time thus includes both network aspects, functionality and the GUI. Considering the solutions of the evaluation were made by experienced developers, the relevant time to compare for the Graphical PML Editor is my time of 9 minutes for finishing the Log-in example. The results are presented in table 1 and as a bar graph in figure 26. Please note that all but the data for the Graphical PML Editor are taken from Björn’s paper[2], and that the times presented are for the client-side of the solution (not the server side).
<table>
<thead>
<tr>
<th>Solution (Network/GUI)</th>
<th>Time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ad hoc protocol/Custom Android GUI</td>
<td>440</td>
</tr>
<tr>
<td>PalCom/Custom Android GUI</td>
<td>337</td>
</tr>
<tr>
<td>PalCom/Manually coded PML</td>
<td>44</td>
</tr>
<tr>
<td>PalCom/The Graphical PML Editor</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 1: Times for developing a Login example using different methods.

Developing the Login GUI using the fully custom solution (ad hoc network protocol and custom Android GUI) took 440 minutes, which is about 50 times longer than the time to develop the GUI using the Graphical PML Editor. The Login GUI took 44 minutes to develop by manual PML coding, which is 5 times slower compared to using the Graphical PML Editor. In conclusion, creating a GUI in the Graphical PML Editor is significantly faster than coding it manually in PML. In the larger context of comparing PalCom and the Graphical PML Editor to a fully custom solution, the time savings can be considered vast.

5.3 Flexibility

In trying to make the Graphical PML Editor easy to use and stream-line the work-flow, some compromises had to be done to the flexibility. When coding, a
great flexibility is achieved by being able to cut and paste as well as copy and paste code. Some of the flexibility of cutting and pasting code was achieved by implementing a drag-and-drop feature for moving already created graphical components during iteration 4 of development (section 4.2.5). Copying graphical components similarly to copying code is not possible in the editor at this point. Future testing is required to decide whether users find this to be an important feature.

One suggestion for a future improvement that might help mitigate the decreased flexibility of not being able to code is to include a code view to the editor (i.e. allowing the user to code within the editor). This is further detailed in section 6. It is however worth noting that there is a work-around to this, considering that the Graphical PML Editor saves and loads PML files (i.e. saving in the Graphical PML Editor, editing the code manually, and then loading the PML file back into the editor.

Another feature present when coding PML is the ability to undo/redo actions. This not possible in the current version of the Graphical PML Editor, but it is a high-priority feature for future versions. It is expected to take a considerable amount of time to implement, and had to be left out of this version of the Graphical PML Editor to keep the scope of the project reasonable.

In addition to this, the editor is also missing some of the features of the PML language, notably the ability to create associations directly between two graphical components (e.g. a button triggers a Yes/No-dialog), as well as having services trigger graphical components. These omissions are described in detail in section 6.

5.4 Usability of the editor

Both Mia and Björn finished the example camera GUI in 8 minutes after never having used the Graphical PML Editor before. It is worth noting that before the time of testing, they’d never used the editor before nor read the manual (and only had access to the quick start guide shown in figure 27). Moreover, the timer was started at the time they started looking at the Quick Start guide. The time 8 minutes thus represents the time from never having used the editor before to creating a functional GUI. This speaks for the ease of learning to use the program fast. The lack of need for consulting the manual (and finishing the GUIs with only the quick start guide as a help) indicates that learning how to use the Graphical PML Editor is easy.
Figure 27: The quick start guide proved to be all the instructions needed to start creating GUIs in the Graphical PML Editor.

Furthermore, when attempting the Log-in example GUI, Björn finished it in 40 minutes (although this was done while discussing the program.) This can be compared to when he created the same GUI by coding in PML, a language he wrote himself, which took 44 minutes[2]. Although 40 minutes is significantly longer than the 9 minutes it took me (an experienced user), it is still a good result that Björn (a novice of the Graphical PML Editor) can create a GUI in the editor faster than he can code it in PML (being an experienced PML coder).

It should also be noted that the fully graphical nature of the Graphical PML Editor constitutes a quantum leap with regards to who the editor is usable for, since it can now also be used by non-coders (“code muggles”).

5.5 Space efficiency

Due to the Graphical PML Editor outputting PML code, which typically has one line of code per part or property, not much is to be said about the space efficiency of the code output, compared to coding manually in PML. The editor might output more explicit standard values for properties (the standard value would otherwise be selected automatically if the property is omitted in the PML code.) If this would ever be a problem, it should be relatively easy to fix.
Due to this evaluation primarily dealing with coding PML manually compared to coding in PML, the reader is suggested to read Björn’s paper on PML[2] for more information on the space efficiency of PML compared to other GUI solutions.

5.6 Conclusion

The Graphical PML Editor satisfies the main goals of the project, i.e.:

Efficiency. It is faster to use compared to coding manually in PML. It is faster by a factor of about 5 when comparing an experienced PML coder to an experienced user of the Graphical PML Editor. Furthermore, even for a seasoned PML coder (and editor novice) the Graphical PML Editor proved to be a faster tool for creating PML GUIs. In a larger context, the Graphical PML Editor proved to be faster than a fully custom solution by a factor of about 50.

Usability. The editor is easy to get started with and use. New users were able to create a sample GUI in a reasonable amount of time by only consulting a quick start guide.

No coding skills required. The editor does not require the user to be able to know how to code. Further testing would be desirable with non-programmers, as detailed in section 6.1.2.

When comparing using this version of the editor to manually coding in PML, there are some drawbacks with regards to flexibility (a main one being able to copy and paste). Considering that the editor saves and loads PML code, it is possible to do such tasks by manually coding in PML when necessary (then reloading the GUI into the editor). Furthermore, some functionality of PML is yet to be integrated in the editor, as detailed in section 6.3.

A goal of the Graphical PML Editor was to try a different approach to work flow, namely to identify functionality and give suggestions for graphical components that can express that functionality as opposed to creating graphical components and then adding functionality to them. Considering that the editor satisfied the goals of the project utilizing this work-flow, the approach proved to be successful. In order to evaluate this approach more thoroughly compared to the conventional approach, some additional work would be needed. This is detailed in section 6.1.1.

6 Future work

The Graphical PML Editor version described in this paper is not intended to be the final version. In this section, a couple of notable omissions as well as suggestions for future additions and improvements will be presented. Some suggestions for further evaluation of the project are also presented. As of the time of writing, it has been confirmed that the development of the Graphical PML Editor will continue for (at least) half a year (by the author).
6.1 Evaluation

6.1.1 Comparison to a specialized solution with the opposite work flow approach

Comparing the approach to work flow attempted in the Graphical PML Editor to the work flow of other editors (i.e. identifying functionality and suggesting graphical components vs. creating graphical components and adding functionality) is tricky. As seen in figure 26, creating a custom Android GUI (which is done with the conventional approach) is a lot slower compared to creating the same GUI in the Graphical PML Editor. The comparison is not entirely fair however, considering that the Graphical PML Editor is designed for a much more specific purpose compared to Android Studio or other graphical editors for Android. A fair comparison would include the creation of a tool for creating PML GUI:s that relied on the conventional GUI creation approach. This was unfortunately outside the scope of this project.

6.1.2 Testing with a non-programmer

One of the main goals of the Graphical PML Editor is to provide a tool for PML GUI creation that is usable by the GUI users themselves. The Graphical PML Editor satisfies this by not requiring the GUI creator to know how to code.

Unfortunately, there was not enough time to test the program on a non-programmer. Testing the Graphical PML Editor on a non-programmer is likely to provide valuable insight regarding the usability of the editor. Even though the user is not required to know how to code, there is a risk that the editor might have assumed the user to think like a coder. Testing might be done with regards to criteria such as:

- If the user understands and follows the intended work flow, without any previous experience in Internet of Things, PalCom or PML.
- If the user creates a sample GUI that works as intended.
- How long time it takes for the user to make a sample GUI (and specifically, if the user will be able to finish the creation of a sample GUI before giving up).

6.2 Improvements in usability

6.2.1 Dynamic development tips

Dynamic development tips for the user with regards to the functionality of the created GUI was considered during the work on this program. A simple example might be to mark potential mistakes that the user has made. As an example, let’s say that the user has supplied a parameter with information from a text field, but no component of the GUI is used to send the command containing the parameter to the device. Since supplying a parameter with information and never sending it has no purpose, it is likely that the user has simply forgotten to make something send the command or haven’t understood that the command doesn’t send itself. A solution might be to color-mark such commands. Another instance where color-coded labels might be useful are graphical components with
no links to units (which can exist if the user creates a graphical component for a unit, and then removes the link).

Another idea for increasing the speed of the work flow might be to include a text label (or similar) somewhere in the GUI, which would dynamically feature development tips (e.g. suggest keyboard shortcuts for things the user just did manually). Thought would have to be put into this in order to avoid this feature meeting the same fate as the much-hated Clippy the Paperclip of earlier versions Microsoft Office [9] (figure 28).

\begin{figure}[h]
\centering
\includegraphics[width=0.2\textwidth]{clippy.png}
\caption{“Clippy the Paperclip” was a controversial helper in some previous versions of Microsoft Office.}
\end{figure}

6.2.2 Help texts

Buttons for displaying a short help text could be placed in appropriate places in the editor GUI. Help buttons are already present to a very limited degree in the editor at this point, namely for the PalCom units in the command palette. The text for these buttons could also do good from an overhaul. In case the GUI becomes too cluttered by numerous help buttons, an alternative might be to have a big help-button which can be pressed to display the smaller buttons (which would otherwise be hidden), or perhaps display help texts directly adjacent to their respective GUI component.

6.2.3 Undo/redo support

Since the editor makes all of it’s changes directly to the internal PML-model, an undo/redo-system might constitute a significant amount of work. Never the less, users of modern programs are so used to the undo/redo-functionality that the author of this paper considers it to be worth the extra effort.

6.2.4 Optimization

Even though some work has been done to improve the performance of the editor, more work is probably needed for it to remain zippy in some use cases. Recreating palettes etc. from scratch when they are updated is likely to be a reason if the editor appears to get sloppy performance with large PML files.
6.2.5 Code view

For advanced users, it might be more efficient to allow them to occasionally edit the code directly. A case where this might come in handy is if the user intends the GUI to include multiple copies of a certain graphical component, e.g. if the GUI contains a couple of windows that should all have a button to return back to the main window. Rather than being forced to create all these identical buttons individually, the advanced user might prefer to copy and paste the code instead. A code view might thus come in handy. At the moment, this can be achieved by importing/exporting files (considering the Graphical PML Editor loads and saves valid PML code). An integrated code view in the program might however be considered to make this process more stream-lined.

6.3 Missing functionality

6.3.1 Links between graphical components

Links between graphical components are not implemented in this version of the editor. Similarly to how a unit can link to a graphical component, e.g. a command is triggered to be sent by a button, a graphical component can link to another graphical component.

Let’s say for instance that we want to have a button to close the program, and display an OK/Cancel pop-up message upon pressing the close-button to check if the user wants to save first as shown in figure 29. To achieve this, we would have to have the button trigger the pop-up, and then have the pop-up trigger the sending of the save-command (upon selecting “Yes” in the pop-up).

This scenario is not doable in the current state of the editor due to the “graphical component-to-graphical component”-link (the button triggers a pop-up). The main reason for this omission is the priority of linking graphical components to units during the development of the editor (section 1.3). Furthermore, considering that this feature is not as widely used as unit/graphical component links, it was considered to be outside of the scope of this thesis.
Figure 29: This scenario is unfortunately not possible at the moment in the editor, due to the link between the close button and the pop-up dialog (which constitutes a PML link between two graphical components.)

A natural way to present the added functionality of linking graphical components to each other might be to present graphical components in the command panel, where the unit/graphical component links are presented. This would be intuitive considering it’s the same kind of functionality that would be added. A potential drawback of this solution is that the command panel might end up being extensive, and although the settings for graphical components can be collapsed (thus partly mitigating the problem), there is a risk of the user getting confused. A more serious concern for this solution is that the core design decision of linking units to parts might get muddled, and the user might lose the sense of development flow that has been attempted. A better way might thus be to put the linking options for graphical components in the graphical component options panels (where it could be found by more advanced users, without confusing casual users trying to get the hang of the main functionality of the program).

6.3.2 PML links to services.

In addition to PalCom commands and parameters being associated with graphical components, PalCom services also have potential to connect with graphical components. Services might for example signal when they become available. Due to how the code for PalCom Services is structured, actually implementing options for services in the command palette poses similar difficulties to implementing associations directly between different graphical components (as discussed above), and has thus been postponed to a future version of the Graphical PML Editor.
6.3.3 More solid support for custom parts

In this version of the editor, custom parts (graphical components not included in the editor) can be loaded at run-time. A more elaborate solution for setting the options for custom parts would however be desirable.

Figure 30: You have reached the end of the main part of the report. Thanks for reading!
7 Appendix 1: Known issues

In addition to the missing desired functionality in section 6, there are a few minor bugs known to be present in the program. Due to the graphical nature of the program, testing has been performed on new features as they were added (with the addition of the test cases described in section 5.1). The program should thus be more thoroughly tested before a release. Minor known bugs include:

**Mouse-over at times not registering (appearance, some computers).**

On some computers, setting the editor window to full screen (clicking the ‘maximize’-button at the top of the window) makes the canvas part of the editor loose mouse-over sensing. Normally, as a user hovers graphical components in the canvas, they turn purple and a small “crossed arrows”-symbol appears next to them. The purple color indicates that they can be clicked, and the small “crossed arrows”-symbol enables them to be dragged to another spot in the canvas. Clicking the graphical components will have the same effect as usual, and the “crossed arrows”-symbol will still appear when a graphical component is clicked. This means the bug reduces appearance rather than disables functionality. Effort has been put into solving this bug, but the attempts so far have been proven futile. Due to this issue not being present on some computers, it is unknown if the Graphical PML Editor is to blame.

**Superfluous layout parts (appearance).** If the user has a window with a couple of graphical components in a vertical layout, but wants to put a new graphical component to the left or right of one of these components (which is not possible in a vertical layout), a new area part containing the two graphical components will be created with a horizontal layout. The problem arises if one of these graphical components are deleted or removed. The new horizontal-layout area will still be there, despite having lost its purpose. For an example, see figure 31. Since the superfluous area doesn’t do anything, the bug affects appearance rather than function.

![Figure 31: When the user attempts to add a new button to the left of the “up”-button despite the window having a vertical layout, a new area with a horizontal layout is created. If the user chooses to move the new “you”-button out of the new area, the area won’t disappear despite having lost its purpose (the thin lines surrounding the “up”-button indicates the area is still there). The problem is purely cosmetic, and a work-around would be to move the “up”-button out of the area, since areas with no contents are removed automatically.](image-url)
8 Appendix 2: Instructions manual

8.1 Quick start

When the editor is started for the first time, you will be greeted by a welcome message with quick start instructions as seen in figure 32. These instructions are intended to give you some pointers to get you started, but for more in-depth instructions, please refer to this manual.

If you choose to disable the welcome message, you can enable it from the Settings menu later. The quick start guide can be shown at any time as a separate window from the Help menu.

Figure 32: Welcome message including quick start instructions shown upon starting.

If you choose to disable the welcome message, you can enable it from the Settings menu later. The quick start guide can be shown at any time as a separate window from the Help menu.
8.2 A short introduction and some useful words

The program that this manual is written for is called the Graphical PML Editor, or the “editor” for short. With it, you can design Graphical User interfaces (GUIs). A GUI is the part of a computer program or a mobile app that is displayed on the screen, and that the user interacts with. A GUI consists of parts like buttons, images, text fields etc. All of these “Lego blocks” that build the GUI are called Graphical components.

The GUIs you create with this editor will communicate with your devices using the PalCom system. PalCom helps different devices communicate with each other, even though the devices are not made by the same manufacturer (and wouldn’t communicate with each other normally).

Your GUI will communicate with your devices by sending and receiving commands. If you want to tell a camera to take a photo, for example, you can send a “take photo” command. Groups of commands that are part of the same functionality of the device are grouped into services. If a command carries information, like a text or image, the information is stored as a parameter. If for instance we want to send a text using PalCom, we will use a “Send message” command to send it, and the command will have a “Text” parameter containing the actual text we want to send. Devices, services, commands and parameters on a PalCom network are jointly called PalCom units. An example of this can be seen in figure 33.

![Diagram of PalCom units and hierarchy in an example device](image)

**Figure 33:** PalCom units and their hierarchy in an example device. You can communicate with the device using commands. If a command carries information, it will be stored in a parameter of the command (like the image of the “photo”-command.)

When you save a GUI you created in this editor, you will get a PML-file. PML is short for PalCom Markup Language, and it is a specialized computer language which is used to store the GUIs you create. A PML file has the file ex-
tension “.xml”, and will contain information about what graphical components
the GUI contains, and what all the graphical components should do (what de-
vices they should communicate with etc). Basically, the PML-file describes what
the finished program/app will look like and what it will do. If you want to edit
your GUI later, you can load the PML-file back into the editor and continue
where you left off.

To run a PML file (and actually display its GUI), a PML interpreter is needed.
The PML Interpreter available at the time of writing is called AndroidPUIDI,
and can be used to run PML files on Android phones and tablets.

8.3 Overview

After dismissing the welcome message, the editor will look something like figure
34. The editor has two main parts:

The palette part to the left in the window contains the network palette and
the command palette (1. and 2. in figure 34). These palettes focus on the
PalCom units such as devices, services, commands and parameters.

The canvas part to the right in the window contains the canvas and the op-
tions panel (3. and 4. in figure 34). The canvas part focuses on the
graphical components (such as buttons, text fields etc.) of the GUI.

Figure 34: Overview of the main components of the editor interface.
1. The network palette
2. The command palette
3. The canvas
4. The options panel
5. Save/Load/Help buttons

In addition to these two parts, we also have a menu bar (5. in figure 34)
with options for saving/loading, help, and settings for the program.
8.4 Camera example

In the following subsections, an example camera will be used to illustrate the usage of the Graphical Editor. The camera is not a real camera, but rather a computer program that simulates how a real camera might connect to a PalCom network.

If you want to try this example yourself, run the file `palcom-applications.jar` that is included in the PalCom download (which can be found at [http://palcom.cs.lth.se/PalCom/Download/Download.html](http://palcom.cs.lth.se/PalCom/Download/Download.html)). In the program, click `Device` in the menu bar, and select `New device...`. In the dialog that pops up, select `CameraDeviceGUI` as seen in figure 35 (its full name is `se.lth.cs.palcom.simulated.geotagger.CameraDeviceGUI`). When prompted with supplying a name, you may write any name you see fit. I choose to call it `GeotaggerCamera`.

![Device factory](device_factory.png)

*Figure 35: Adding the example camera.*

The camera will now show in the list of devices. However, you will also need to start it. This can either be done by right-clicking it and selecting `Start` as seen in figure 36, or by selecting it and choosing `Device` then `Start` in the menu bar. A picture of the camera will pop up in a new window. Please note that if you close this window, the camera will stop running (and need to be started again). It is however safe to minimize it, if you want to keep it out of the way.
8.5 The network palette

The editor puts a focus on PalCom units when creating a GUI, so in order to have something to work with we need to start by importing PalCom units from the network. This is done in the network palette (1. in figure 34). Please note that you need to have the devices you want to include in your program connected to a PalCom network. If no items show up in the network palette, you might want to check that your computer is connected to a PalCom network (the Graphical PML Editor is a PalCom unit itself), and that you only have one instance of the Graphical PML Editor running.

As can be seen in figure 37, I have various units connected to my PalCom network. If you followed the instructions in subsection 8.4, the Camera should be present. As an example, let’s create a basic GUI with the GeotaggerCamera device. In our example GUI, we will add functionality to take a picture (e.g. with a button), and then display the taken picture.
Figure 37: The network palette, with the PalCom units currently connected to the PalCom network.

We click the commands and parameters that are needed in order to import them. Note that the units turn yellow when imported. When a PalCom unit is imported, its parent units are imported automatically if not already imported (e.g. when a command is imported, its service and device are imported automatically).

For our example, we will need the following commands:

**takePhoto (in)** for taking the photo.

**photoTaken (out)** in order to know when the photo has been taken.

**getPicture (in)** to tell the camera that we want the photo we just took.

**photo (out)** which is a command with a parameter containing the photo we just took. We will use this command to display the photo. Note that we need to import the parameter (“img”) as well.

The small *(in)/(out)-text* next to the commands tells us the direction of the command. In-commands are commands from us to the device, and out-commands are from the device to us, e.g. *we tell the camera* to take a photo with the in-command *takePhoto*, and *the camera tells us* that the photo has been taken with the out-command *photoTaken*.

When all units are imported, the network palette should look something like figure 38. You can import units at any time, so it’s no problem if you’re not
sure about all of the units you’re going to need at this stage. Also note that you can remove imported (yellow) units by clicking them again.

Figure 38: Units are imported by clicking them, upon which they turn yellow. When a unit is imported, all of its parent units are imported automatically.

8.6 The Command Palette

The imported units show up in the command palette (2. in figure 34). The editor should now look something like figure 39 (but the order of the units might be different).
In the command palette, we get to choose what to do with our units. What we get to choose from depends on the kind of unit:

**In-command:** How should we trigger the command to be sent to the device? Maybe by pressing a button?

**Parameter of in-command:** How should the information be provided to the parameter? If it’s a text, maybe we want the user to input text in a text field?

**Out-command:** What should happen when the device sends this command to us? Maybe we want to display a pop-up dialog to the user, so the user knows what has happened?

**Parameter of out-command:** What should we do with the information of the parameter the device has sent to us? If it is an image, maybe we want to display it?

**Window:** In addition to the PalCom units, the windows of the GUI are also displayed in the command palette. If we have more than one window in the GUI we are creating, we need something to navigate between them (maybe buttons?).

Let’s illustrate this by continuing our camera-example. We begin by choosing how to send the takePhoto-command to the camera. As can be seen in figure 53.
40 we get a lot of choices for different graphical components that can be used to trigger the takePhoto-command to be sent. Let’s go for a button.

![Figure 40: The choices we get for triggering the takePhoto-command to be sent.](image)

We are however not finished after clicking the button in the drop-down list. We also need to drag it to the canvas. This is done by clicking and dragging the little purple “New LButton”-box in figure 41 to the canvas (3. in figure 34), which at the moment is empty and looks like in figure 42. When hovering the canvas while dragging the button-box, you should notice that the canvas turns purple. This means that you can drop the button.

![Figure 41: After we have selected a button for triggering takePhoto.](image)

![Figure 42: The empty canvas.](image)

After the button has been dropped, you will be prompted to give the button a reference name (figure 43). The reference is what the button is called when working with it in the Graphical PML Editor (and will not be displayed in the finished GUI). You should name it so you can recognize the button later. I chose to call it “takePhoto”, because that’s the command it should trigger.
After clicking “OK”, the editor should look like in figure 44. As you can see, three things have happened. Firstly, the button is now in place in the canvas. Secondly, a yellow label has been added to the command palette under the takePhoto-command, letting us know that the takePhoto-command is now sent when the button called “takePhoto” is pressed. Thirdly, the options panel (4. in figure 34) has been updated to show the options for our newly created button. We’ll leave the options for now though.

By clicking the small “(X)”-icon in the yellow “takePhoto (LButton)”-label in the command palette, we can remove the button as a trigger for takePhoto (if you accidentally just did this, you can find the button under “existing graphical components” in the takePhoto dropdown-list and select it again). If we click the rest of the label, we will highlight the button in the canvas and update the options panel to show options for the button.
So far, we have a button that tells the camera to take a photo. What we want to achieve now is to display the photo when the camera has taken it. The photoTaken-command from the camera tells us that the photo is indeed taken. We can send the getPhoto-command to the camera to make it send us the photo. We can use these two commands to get the photo when it’s been taken, by letting the photoTaken-command trigger the sending of the getPhoto-command.

This can be done in two ways as shown in figures 45 and 46, either by setting photoTaken to trigger getPhoto or by setting getPhoto to be triggered by photoTaken. The end result is the same regardless of which way we choose, as can be seen in figure 47.

Figure 45: One way to get photoTaken to trigger getPhoto.

Figure 46: Another way to get photoTaken to trigger getPhoto.

Figure 47: photoTaken now triggers getPhoto.
The final command we need to use for our camera GUI is the photo-command which is sent from the camera and carries our picture as a parameter called “img”. Similarly to when we created the button for the takePhoto-command, we simply choose “image” from the drop-down list of the img parameter, as seen in figure 48. This time we have four options for where to drag and drop the image on the canvas, namely above, below as well as to the left or right of the button. We chose to drop it below the button (where to place it is a matter of personal preference).

![Figure 48: Send the parameter of “photo” to an image.](image)

### 8.7 The Canvas and the Options Panel

The canvas (3. in figure 34) is a sketch of what the finished GUI will look like. At this point, it should look like figure 49 (if you gave the same reference names as I did). What the GUI will look like when it’s run depends on what platform it will run on after it has been exported (saved) from the editor, i.e. the same GUI might look different when run on an android phone/tablet or on a computer. What stays the same however, is the layout and what the different graphical components do.

![Figure 49: The canvas with a button and an image.](image)

Each of the graphical components in the canvas can be clicked in order to edit the settings of it in the options panel (4. in figure 34). For example, we might want to give the button we placed earlier a more relevant text. This is done by clicking the button in the canvas, and editing the “Set default text”-field as seen in figure 50 (I chose to set the text to “Take a photo!”).
The text of the button was the only option we needed to set for our camera GUI example. There are however some other aspects of the canvas and the options panel that are worth knowing. As you might have noticed, in addition to the button and the image, we can also click the background of the canvas (the purple area in figure 51). This brings up options for the window or layout part that is clicked. The editor will create new layout parts automatically if needed, depending on where the user places the graphical components from the command panel.
Another important aspect of the options panel are the general options, that are displayed if no particular graphical component’s options are displayed (or when you click the “Back to general options”-button in a graphical component’s options). The general options are displayed in figure 52. The general options contains a button to add a window, a button to access the application options and an option to add a static text label (just write your text in the text field, and drag the box to the canvas as usual).

If your GUI contains graphical components that have no specific place in the GUI (e.g. pop-ups), they will be listed in the general options. By clicking them in this list, you can edit the options for these graphical components. This list includes constants and variables.

If you choose to add a window by clicking the “Add window”-button of the general options, please remember that you need a way to reach it in the GUI. You can add buttons etc. to switch between windows in the window tab in the command palette, similarly to how we added the button to send the takePhoto-command. The main window is preset to launch upon starting the application.

As you might have noticed, when hovering the graphical components in the
canvas, a small “crossed arrows”-symbol will appear to the left of their refer-
ence names (see figure 53). You can move the graphical component to another
part of the canvas by clicking this symbol and dragging it to where you want the
graphical component. Please note: Sometimes the canvas might not recognize
mouse hovering. This is a known issue. You can however make the “crossed
arrows”-symbol appear by clicking the graphical components.

Figure 53: When hovering a graphical component, a small “crossed arrows”-symbol
will appear. Click and drag this symbol to move the graphical component to another
part of the canvas.

8.8 The Menu bar

The Menu bar (5. in figure 34, or figure 54) contains general options for sav-
ing/loading the current GUI, importing custom graphical components (such as
a custom button) or displaying the quick-start help.

Figure 54: Use this menu to save/load PML descriptions, import custom graphical
components etc. Please note that saving produces a PML description file that can be
run. In the “Help” menu, you can view the quick start help, or see who created this
program.

If you have followed the instructions, the editor should now look something
like figure 55. In order to run our newly created GUI, we need to export it to an
XML-file. The Graphical PML Editor saves and loads the same kind of file that
we use to run a PML GUI, and saving and exporting are thus the same thing.
Similarly, if you have a PML GUI you want to edit, simply load it into the
Graphical PML Editor by clicking the “Load”-button and selecting the PML
file in the dialog that appears. To save the GUI, click “File” in the menu bar
and then choose “Save”. You will be asked what to call your PML file, and
where to save it (if you don’t end the file name with “.xml”, it will be added automatically). Save it to a place where you will find it later.

Figure 55: This is what the editor should look like when the example camera GUI is finished.

8.9 Running a PML GUI on Android

Even though a PML Interpreter for Java is in the making, the only interpreter available at the time of writing is an interpreter for the Android OS (i.e. an app to run PML descriptions on an Android phone or tablet).

The AndroidPUIDI will be available on the PalCom website (http://palcom.cs.lth.se/Palcom/Palcom.html). For further instructions on running your PML file, please refer to the AndroidPUIDI instructions manual.

Good luck!
Appendix 3: Instructions given to Mia

During the initial evaluation, Mia was given the quick start guide (figure 56), a sketch of the GUI she was to create (figure 57) as well as the following description of the commands of the camera device:

**takePhoto (in) (Service: “Photo”-service)** Command for telling the camera to take a photo.

**photoTaken (out) (Service: “Photo”)** Command that the camera sends to tell us that a picture has been taken.

**getPicture (in) (Service: “Storage”)** Command for telling the camera to send the latest taken photo.

**photo (out) (Service: “Storage”)** Command with a parameter containing a photo. Sent in response to the getPicture-command.

Quick start guide

![Quick start guide](image)

*Figure 56: The version of the quick start guide at the time of testing.*
Figure 57: Sketch of what the finished camera GUI was supposed to look like.
References

[1] PalCom homepage
http://palcom.cs.lth.se/Palcom/Palcom.html


https://eclipse.org/windowbuilder/

https://netbeans.org/features/java/swing.html


http://digitalcommons.utep.edu/cgi/viewcontent.cgi?article=1010&context=cs_papers

http://webdesign.tutsplus.com/articles/visual-direction-in-web-design--webdesign-2545
Vad hade du gjort om du kunde styra dina apparater med en smartphone?


“Du ska inte behöva kunna programmera för att kunna styra dina apparater” har varit ett av huvudmålen under utvecklingen av Graphical PML Editor, ett datorprogram för att styra apparater uppkopplade med hjälp av ”Internet of Things”-lösningen PalCom. “Internet of Things”, eller ”Sakernas Internet” på svenska, är det ”nya fräcka” inom programmering, och innebär att vilken apparat som helst ska kunna vara uppkopplad till internet. ”Kul för programmerarna” kanske du tänker (och det har du rätt i), men delad glädje är dubbel glädje!


I sjukvården finns ett stort antal apparater av olika tillverkare. PalCom, som ”Internet of Things”-projektet på datainstitutionen vid Lunds Universitet heter, har hittills testats mot Lunds Universitetssjukhus. Inte minst inom hemsjukvården är det praktiskt att kunna mätta av apparater och se mätdata direkt i sin surfplatta. Med Graphical PML Editor kommer läkarna (om de vill) själva kunna skapa specialiserade appar för vad de behöver mäta.