Experience of Developing Advanced Mobile Thin-Client User Interfaces

Dejin Zhao

This thesis is submitted in fulfillment of the requirements for the degree of Master of Science in Computer Science, completed at the University of Auckland.

February 2005

Abstract

Mobile display devices such as mobile phones, PDAs, and tablet PCs have become very widely available and used. As a result, people now desire ubiquitous access to their software applications from a wide range of devices in different circumstances. However developing multiple User Interfaces (MUIs) specific to various devices for these kind of ubiquitous applications brings multiple effort and cost for software development companies.

To date, there has been a number of promising research projects to supporting the generation and development of MUIs for such ubiquitous applications. However they have not met with a great deal of success, and in particular no attempt has been made to provide advanced MUIs with high quality and usability. This then becomes the long-term motivation for our research.

Currently most application UIs on mobile phones and PDAs are limited to simple content like text, static images and motion video. The work presented in this thesis is a novel attempt to address on the provision of advanced UIs and interactions, such as dynamic diagrammatic browsing and editing, on very limited displays on mobile devices. It does so by supplying a set of innovative facilities including multi-user runtime configuration and multi-level zooming through webpage-based thin-client UIs.

In this thesis, we present our approach and a proof of concept prototype, Pounamu/Mobile, a plug-in to the meta-CASE tool Pounamu. Pounamu/Mobile generates advanced mobile thin-client UIs for any diagramming tool specified in Pounamu. We have also proposed the MUI Development Process that we have used to develop Pounamu/Mobile, and generalized our approaches to be applied to other applications in a wider range of domains. Finally we present an evaluation of our proof-concept work and conclude with our suggestions for future research.

Acknowledgements

Firstly and foremost I want to express my thanks for the guidance and support provided by my two supervisors, John Grundy and John Hosking. They provide me with an interesting research topic, open the door for me to the peak of academic research, and help me improve my writing and presenting skills. Without their ideas, enthusiasm, experience and knowledge my work would not have been possible.

I would like to gratefully acknowledge the help of the entire software engineering research group. A special thank you is due to Shuping Cao for her generous helps and talks during the last year. Thanks also go to Russell and Maree for their careful proof reading of this thesis.

I would like to thank all of the nameless souls that contribute to my evaluation survey. Clearly this thesis would not be what it is today without all of their kind help.

Heartfelt thanks go to my parents. You are always supportive, encouraging and loving, even when I am not. I have been blessed in many ways, but none more so than to have you as my family.

Finally, special thanks to David, who helped me find the new way to my life!

Table of Contents

List of Figures

List of Tables

Chapter 1 – Introduction

1.1 Introduction

With the widespread usage of more types of display devices, people want to access software from various devices in different circumstances. Applications and services accessible from various devices, which are called Ubiquitous Applications, are becoming popular nowadays and in the future. However developing multiple User Interfaces (UIs) specific to various devices for an application brings multiple efforts and costs for software development companies. To date, though there has been much promising research on supporting UI construction for ubiquitous applications, no technique has met with overwhelming success. This then is our research interest, providing tool support for designing and generating multiple UIs for ubiquitous applications.

In the rest of this chapter, we introduce ubiquitous applications, present difficulties of building multiple UIs and our motivation for the work, establish our general and more concrete goals for this thesis, describe our approach to carrying out the research, and outline the order in which the thesis presents our work.

1.2 Our Motivation

1.2.1 Introduction to Ubiquitous Application

As discussed above, we are interested in helping make applications and services accessible from multiple devices. Followings give several scenarios of kinds of ubiquitous applications.

Scenario 1: Someone using MS Outlook may make a schedule for the day in the early morning on his home PC. He may review tasks and edit some details on a PDA on the way to his office. The schedule and task list can be loaded up to both his office PC and mobile phone, so that he can be kept on schedule during a busy day when he is out of his office in discussion with colleagues in elsewhere. In this case, users want to access not only the same software but also the same set of data from different devices, and also they may want to switch between devices while completing a task. This scenario of using the same software but from different devices is called a Ubiquitous Application, and UIs of such an application are called Multi-device User Interfaces (MUIs). Figure 1-1 is MS office in PC and Pocket PC displaying meeting schedules.

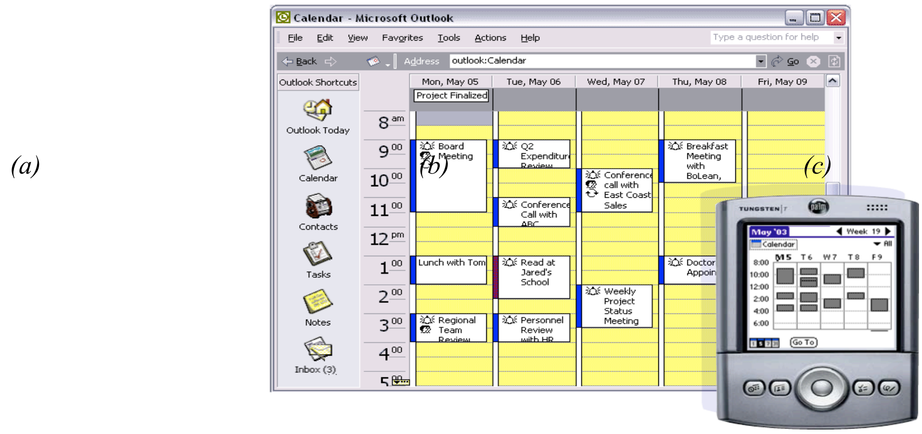


Figure 1-1: MS Outlook Calendar in PC and PDA

Scenario 2: Another example would be in a collaborative working environment. Multiple users may access the same application from various devices to accomplish a task together anywhere and anytime. Examples include on-line games, collaborative design, on-line meetings, and audio/video chatting, etc. E-commerce websites have been popular in the last decade. Many of them can already be accessed from mobile devices. Currently many software companies have started supporting their software on multi-devices by developing different versions specific display technologies for these different types of devices. Figure 1-2 show us a set of UIs of a project management tool(figure needs to be changed) specified by Pounamu [Zhu and Grundy04], including Java swing UI and web-based UI (b) on PC [Chao and Grundy04], and MUPE thin client UI on a mobile phone. Pounamu is meta-modeling tool that is used to interactively define new domain-specific tools and to provide a framework for realizing these tools.

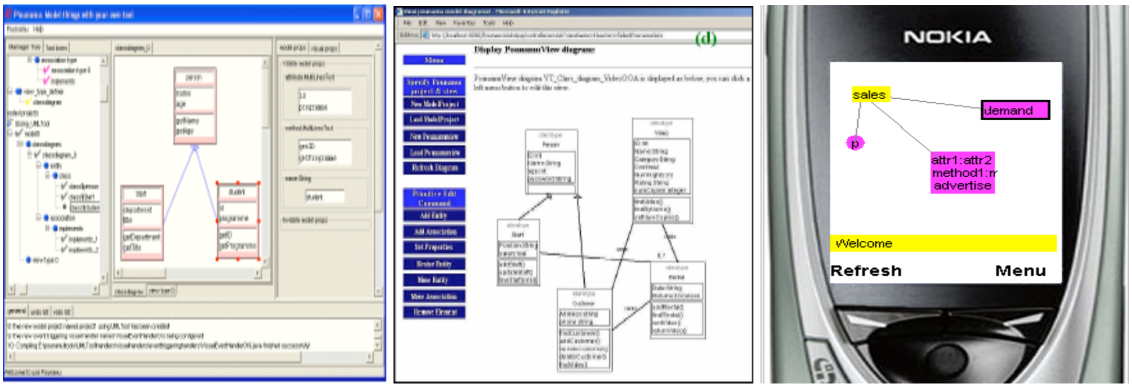


Figure1-2: Three type UIs of meta-tool Pounamu (a) Java Swing UI in PC, (b) web-based thin client UI in PC, (c) MUPE thin client UI in mobile phone (work presented in this thesis).

Scenario 3: Sometimes people involved in such collaborative tasks take different roles. For example, a civil engineer can use a PDA for gathering data when inspecting a new building to the office headquarters. A real-estate agent may use a mobile phone to add comments on it. Finally, another employee can use an office workstation under Windows/Linux to analyze the data and prepare a final report. During the process, employees act with the same information and services to play different roles from various devices. Ideally the multi-device UIs of the application should be adaptable to these different roles and user preferences.

All of these examples demonstrate that ubiquitous applications will bring data and information closer to users anytime and anywhere and supply a more flexible environment for people working together. Consequently, ubiquitous software will make people's lives more efficient and more productive, and thus will have large potential market demand in the future.

1.2.2 Difficulty of developing MUIs for ubiquitous application

To make quick and easy development of ubiquitous software, many aspects have been addressed by researchers, including underlying network protocols, security, data consistency and synchronization. In this research, we are interested in building User interfaces of ubiquitous applications, which is called as Multi-device UIs (MUIs). Characteristics of MUIs are given in the following based on the above scenarios:

- They are adaptable to multiple devices to allow a user to access data information and services using different UI and interaction styles.
- They are adaptable to multiple user roles and different user preferences to allow an individual or a group to achieve a sequence of interrelated tasks in a collaborative work.

Development of MUIs is difficult and time-consuming. Software developers need to be familiar with multiple programming platforms and UI design principles on various devices, and solve many basic common problems repeatedly for each platform. The design and implementation of multiple UIs may be very different due to different constraints of the various devices, including screen size, rendering ability, interaction types, and so on. Examples include a PDA with stylus, mobile phone with small screen and keypad input. A separate software design for each device is possible, but then the designers must find a way to combine all subsystems into a complete and consistent system. Many normal size software-developing companies may not be able to afford the double or even triple efforts to develop multiple UIs of an application. A unified environment and framework for building MUI applications would be very useful and helpful. This is the motivation of my current research and the potential PhD work. The main tasks that I aim to do include the following:

1.3 Our Goals

1.3.1 Long Term Goal

Our long-term research goal is to create languages and tools that support for the design and implementation of thin-client MUIs for ubiquitous applications. Our domain targets include:

- Thin client UI: MUIs will be built based on thin-client techniques to support pervasive access to applications and services, such as web-based UIs.
- Advanced UI: our application domain are software and services that require certain complex UIs to interact with, such as MS Outlook, rather than web pages only presenting information.
- High quality and usability: created multiple thin-client UIs should keep high quality and usability.

We present the long-term goal here to explain the motivation for the specific goal of the thesis in the following. As you will see later, when the state-of-the-art of MUI is presented, we discuss the reasons choosing these targets and discuss the detailed requirements.

1.3.2 The Specific Goal of the Thesis

Mobile phones have become one of the most popular devices and mobile UI is one of the primary UIs for MUI applications. The specific goal of this thesis is to explore issues of creating thin-client mobile UIs for applications that require advanced UIs with high quality and usability. The main tasks include:

- Identify common interesting problems and challenges applying to building thin-client mobile UIs for normal applications running on PCs
- Find or propose approaches to solve these problems
- Try to generalize the approaches to make them reusable as part of a library for creating MUIs, and abstract and automate the engineering process of designing and implementing mobile UIs
- Raise new requirements for our long-term goals in future work.

An even more concrete goal for the thesis is building mobile thin-client UIs for a meta-CASE tool Pounamu [Zhu and Grundy 04], so that any diagram-based visual modeling tool specified in Pounamu can be efficiently realized as a thin-client tool on mobile phones. Pounamu is a meta-case tool that we have been developing, which is used to interactively define new domain-specific tools and to provide a framework for realizing these tools. These modeling tools are typical applications, which provide not only normal UIs including standard menu, tree, and property sheet, but also diagramming features with complex interactions. We believe that successfully building mobile thin-client UIs for these diagramming tools will match our specific goal and contribute to the long-term goal.

In addition, we have been developing some extensions of Pounamu to extend access to tools and support collaborative modeling. These extensions provide different kinds of user interfaces for diagram-based modeling tools built with Pounamu, including single user thick client, multi-user collaborative thick-client [Mehra 2003], and multi-user thin client [Cao 2004]. They are constrained to use on desktop or laptop PCs. To further support pervasive access and collaborative model design, we wanted to make Pounamu modelling tools accessible on a range of mobile devices, such as mobile phones, PDAs, and the Pocket PC. Thus the Master project will make a significant contribution to this purpose as well. We will talk about more about Pounamu in the domain background section of the next chapter.

1.4 Research Approaches

Device-independent User Interface Literature Survey and Assessment. I will investigate thoroughly existing approaches to UI description languages, architecture and tool support for construction of MUI applications across a wide range of application domains and application features. This includes multi-platform adaptation, user and task adaptation, and multi-interaction and input. I will address the problems and issues identified in the existing tools and languages that try to generalize and simplify construction of MUI applications.

Exemplar Device-independent User Interface Applications. Following the literature review, we define more accurately the goals and essential requirements of our work. We explore issues of creating various device UIs of applications in our domain by actually identifying and working on exemplars. Then we collect or propose solutions for these issues and generalize these approaches and solutions. Specifically in this thesis, I will investigate issues about creating mobile thin-client UIs for diagram-based modelling tools generated from Pounamu. In long term, I will try and work with local IT companies to identify examples from their software products.

New Device-independent User Interface Specification. After experiences from building UIs on multiple devices, the next thing to do in future work is to abstract the aspects of multiple UIs and define a new device-independent user interface description language. We may also build several visual programming tools, where each focuses on an aspect of the construction of a MUI to maximize automation of designing and implementing UIs. Pounamu provides a rich exampler source for continually evaluating and revising our MUI Language.

1.5 Thesis Overview

This thesis document is organized as follows:
Chapter 1: Introduction – introduces the thesis topic and summarizes our motivation for choosing the research topic, our final goal and the step goal in this thesis, our approach to achieve the research goals.

Chapter 2: Background and Related Research – covers the background technologies and current research in the topic area.

Chapter 3: Requirements and the Development Process – discusses the key issues of our interests in supporting for the construction of MUIs of ubiquitous applications, defines the requirements for the Pounamu/Mobile, and presents our MUI Development Process, which is used for presenting our proof-concept prototype Pounamu/Mobile.

Chapter 4: Analysis and UI Design – describes the first three steps of the development of Pounamu/Mobile, which are mapping the system Use Case Diagram to the FSM, then abstracting UI designs from the thick client version, and presenting the design solutions for the mobile thin-client UIs.

Chapter 5: Usability Analysis and Our Approach – presents our main approaches to providing high usability advanced mobile thin-client UIs for diagramming tools generated by Pounamu.

Chapter 6: UI Realization and Approach Generalization– describes the class extraction form the FSM representation of the Pounamu/Mobile, more detailed designs for the main system use cases, and generalization of our main approaches.

Chapter 7: Evaluation – discusses evaluation results of applying the cognitive dimensions framework to the system, presents feedback from a small user survey, and discusses the strengths and weaknesses of the system.

Chapter 8: Conclusion and Future Work – summarizes the contributions made by this thesis to the field of research and avenues of future research indicated by this work.

1.6 Summary

In this chapter, we have established our research area, our motivation for doing this research, our general goal and step goal for doing the work, and the approach we take to achieve our goals. This introduction gives the overview for the rest of the thesis, which presents our work in more detail. The next chapter contains the background information and related research for our work.

Chapter 2 - Background and Related Research

2.1 Introduction

We gave an introduction to MUI and discussed the difficulties of developing MUI for ubiquitous applications in the last chapter. In this chapter, firstly we will see whether these difficulties can be overcome by automatically generating multiple UIs based on a single UI description? Are there any tools or development frameworks that support designing and implementing MUI? Is this previous work suitable for the targeted domains in our long-term goals? How they could contribute to our research work? In the section 3, we introduce background knowledge and techniques relevant to our target domains. This is basic information we feel needs to be understood before you can fully understand and appreciated the consequences of our work.

2.2 Related work about supporting for building MUI

As we pointed out in the motivation section, our goal is to help make software accessible from multi-devices. Up to now, imaginable ways to solve the problem are (1) creating multiple user interfaces for an application, which are respectively suitable on multiple devices; (2) general viewers on multi-device, such as, we currently have on PCs, VNC viewer and RDC (Remote Desktop Connection). The later solution is not feasible for controlling a remote machine by multi-user simultaneously, and also it is not feasible for mobile device users to use remote applications with UIs intended for big display screen on PCs. In the following, we review the state-of-the-art of the approaches to providing MUI and Adaptive UIs.

2.2.1 Automatic UI Generation

The idea of automatically generating user interfaces from abstract specifications is not new. It was explored in several systems in the early 1980's and the work was extended by at least a dozen systems since then. The interfaces created by these systems came to be known as model-based user interfaces because they were built from detailed models of program functionality, user knowledge, and presentation environment. One of these early systems was Mickey [Olsen Jr. 89], which automatically generated menus and dialog boxes from function signatures. Jade [Vander 90] is another example of an early model-based system for automatically generating dialog box layouts based on a textual specification of the content. Systems of the late 80's and early 90's, such as UIDE [Sukaviriya 93], HUMANOID [Szekely 93] and ITS [Wiecha 90] built more complicated models that could generate more sophisticated user interfaces. However, many of these systems generate UIs only for applications running on PCs but not other devices. More recently, a number of researchers have identified challenges on generating multi-device UIs, including [Lin 02, Nichols 02, Ponnekanti 01, Puerta 02, Weld 03]. In the following, we briefly introduce two of the most recent systems, SUPPLE [Gajos 2004] and PUC [Nichols 2004].

SUPPLE was recently created for automatically generating layouts for user interfaces. It takes three inputs: a functional interface specification, a device model and a user model. SUPPLE treats interface generation as an optimization problem.

When asked to render an interface (specified functionally) on a specific device and for a specific user, SUPPLE uses a branch-and-bound search to find the optimal choice of controls and their arrangement that meets the device's constraints. Optimality is determined by a cost function that minimizes the estimated cost (user effort) of the person's activity.

The Personal Universal Controller (PUC) system has been developed for automatically generating UIs for appliances in a pervasive computing environment, PDAs or mobile phones. Each interface is generated from an abstract specification of the appliance, written in a language that describes the complete functionality of the appliance. The PUC system will also be able to generate a single combined user interface for multiple connected appliances, such as a home theater. The automatically generated appliance interfaces have been evaluated by comparing to the functionally identical manufacturers' user interfaces.

These model-based UI generation approaches use a UI adapter engine to automatically generate specific device-dependent UIs based on an abstract functional specification and a presentation device model. However, they complement explicit end-user customization, and make it very hard for designers to influence how desired UIs are to be rendered. The common UI adapters are either restricted to a limited range of application domains, or lose specific UI features and quality when applied to generate UIs for general applications. Additionally, the current approaches have only been tested with generation of UIs for simple applications, such as remote control panels of appliances, and may not be suitable for complicated applications with advanced UI features like diagramming and complex data binding with the back-end application cores.

2.2.2 XML-Based Languages and Development Technologies

Apart from model-based MUI auto-generation, several XML-based UI languages and development technologies have been proposed, which allow users and designers to specify UIs under particular device constraints.

The User Interface Mark-up Language (UIML)[Abrams 99] is an XML-compliant language. UIML serves as a single language that permits creation of UIs for multiple devices. UIML is designed to be a relatively simple markup language with a little over two-dozen tags and is also independent of any user interface metaphor, such as graphical user interfaces or voice-response. UIML permits efficient compilation from UIML to any target language (*e.g.*, Java, C, HTML, WML, VoiceXML). UIML describes the appearance of a UI, the user interaction with the UI, and how the UI is connected to the application logic. Some work [Ali 02] has been done to make UIML more device-independent by adding task models in UI descriptions. However, UIML does not generate multiple UIs from a single UI description. The UI designer must still design separate UIs for each device, and then represent those designs in UIML.

The eXtensible Interface Markup Language (XIML) [Puerta 02] is a new general-purpose language for storing and manipulating interaction data. XMIL is capable of storing most kinds of interaction data, including the types of data stored in the application, task, and presentation models of other model-based systems. XIML was developed by RedWhale Software and is being used to support that company's user interface consulting work. They have shown that the language is useful for porting applications across different platforms and storing information from all aspects of a user interface design project. It may eventually standardize the specification of user interfaces and all necessary models. We may use it to describe our future MUI development approach in our long-term goal.

AUIT (Adaptive User Interface Technology) [Grundy and Zou 2002], developed at the University of Auckland, demonstrate how a UI markup language can work with an existing programming language. Scripts written in AUIT can be embedded in conventional Java server pages to provide a single adaptable thin-client UI for web-based applications. At run-time, AUIT uses the tag library classes to automatically create UIs for multiple devices such as desktop HTML and mobile WML-based systems, as well as highlight, hide, or disable interface elements depending on the current user and user task.

Other XML-based UI description languages include the following. The AUIML (Abstract User Interface Markup Language) Toolkit developed by IBM [IBM 04] provides software development tools that allow developers to write an application once and run it in Java Swing or on the Web without any changes. The XML User Interface Language (XUL) developed by Mozilla [XULPlanet] is used for creating rich, sophisticated cross-platform web applications on desktops. The "Longhorn" markup language, code-named XAML[Microsoft 04], provides a way to separate UI definition from logic and enables programmers to integrate code within or behind markup. Xamlon Inc. [Xamlon] has developed supporting tools for the XAML runtime library for the .Net and .Net Compact framework. W3C Device Independence Working Group [W3C 2005] is working on integrating web technologies into various devices.

XML-based markup languages with the help of model-based techniques can lead to an advanced framework for development of multi-platform UIs. They supply designers a single UI description language that wraps knowledge of multiple programming languages and platforms. Although some tools have been built on top of these languages, programmers still need to specify almost everything themselves, such as using which widget under which device. Most of them have a weakness in interfacing with the implementation and deployment platforms, such as data binding with the back-end application/database. Programmers still need to be familiar with the deployment platforms to integrate transformed UIs with the backend programs. In addition, advanced UI features such as diagramming have not been addressed in most of these mark-up languages.

2.2.3 Section Summary

Both model-based UI auto-generation and UI mark-up language have their strengths and weaknesses. Model-based approaches automate the process of designing and generating multiple UIs from an abstract UI specification, but lose the quality of and designer input into generated UIs. Mark-up languages are very flexible, but require considerable programming ability to create UIs due to the lack of intelligence and automation. Both approaches have not addressed advanced UI features and are independent from server-side platforms used to bind data to generated UIs.

One proposition is that we could effectively incorporate the flexibility of mark-up languages with the automatic generation of model-based approaches. However, how do we determine which aspects of creating MUIs should be flexible for user inputs, and which should be automatically generated? Also how do we apply both approaches to building thin-client UIs that cover our application domain: diagramming features, and improve the data binding on specific deployment platforms. Before we go further to answer these questions, we have to collect experiences of building MUI in our targeted domain, either from other previous related work or our own exemplar projects. We talk about this background knowledge and related work in the following.

2.3 Domain Background

2.3.1 Diagramming tools and the meta-model tool Pounamu

Diagrammatic applications are popular tools in a wide variety of domains. Examples include information structuring and browsing, concept sketching and discussion, project management, and other visual modeling tools such as software design tools [IBM Rational Rose], circuit designers [Quasar Electronics], visual programming languages [Burnett 95], user interface design tools [Myers 97], and children's programming environments [Cypher 95]. Many frameworks, meta-tool environments and toolkits have been created to help support the development of such visual designing environments. These include MetaEdit+ [Kelly 96], Meta-MOOSE [Ferguson 99], Escalante [McWhirter 94], PEDS [Zhang 01] and DiaGen [Minas 95]. But all current approaches suffer from either difficulty to use or lack of flexibility to provide support for a wide range of domains.

Pounamu [Zhu 2004] is a meta-tool developed at the University of Auckland. Pounamu is used to interactively define new domain-specific diagramming tools and to provide a framework for realizing these tools. Figure 2-1 shows the main components of the meta-tool including "Shape Designer", "Meta-Model Designer", "View Designer", "Event Handler Designer", and "Modeler". A user of Pounamu firstly specifies a meta-specification of the desired tool, using the Meta-Model Designer to define the tool's underlying information model as meta-model types, the Shape Designer to design the appearance of visual notation for meta-model types, the View Designer to define view types for graphical display and editing of information, and the Event Handler Designer to define behavior semantics of the tool. Having specified a tool or obtained someone else's tool project specification, users can create multiple project models using that tool in the Pounamu environment. Modelling tools allow users to create modelling projects, modelling views and edit view shapes, updating model entities. Figure 2-2 shows example tools created using Pounamu, including (1) project management, (2) circuit designer, (3) process modeling tool, and (4) web service composition tools

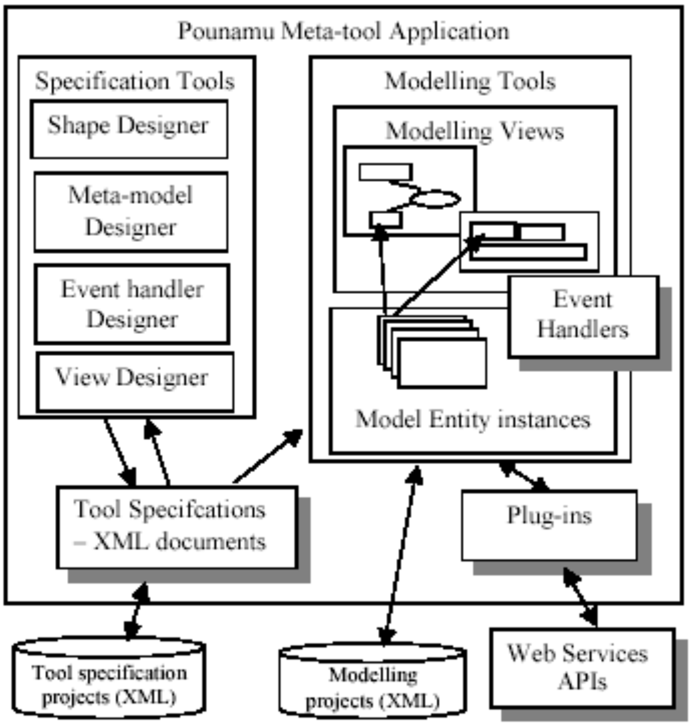
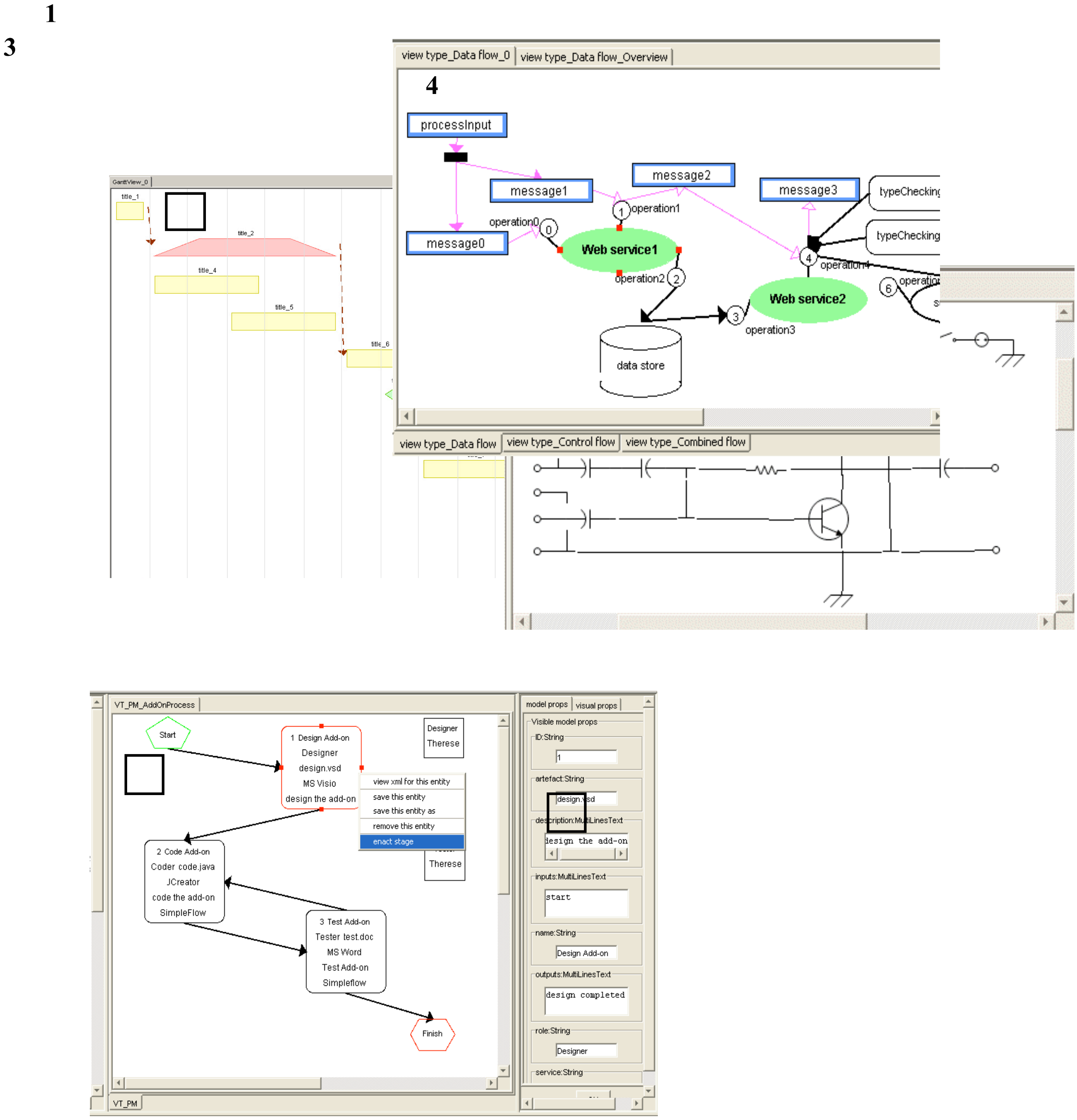


Figure 2-1: Component structure of the meta-tool Pounamu [Zhu 2004]

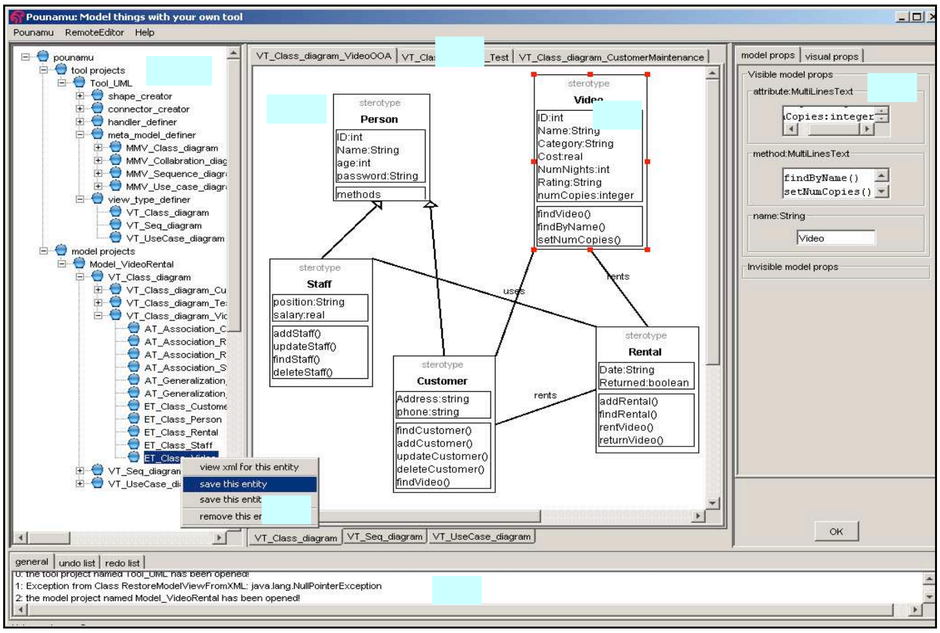


ID:String: 1
artefact:String: design.vsd
description:MultiLinesText: design the add-on
inputs:MultiLinesText: start
name:String: Design Add-on
outputs:MultiLinesText: design completed
role:String: Designer
service:String

Figure 2-2: Examples of Pounamu-specified tools in use

To support ease of use, the shape, view and meta-model designers use relatively simple appearance and semantics. To provide flexibility, the event handler designer allows tool designers to choose predefined event handlers from a library or to write and dynamically add new ones as Java plug-in components. Pounamu uses an XML representation for all tool specification and model data, which can be stored in files, a database or a remote version control tool. In addition, Pounamu provides a full web services-based and RMI API used to integrate the tool with other tools, or to remotely drive the tool.

Figure 2-3 shows the Java Swing-based UIs of the Pounamu environment for running specified modeling tools, which are composed of Windows widgets including the element management tree (1), pop-up or pull-down menus (2), drawing canvas (3), property editor (4), status window (5), and shapes (6) and tabbed-pane (7). Traditional diagramming tools, such as tools built by our Pounamu, are built using the thick-client, window-based interfaces. Although there is nothing wrong with their rich rendering ability and fast response speed, they suffer from some of the common problems of thick-client applications. For example, they often provide complex and difficult-to-learn UIs; they have to be installed and periodically updated on each user's computer, and they require quite heavy-weight infrastructure to support multi-user collaborative work, especially synchronous diagram editing. These problems have motivated researchers to pursue web-based applications.



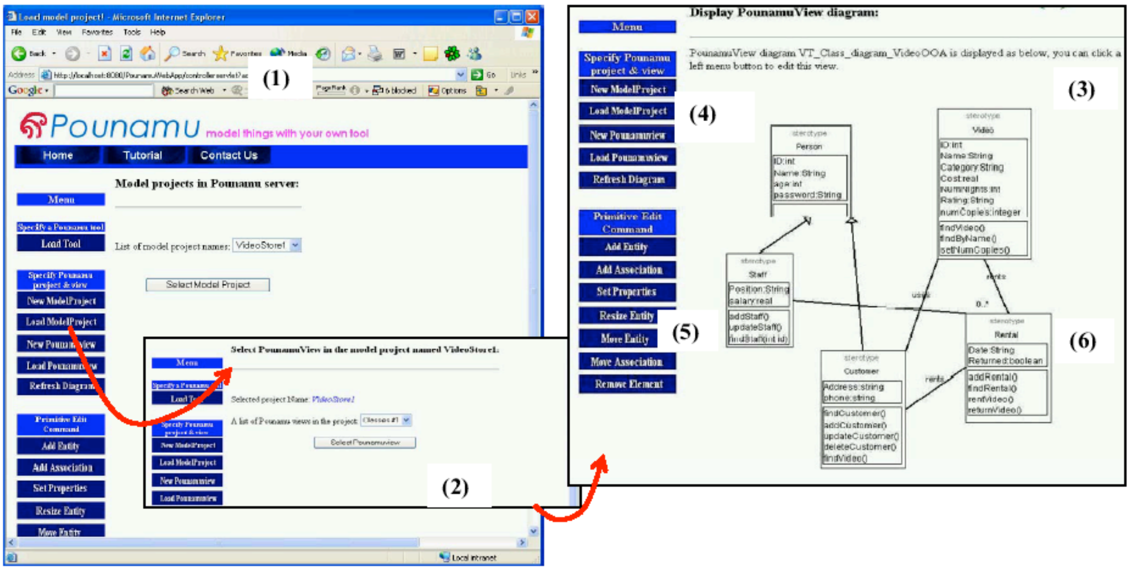
0: the tool project named Tool_UML has been opened!
1: Exception from Class RestoreModelViewFromXML: java.lang.NullPointerException
2: the model project named Model_VideoRental has been opened!

Figure 2-3: Pounamu-specified UML design tool in use

2.3.2 Thin-client UIs and thin-client diagramming tools

As web and related script languages have become popular, web-based UIs have become more pervasive in recent years. Many applications with traditional thick-client UIs have often developed thin-client versions. Examples include enterprise management systems, accounting packages, document management and email tools. Key design features of these web-based applications are a focus on simple, easy-to-use and consistent user interface design, seamless support for collaborative work via the client-server approach inherent to web applications, and server side integration with legacy systems and their facilities.

To improve access to diagramming tools, some web-based CASE tools have been proposed and prototyped following the above design principles. The recent research work includes Seek [Khaled 02], a UML sequence diagramming tool, NutCASE [Mackay 03], a UML class diagram tool, and Cliki [Gordon 03], a thin-client meta-diagramming framework. Pounamu/Thin [Cao 04], a thin-client diagramming extension for Pounamu, has been developed, allowing any specified tool to have a web-based thin-client editing UI. The extension is a set of Java servlets that implement thin-client diagram rendering via GIF images, or SVG with optional drag-and-drop move/resize and edit buffering. The original meta-tool interpreter is used to provide a shared application server for the thin-client tools. Multi-user diagramming is supported using this conventional web-based architecture. Figure 2-4 shows the web client UIs of Pounamu/Thin, when selecting a class diagram view to edit (1-2) and the resultant rendered diagram (3), in this example using a GIF image to render the Pounamu class diagram view. This web browser-based user interface includes a set of buttons to control the project (add views, open views) (4); a set of buttons to manipulate the diagram (add shapes and connectors, move and resize elements, edit element properties) (5) and a diagram, which can be clicked on to manipulate it (6).



Load model project! - Microsoft Internet Explorer

Model projects in Pounamu server:

List of model project names: VideoStore1

Select PounamuView in the model project named VideoStore1:

Selected project Name: VideoStore1

A list of Pounamu views in the project: Classes #1

Display PounamuView diagram:

PounamuView diagram VT_Class_diagram_VideoOOA is displayed as below, you can click a left menu button to edit this view.

Figure 2-4: Web client UIs of Pounamu/Thin with a

To further support pervasive access and collaborative model design, we wanted to make Pounamu modelling tools accessible on a range of mobile devices, such as mobile phones, PDAs, and the Pocket PC, by taking advantages of thin-client techniques. However, unlike desktop PCs and laptops, mobile devices have many constraints on screen display size, interaction techniques and bandwidth over mobile cellular networks. This makes existing thin-client diagramming approaches unsuitable for these devices.

To further support pervasive access and collaborative model design, in the last chapter, we indicated we wanted to make Pounamu-specified modelling tools accessible on a range of mobile devices, taking advantages of thin-client technique. We also wanted to explore the technical and usability issues of building diagram-based complex UIs on mobile devices. We can adopt a similar mechanism to the web-based Pounamu/Thin plug-in to build a mobile web server using Pounamu's plug-in mechanism. However, unlike desktop PCs and laptops, mobile devices have many constraints on screen display size, interaction techniques, rendering ability, and bandwidth over mobile cellular networks. This makes existing thin-client diagramming approaches unsuitable for these devices. In the next section, we review related research in developing thin-client diagramming tools on mobile devices.

Related work on mobile thin-client technologies

In recent years, cell phones and other mobile devices offer a range of sophisticated content including styled text, rich graphical images and full-motion video streaming. Many now offer quite high-resolution display capabilities along with a variety of advanced interaction techniques, such as stylus-based sketching. However, very few applications for mobile devices provide dynamic diagrammatic display and almost none provide interactive diagram editing. In the following, we present the state-of-the-art of thin-client technologies on mobile devices.

As the Wireless Application Protocol (WAP) Forum has standardized the Wireless Mark-up Language (WML) [WAP Forum], many companies, like Nokia, offer WAP browsers with their cell phones to allow users to browse mobile web pages. More recently, many server-side and client-side web-browsing technologies have been proposed and developed to allow web-based UIs intended for PCs to display on small handheld screens. Web Clipping [IBM 01, Oracle 99, Palm 01] transforms the original HTML page content to the HTML subset, which is rendered by a web-clipping client running on handhelds. XSL transformation [XMLtoAny, 2001] translates HTML pages to other formats including WML. XHTML [W3C XHTML], the next generation of HTML, makes web pages compatible with web browsers on multiple devices. Other approaches include zoomed-out copies of pages [Wobbrock 02, Milic 02], summarized versions of the page [Buyukkokten 01, Chen, L.Q. 02], outlining approaches that transform pages into sets of tiles [Björk 99, Buyukkoten 00, Wobbrock 02], combinations of tiling with zooming that allows users access individual tiles from an overview [Milic 02, Chen, L.Q. 02, Chen, Y. 03], and collapsing page content that allows users to zoom into relevant areas by collapsing areas that are less relevant [Baudisch 04]. To date however most of these techniques have been limited to text and form-based data and not suitable for richer diagrammatic content.

Many web browser plug-ins have significantly enriched 2D and 3D rendering ability of web-based UIs on desktop PCs, such as Flash, SVG, VRML, and QuickTime. Some mobile versions have been recommended or developed to improve thin-client rendering ability on mobile devices, such as SVG Mobile and Flash Pocket PC version. However, most of these technologies are restricted to data information browsing and static interacting, rather than dynamic editing and easy data transaction with server-side applications. Currently, online multi-user gaming brings many ideas for building both thick-client and thin-client collaborative working applications that require advanced UIs and interactions. MUPE, which is introduced in the following, is an open source client-server platform for creating mobile multi-user publishing/gaming applications, developed by Nokia [MUPE 2004]. The XML-based MUPE client script language supports not only the traditional form-based but also canvas-based UIs, such as diagramming. We chose MUPE as the implementation and deployment platform, because it also has research potential to be extended to support for creating MUIs for thin-client applications.

2.3.4 MUPE

MUPE Architecture:

MUPE, the Multi-User Publishing Environment, is an open source application platform for creating mobile multi-user context-aware applications, such as mobile games, virtual worlds, collaborative applications, and any other user authenticated services. As shown in Figure 2-5, MUPE consists of several components including Client, Server, Context, and Core. The MUPE client runs on any Mobile Information Device Profile (MIDP) enabled mobile device. The single thin client works like a web browser and can be used to connect to all MUPE servers in the network. The clients download and display UI and functionality from the server, described with MUPE client XML scripts. Applications can react to changes in the real world with the context-aware component. MUPE core, the connection framework, ties all the components together and it usually does not need any extra programming in the different applications.

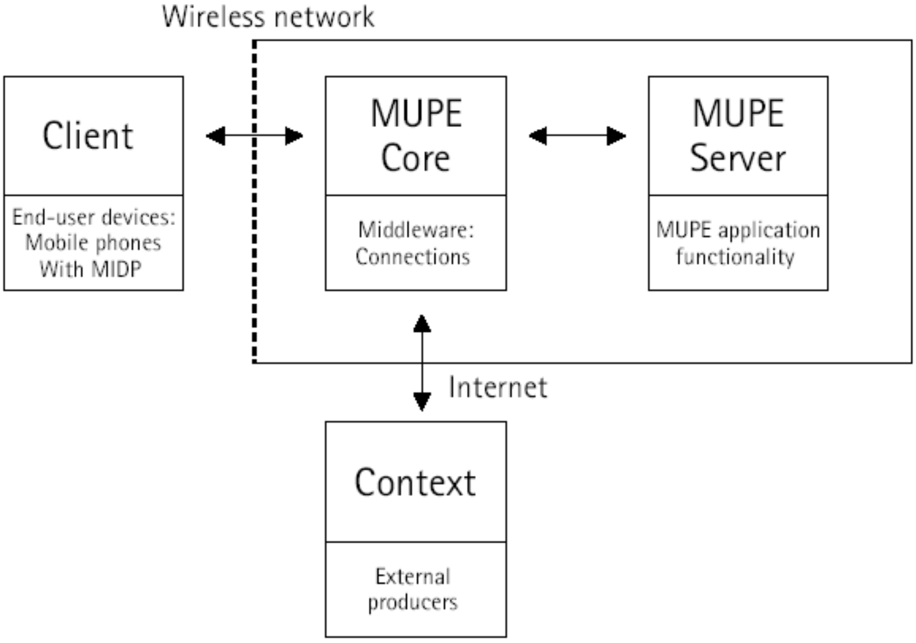


Figure 2-5: MUPE platform

MUPE Client and Script Language:

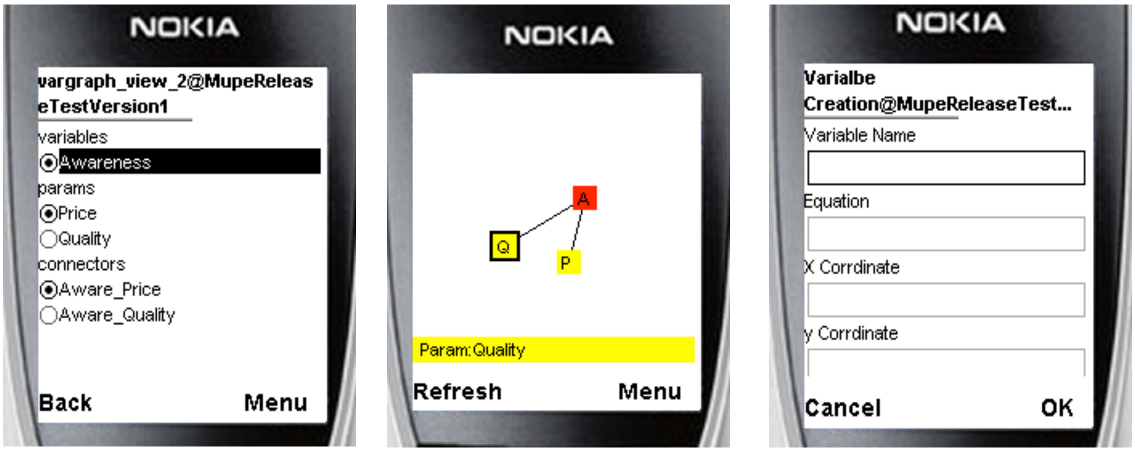
The MUPE client is implemented based on MIDP 1.0, which is an industry standard specification and is used in many current mobile devices. The MUPE client wraps most of the functionality of MIDP into its XML script language. The scripts currently support two UI styles: form UI and canvas UI. Form UI is mainly used to get text input from the end-users, which is composed of standard input widgets, such as text fields, edit box, radio button groups, etc. The canvas UI, graphical UI, supports more features for data visualization and advanced user interaction. The client script language is dynamic, for example once the initial UI items are loaded only the changes to the UI need to be sent. With properly designed scripts it is possible to minimize the number of network accesses, which is preferred in a high latency network usually associated with the mobile devices.

MUPE Server:

MUPE server is the main container for the applications. The applications can be created either by customizing dynamic XML UI description or writing extensions to the server code with the Java programming language [Java]. The server implements a user-authenticated persistent system where the end-users can register and re-connect with their accounts. The server contains three main objects: users, rooms, and items, where user and room objects can contain item objects. Each of these objects in the server has an arbitrary number of XML UI descriptions that defines its visual appearances at various stages. At the least there are two XML description files for each object: creator and description. The creator defines how the mobile end-user can create a new object of that type and the description defines how an object of that type is shown to the user. The visual appearances of the objects can be customized by modifying the XML files and the functionality can be extended with new Java code. The XML files can be customized while the server is running, whereas extensions to the Java code require a compile and restart to the server.

MUPE Programming and Examples:

MUPE aims to support rapid development of mobile client-server applications. It needs a little programming since many parts of the system can stay the same in different applications. As we discussed in the above, the simple way to create a new application is to reprogram the MUPE server and modify the XML UI description files. The following figure shows a simple diagramming tool we created using MUPE; the corresponding XML UI descriptions are in Appendix A.



vargraph_view_2@MupeReleaseTestVersion1

variables
● Awareness
params
● Price
○ Quality
connectors
● Aware_Price
○ Aware_Quality

Param:Quality

Varialbe Creation@MupeReleaseTest...

Variable Name

Equation

X Corrdinate

y Corrdinate

Figure 2-6: A simple Equation-based Business Modelling Tool

2.4 Summary

In this chapter, we have reviewed recent approaches to providing MUI. Systems that automatically generate multi-device UIs from a single abstract UI description sacrifice user input into UI design and limit the range of application domains. Markup language-based development frameworks allow users flexibility in specifying their desired UIs, but require work in integrating translated UIs with the back-end applications. XIML has potential to standardize UI description and we may choose it to present our MUI construction techniques in the future. AUIT is most related to our research goal, which generates adaptive web-based UIs in run-time from the JSP server. Many of its ideas contribute to the thesis, although, same as most of other techniques above, its approaches are not quite suitable for generating advanced and high quality diagrammatic UIs.

After giving a brief overview of the state of the art, we gave the background knowledge and current techniques in our research domain of creating thin-client MUI for diagramming applications. While reviewing previous related work on diagramming tools, thin client diagramming tools, and mobile thin-client technologies, we found that little has been done relevant to our goal, especially mobile thin-client UIs support diagramming features. We talked in more detail about Pounamu, the meta-CASE tool for creating other new diagram-based modeling tools, and MUPE, the mobile thin-client platform. Pounamu and MUPE provide our most significant research background context. In this thesis, we experience and explore issues of creating Pounamu/Mobile, using MUPE, which generates thin-client mobile UIs for any specified tool in Pounamu. In the future, Pounamu continually supplies application source for us to evaluate our MUI approaches, and MUPE is the platform, which we will implement our ideas on and integrate our approaches with.

Chapter 3 - Requirements and the Development Process

3.1 *Introduction*

This chapter we discuss the motivation of our long-term goal and the thesis specific goal. We present the general requirements for the kind of MUI systems that we are interested in providing tool support for. Then we describe the concrete requirements for the exemplar system that we have developed in the thesis work. Finally we present a multi-device thin-client UI development process that we have proposed and used for developing the mobile thin-client UI system for Pounamu. We present this process here, because it is also being used for presenting our proof-concept work in the following chapters.

3.2 *Our Long-term Goal and the requirements*

As we described in the Introduction Chapter, providing tool support for MUI development for ubiquitous software applications is our long-term motivation. After the literature review in the last chapter, we defined more accurately the goals of our work, and identified the key requirements for a typical MUI system developed by such a tool:

- The quality and usability of the multi-device UIs to the end-users of the ubiquitous software are what we think the first priority.
- The MUIs should contain advanced UI features, such as wizards and diagrammatic features, besides the popular form-based UI features, to allow the end-users easily to learn the system, especially on those devices that have limited ability, such as small screens on mobile phones.
- The MUI system needs to provide the web-based thin-client UIs. This is because thin-client offers a lot of advantages to applications versus the traditional thick-client approach as we discussed in the last chapter, and thin-client particularly matches the access requirement of ubiquitous applications: anytime, anywhere, on any computer devices. Additionally, thin-client technologies naturally support multiple users and collaborative working.
- To address on the usability and quality of the MUIs, an ideal tool should provide facilities to allow psychologists, graphic artists, and UI designers to easily input, test, and modify their design ideas on multiple devices.

So, in a word, our long term goal is to provide tool support for developing advanced thin-client MUIs for ubiquitous applications, with high quality and usability for the end-users. However, as we pointed out, none of the previous research work has addressed these problems that we think are important to the end-users of MUI applications. Therefore, firstly we have to go ahead to collect experiences by identifying and building exemplars by ourselves, before we can really identify the concrete requirements for tools and languages to be created to support development of such MUI system.

3.3 The specific goal of the thesis

Our research background here in the University of Auckland supplies us a huge source of such a kind of applications, diagram-based visual modeling tools defined by a meta-CASE tool Pounamu[Zhu 04], which require such MUIs that match our key requirements exactly. These modeling tools require advanced UIs, such as a range of diagramming features, a certain high quality and usability that is necessary to support end-users modeling effectively, and thin-client access from a wide range of devices to support pervasive collaborative design. Pounamu/Thin [Cao 04], a thin-client diagramming web server for Pounamu, has been developed, allowing any specified tool to have a web-based thin-client editing UI on PCs. We would like to extend this work to provide access to these Pounamu-specified modeling tools from a wider range of devices, such as mobile phones, PDAs, and tablet PCs. Ideally we want to build one server, which could intelligently generate suitable web-based UI page according to user requests from various devices. Alternatively, we can use multiple UI servers with a central manager server, which is in charge of things like user management and collaborative environment.

However as we pointed out in the last chapter, few mobile applications have been developed with advanced UI features like diagrammatic browsing, and no attempt has yet been made to provide dynamic diagrammatic editing on mobile devices. This attracted us to firstly experience creating thin-client diagramming UIs on mobile devices. Our specific thesis goal is building a mobile thin client UI server for Pounamu, which we called Pounamu/Mobile. Pounamu/Mobile aims to dynamically generating mobile thin-client UIs for any modeling tools defined using Pounamu. To give you an overview idea about the system being built, Figure 3-1 shows a high-level system diagram with the scenario that a client user is modeling a class diagram using the UML tool created and running on the Pounamu Application Server.

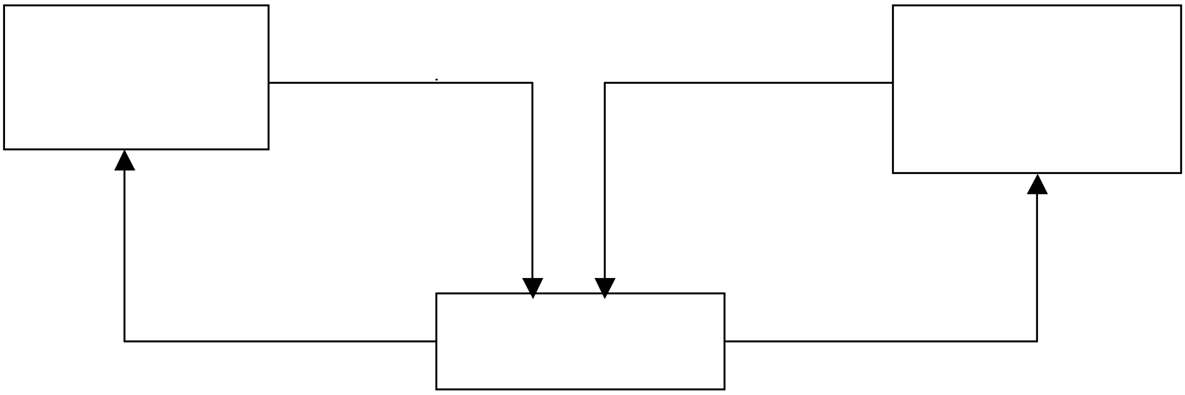

Mobile Client
(1) A user, who is designing a class diagram using the UML tool defined
(3) Send the updated class diagram view data including the new created class
Pounamu (where the UML is running)
(4) Generate a mobile webpage for rendering the class diagram view data on the user's device
Pounamu/Mobile
(2) send the command to Pounamu and request the updated class diagram view data

Figure 3-1: Pounamu/Mobile high level system diagram

3.4 The Requirements of Pounamu/Mobile

System Requirements: Based on the goal of the thesis, we directly obtain the system requirements, which are thin client, plug-in to the Pounamu, deployed on a range of mobile devices, multi-user, and supporting collaborative editing.

Functional Requirements: We don't need to perform a detailed scenario study of use cases, because the problem has been well defined in the Pounamu. In this project, Pounamu/Mobile is supposed to realize a similar set of functionalities to Pounamu, but through thin-client UIs on mobile devices. The following figure shows the system use case diagram, which represents the system functionality requirements.

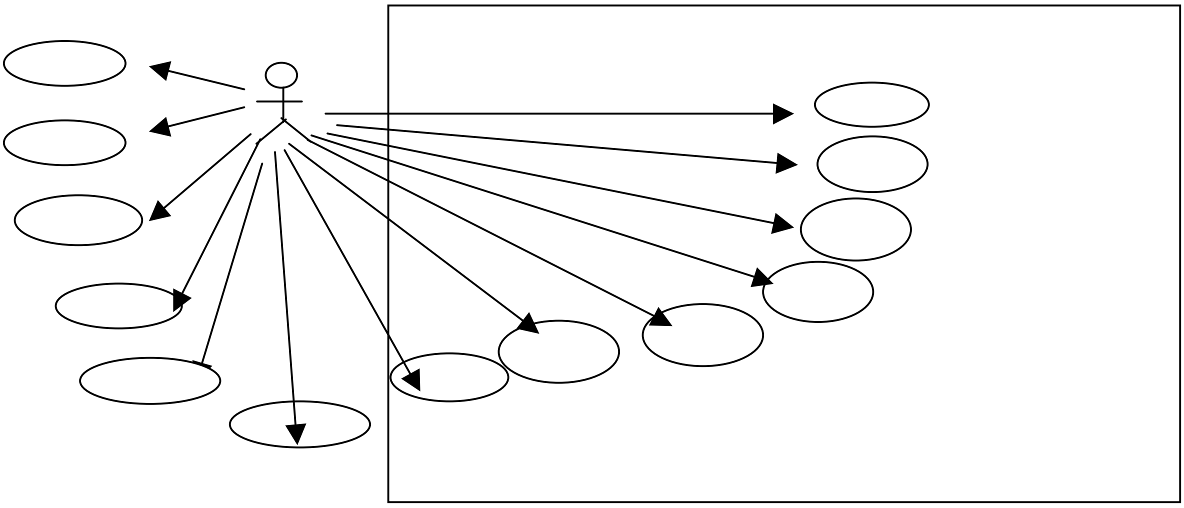


Figure 3-2: Use Case Diagram of Pounamu/Mobile

Usability Requirements: There is a set of constraints on mobile devices different from those on PC. These constraints include screen size, input ways, memory size, and CPU power. They affect the quality of data visualization, effectiveness of user interactions, and difficulty of learning and using the system.

Due to the small screen can't display all information as the Pounamu's thick client and thin client versions do on PC. So what basic necessary information has to be displayed for users to complete different tasks? For example, a UML tool, a full screen might be used to display only one class shape when users want to browse the information detail or edit its properties. The UIs should at least display a part of a view when users request to browse associations and their dependencies. Different tools and even different circumstances of using the same tool, it may require different necessary information to be displayed for completing a particular task.

Due to the limited input way on most mobile phones, some user interactions used to be done by a combination of keyboard and mouse have to be realized on mobile phones using only phone keys. However the mobile thin client UIs should create some facilities to complement this limitation to ensure certain usability. Many operations and tasks in the system use case diagram presented above are affected by this constraint, such as moving shapes, creating associations, and selecting elements.

Based on the above analysis, we set up the usability requirements for the Pounamu/Mobile:

1. Cognitive UIs should be provided for end-users to easily learn and use the modeling tools
2. A readable overview of a Pounamu View inside one mobile page,
3. Diagram details should be quickly and easily visible as a user requests. Ideally, the UI can fit a whole Pounamu View in one page and display all details of diagrams, though this sounds impossible.
4. Diagrams should be able to represent their problem domains. The worst case is that all different modeling tools appear the same.
5. Diagram rendering should be adaptable to users' preference.
6. Users should be able to easily navigate a large Pounamu View

3.5 The MUI Development Process

Based on the experiences of task-based modeling from the previous work, such as UIML and XIML, during developing the Pounamu/Mobile, we proposed a development process, which we called **MUI Development Process.** We designed and implemented the mobile thin-client UI system for Pounamu by using this process, where we experienced a set of benefits from merely using the traditional OO approach. However due to limited time, we haven't developed a specification for this process. To better convey what we do in the following chapters, we present the process here in Figure 3-3.

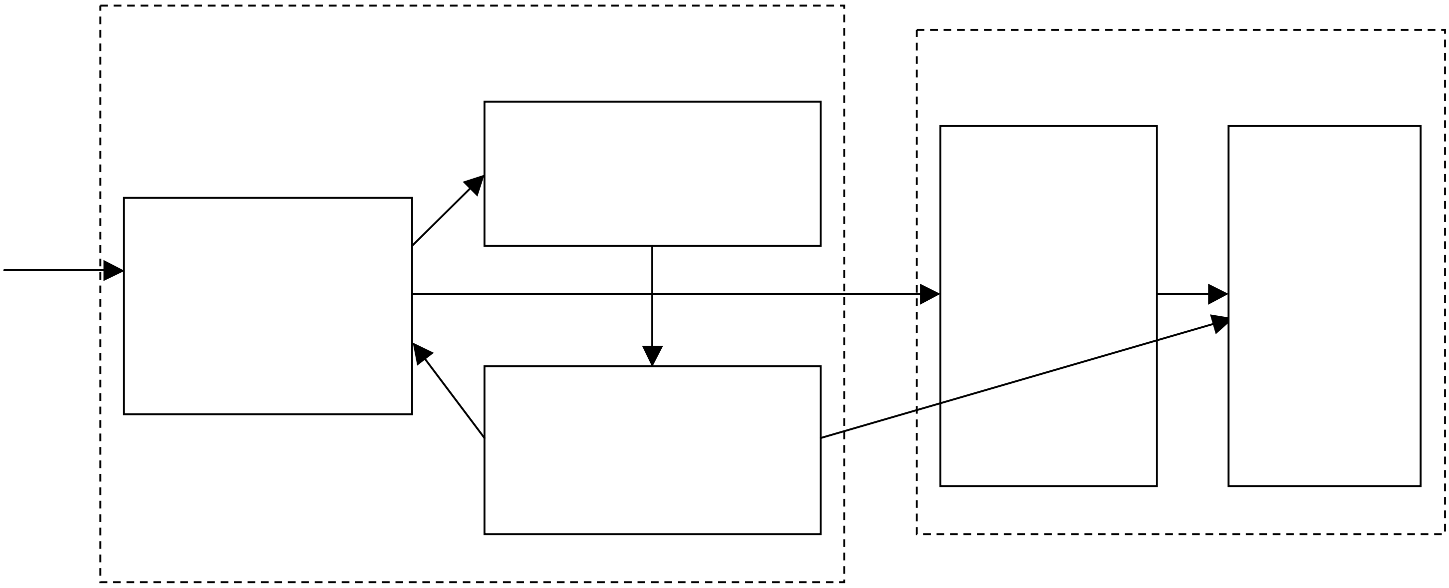

Analysis and UI Design
Realization
(2)
Design abstract UI interaction for each task state, using the state concept
Extract classes and functions by mapping states to classes, and events to functions
Further Implement
Use
Represent a UI system using a FSM by mapping use cases to task states, with supports of location states
(5)
(6)
(3)
Design device specific UIs based on the abstract UIs and the device constraints
(6)
(4)
Refine

Figure 3-3: The MUI Development Process Diagram

3.6 Summary

We discussed the in-depth motivation for our long-term and the thesis specific goals, and described the requirements. We presented our MUI Development Process, which is used for presenting our proof-concept prototype, Pounamu/Mobile, in the following chapters. As going through each steps shown above, we discuss the benefits of using the MUI Development Process to develop thin client UI systems, present and generalize our approaches to providing advanced UIs for mobile thin client system, and discuss the potential directions to future work.

Chapter 4 –Analysis and UI Design

4.1 Introduction

We describe the Analysis and UI Design Workflow of the Pounamu/Mobile system using our MUI Develop Process, which was presented in the last chapter. We went through the workflow twice. In the first cycle, which is presented in this chapter, we created the first version of Pounamu/Mobile by only taking the functional requirements into account. In the second cycle, which is presented in the next chapter, we improved the UIs of the first version according to the usability requirements. We did so because we felt we couldn't attempt too much the first time when creating such a innovative system with little experience from the previous related research, and second, a rapid prototype would help us identify more concrete problems and establish more exact requirements for further improvements.

In this chapter, firstly we introduce the FSM and list its advantages for construction of thin-client UI systems, which we have experienced during the development of the Pounamu/Mobile. Then we represent the system functional requirements as an abstract FSM by mapping the system use cases to the event-driven states in the System State Diagram. Then we describe the mobile thin-client UI designs for the states by firstly mapping the original thick client UI interaction designs to the abstract device-independent descriptions for each use case (task/task state), and then transforming them to our Pounamu/Mobile specific UI designs based on the constraints and facilities of mobile phones. Finally, based on the ***Experiences*** discussed through the chapter, we summarize the benefits of using our MUI Development Process in this workflow and discuss the possibility of the process automation.

4.2 Finite State Machine (FSM)

Finite State Machine (FSM): also known as Finite Automaton, which was originally used for describing computation models in computation theory [Sipser, M 97]. Lately FSMs have been very widely used in modelling of application behaviour, design of hardware digital systems, software engineering, study of programming languages, and even 3D computer games for controlling robots in Quake 2 and Warcraft III [Meyer, N. 03].

Though FSMs have been used widely in many areas, they have not been successful for software modeling in recent years. This is mainly because a huge number of states and events in a big software application will cause a messy FSM representation, and so make it hard for user modeling, though this problem could be potentially solved using sub states. However, web-based thin-client UI systems, unlike the traditional software systems, dramatically reduce the UI complexity and the number of events available in a single page, and therefore reduce the complexity of the FSM representations. So it is potentially feasible to use FSMs to support describing, designing, and implementing thin-client UI systems.

Based on our previous software development experiences and user survey, we feel that it is a natural way for people to describe a system's functionalities by starting from scratching the simple UI diagrams, which represent the system states in the FSM, and then designing UIs in detail for each of these states. Due to limited time, we haven't done enough review work to find any previous work relevant to using FSMs to support designing thin-client UI systems. We list the advantages that we have experienced during creating the mobile thin-client UI system for the Pounamu. As we describe our ***experiences*** though this chapter later, it will become evident how our work benefits from using the FSM.

The advantages of using FSMs in developing thin-client UI systems

- Easy for describing event-based systems. The UI system of a software application is a typical event-based system. FSMs help in designing and implementing event-based systems, and support mapping between a FSM representation and its implementation
- Help describe a user's mental functional requirements of a UI system and take them one step further to the final UI implementation.
- Supports abstracting UI interactions. It helps to understand use case scenarios of a UI system and support describing the use cases in detail using abstract UI interaction steps.
- Supports mapping abstract UI designs to device specific UIs. It helps us to better understand the mapping process between device-independent UI descriptions and the concrete UI interaction designs.
- Improves the visibility of the UI implementation. Using the state concept makes the program code much clearer, and therefore easy to be mapped back to the visual representation. Without it, other condition statements, such as if-condition, would be hugely used to check system statuses and tell different meanings from messy similar event sequences.

4.3 Map the Use Cases to the System States

In traditional OOA, State Diagrams are normally used at the last step and play a similar role to Sequence Diagram, to describe events and message passing between objects. We used it at an early stage of the requirement analysis of this UI system for a set of reasons as listed above. However, mainly because it helps describe our conceptual functional requirements and helps us understand the use case scenarios of the system to be built, and supports further UI design presented in the next section.

For convenience, we put the Use Cases Diagram (Figure 4-1) obtained from the last chapter here. Figure 4-2 shows the original System State Diagram mapped from the use case diagram. We explain this mapping process in detail in the following, together with the mapping table 4-1.

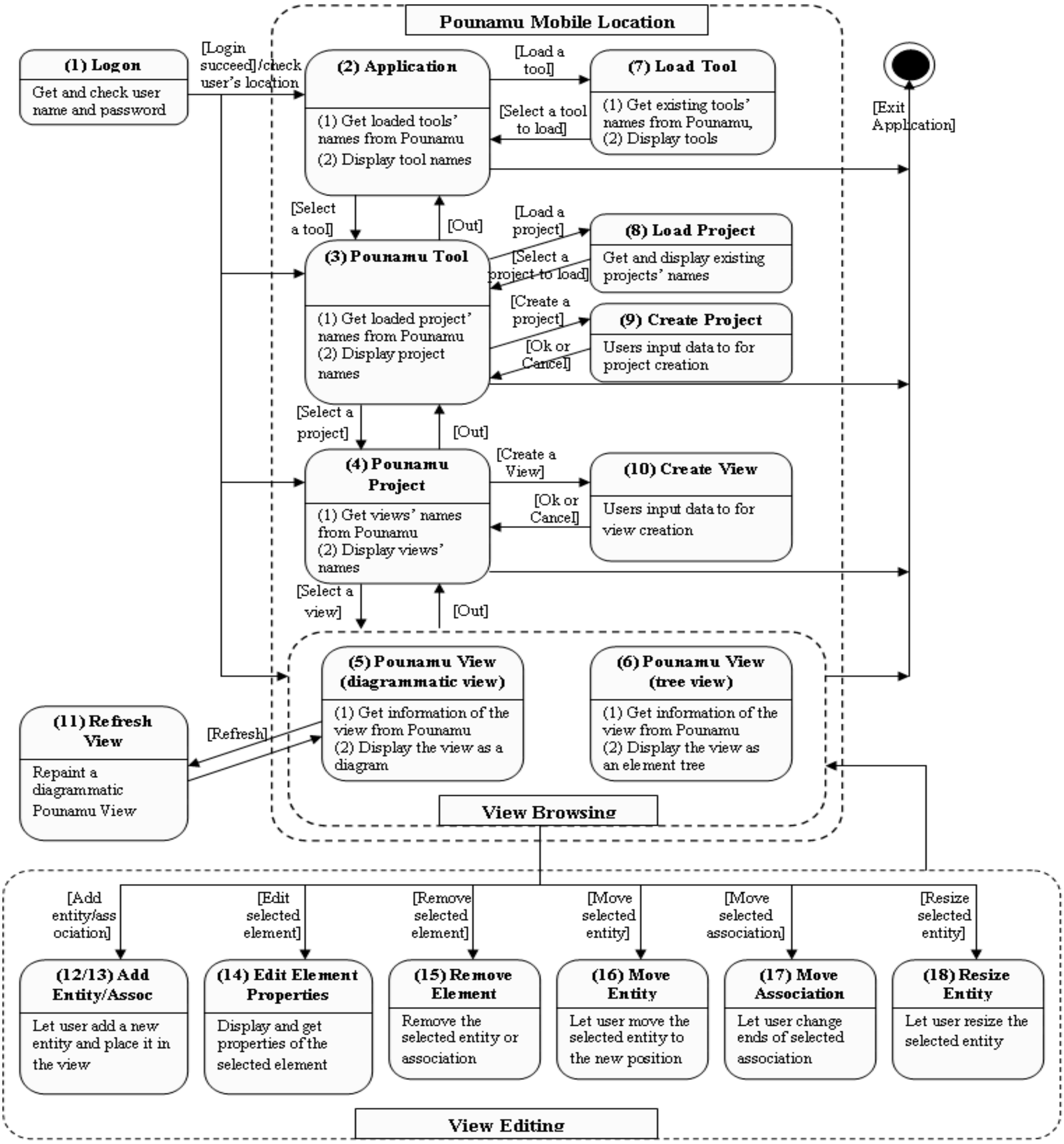

Pounamu Mobile Location
(1) Logon
Get and check user name and password
[Login succeed]/check user's location
(2) Application
(1) Get loaded tools' names from Pounamu
(2) Display tool names
[Load a tool]
[Select a tool to load]
(7) Load Tool
(1) Get existing tools' names from Pounamu,
(2) Display tools
[Exit Application]
[Select a tool]
[Out]
(3) Pounamu Tool
(1) Get loaded project' names from Pounamu
(2) Display project names
[Load a project]
[Select a project to load]
(8) Load Project
Get and display existing projects' names
[Create a project]
[Ok or Cancel]
(9) Create Project
Users input data to for project creation
[Select a project]
[Out]
(4) Pounamu Project
(1) Get views' names from Pounamu
(2) Display views' names
[Create a View]
[Ok or Cancel]
(10) Create View
Users input data to for view creation
[Select a view]
[Out]
(5) Pounamu View (diagrammatic view)
(1) Get information of the view from Pounamu
(2) Display the view as a diagram
(6) Pounamu View (tree view)
(1) Get information of the view from Pounamu
(2) Display the view as an element tree
(11) Refresh View
Repaint a diagrammatic Pounamu View
[Refresh]
View Browsing
[Add entity/association]
[Edit selected element]
[Remove selected element]
[Move selected entity]
[Move selected association]
[Resize selected entity]
(12/13) Add Entity/Assoc
Let user add a new entity and place it in the view
(14) Edit Element Properties
Display and get properties of the selected element
(15) Remove Element
Remove the selected entity or association
(16) Move Entity
Let user move the selected entity to the new position
(17) Move Association
Let user change ends of selected association
(18) Resize Entity
Let user resize the selected entity
View Editing

Figure 4-2: The original System State Diagram of Pounamu/Mobile

Basically, we map each use case/task to a state, which we call as ***task state***. The state diagram introduces two additional kinds of information that the Use Case Diagram doesn't have originally. Firstly it introduces information, such as events and conditions, which link all states together and drive the system among various states. Secondly, it introduces another kind of state, other than the task states directly mapped from tasks, which we call as ***location state***. These additional states are necessary for linking task states together, in order to better describe how a system works.

For example, based on our experience of Pounamu, we added the state “Pouanmu Tool” as a location state to provide users the access to performing the tasks “Load a Pounamu Project” and “New a Pounamu Project”. We did so because the system can’t perform these two tasks from the starting state without any tool already being loaded in the system. We also added location states “Application”, “Pounamu Project”, “Tree View of a Pounamu View”, and “Canvas View of a Pounamu View”. We describe each state in detail in the following table. We added events to drive the system from the location states to the task states. For example, the event “Load a Tool” in the location state “Application” drives the system to the task state “Load Tool”. We also added events for driving the system between location states, such as the event “Using the tool” driving the system from the “Application” state to the “Pounamu Tool” state.

As you may have noticed, instead of mapping the use case “Load Pounamu View” to a task state, we use two location states to represent it, because the use case is to display contents contained in a Pounamu view using either a tree or a diagrammatic UI and displaying contents is one of the basic features of a location state as described in the mapping table. In addition, to make the state diagram tidy, we group the states to three bigger states, which are “Pounamu/Mobile Location”, “View Browsing”, and “View Editing” state.

Table 4-1: State Description of the original System State Diagram

State name	**Mapped from use case/task**	**Description**	**Other states drive to by its events**
1. Logon (start state)	Introduced	Allow users to log into the system, drive the system to	location state: 2, 3, 4, 5, 6
Location States			
2. Application	Introduced	Display all Pounamu tools loaded in the system and event options	location state: 3 task state: 7
3. Pounamu Tool	Introduced	Display all Pounamu projects loaded using the tool and event options	location state: 4 task state: 8, 9
4. Pounamu Project	Introduced	Display all Pounamu views in the project and event options	location state: 5, 6 task state: 10
5. Pounamu View (diagrammatic)	Load Pounamu View	Display contents of a Pounamu view in a diagrammatic view and event options	location state: 4 task state: 11, 12, 13, 14, 15, 16, 17, 18
6. Pounamu View (tree)	Introduced	Display contents of a Pounamu view in a tree view and event options	location state: 4 task state: 11, 12, 13, 14, 15, 16, 17, 18
Task States			
7. Load Tool	Load Pounamu Tool	Perform the task and display event options	location state: 2
8. Load Project	Load Pounamu Project	Perform the task and display event options	location state: 3
9. New Project	New Pounamu Project	Perform the task and display event options	location state: 3
10. New View	New Pounamu View	Perform the task and display event options	location state: 4
11. Refresh View	Refresh Pounamu View	Perform the task and display event options	location state: 5
12. Add Entity	Add Entity	Perform the task and display event options	location state: 5, 6
13. Add Association	Add Association	Perform the task and display event options	location state: 5, 6
14. Remove Element	Remove Element	Perform the task and display event options	location state: 5, 6
15. Edit Element Properties	Edit Element Properties	Perform the task and display event options	location state: 5, 6
16. Move Entity	Move Entity	Perform the task and display event options	location state: 5, 6
17. Move Association	Move Association	Perform the task and display event options	location state: 5, 6
18. Resize Entity	Resize Entity	Perform the task and display event options	location state: 5, 6

Experience 1: FSMs helps describe a typical event-based system such as a UI system. Without using the FSM, one still can describe a UI system design for performing all these tasks. But the system design would probably contain a lot of if-conditions to check whether the tasks are eligible to be executed at a certain system status as we discussed above. This would make it very hard for users to clearly describe how a system works. A bad design would then cause poor usability of the UI implementation. I have seen many software applications explaining the failure reasons in their UIs after performing a user request. This would not only waste users' time, especially for the case of web-based applications, but also confuse users sometimes.

Experience 2: Ideally a FSM should be used to represent a device-independent UI system. Though this FSM is quite abstract and device-independent, I have taken some factors related to mobile devices into account after all. For example, in the Pounamu thick client version, there would be a normal state and the system would come back to the normal state after performing any task, so that the system could be driven to any location state from there to get access to a task state. In future work, we need to study more on designing a real device-independent FSM.

4.4 UI and Interaction Designs of the System States

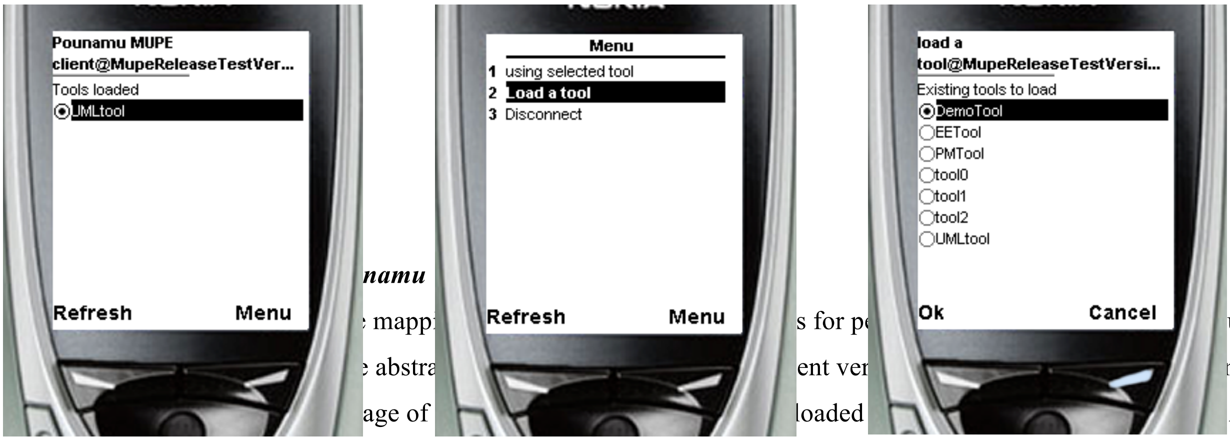
In the section, we present UI and interaction designs of Pounamu/Mobile for each use case (task state) in detail, including contents, layout, and user interactions. We do so by firstly using UI interaction abstracting tables to present the mapping between the UI interaction designs of the Pounamu thick client version and the ones of the abstract device-independent version, then using the use case description tables to present our UI design solutions for the mobile thin-client version, and then demonstrating the UIs generated by the Pounamu/Mobile prototype using two modelling tools defined by the Pounamu: the UML tool and the Project Management Tool.

Approach used for following constraint analysis:

Shown in the following analysis, we design mobile thin client UIs and interactions mainly considering two constraints, the small screen and the only input way – phone key. Certainly there is a brunch of other constraints on mobile phones, such as small memory size, low CPU power, low response time of thin client UI due to limited network bandwidth, and etc. Although we don't really focus on these factors in the thesis, because, we believe, those problems caused by these constraints will be solved with the rapid evolution of hardware and network, we will come back to some of them in the next two chapters where we discuss the usability and system requirement analysis.

To UI designs of each state, if in big screen, all location states could be displayed in one screen. For example in the original thick client running on PCs, all location states except Diagram Pounamu View state are displayed using a left hand tree, and all events handled by a state are displayed in a tree node's right-click menu. For tiny screen, we even may have to slip states to several sub states until each of them could be fitted into a screen. In our cases, we simply map each location state to a page. To task states, some may not need their own pages. Instead, they share pages with the other states. For example the "Move Entity" state shares the pages with the location state "Diagram Pounamu View". As you will see later, the detailed UI design (widgets and layout) of each state page is shown using screen dumps, while we describe interaction designs of tasks relevant to them.

The interaction designs described in the following use cases allow users to perform the same tasks through web-based UIs on mobile phones as they do in the Pounamu thick client and thin client versions on PCs. To achieve this, we don't need to start from scratch, because, based on our experience of using Pounamu, we have already had a clear mind how these tasks are performed through the thick client UIs. We firstly map the thick client UI interactions, such as mouse button click and drag-and-drop, to the abstract interaction designs, then realize them in mobile web-based pages using the facilities supplied on mobile phones, by choosing available widgets to render the data, designing layout of pages in the small screen, and designing user interactions using phone keys. For your convenience, we present our UI and interaction design solutions of the location and task states in use cases in the following.



Pounamu MUPE
client@MupeReleaseTestVer...
Tools loaded
◉ UMLtool

Refresh Menu

Menu
1 using selected tool
2 Load a tool
3 Disconnect

Refresh Menu

load a
tool@MupeReleaseTestVersi...
Existing tools to load
◉ DemoTool
○ EETool
○ PMTool
○ tool0
○ tool1
○ tool2
○ UMLtool

Ok Cancel

(a) (b) (c)

namu

mapp s for p u Tool" among the abstra ent ver mple of the use age of loaded Application state showing all events handled by the state, and (b) is the page of the task state "Load Pounamu Tool" that displays all existing tools can be loaded.

Table 4-2: UI Interaction Abstracting and Use Case Description "Load Pounamu Tool"

	Load a tool to the system from the existing specified tools		
Step	Thick client	Abstract	Mobile thin client
1	mouse right click on the "tool projects" node of the tree	switch the system to the Application state	Load up the page of the Application state
2	popup menu	show all events available in the state	display the menu in a separate page
3	select "Load Tool" by mouse left clicking on the popup menu	trigger the event "Load a tool"	display the menu using a menu list, where menu commands can be selected using up and down direction keys.
4	popup tool selection dialog	switch the system to the state "Load Pounamu Tool", where all tools available for loading are shown.	Load up the page of the task state "Load a Tool", which displays all tools for selection using a radio button group
5	double click on a tool to complete the tool selection	Select a tool	select a tool using direction keys
		Trigger the event and come back to the previous state	Press Ok key to trigger loading command and load up the page of the Application state

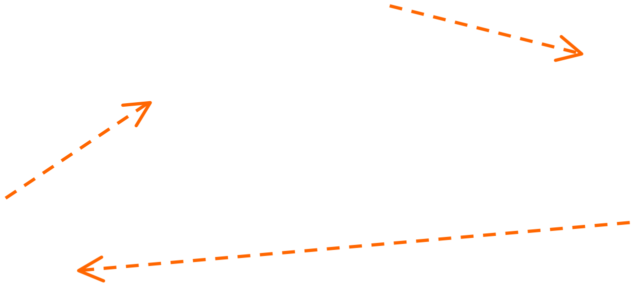
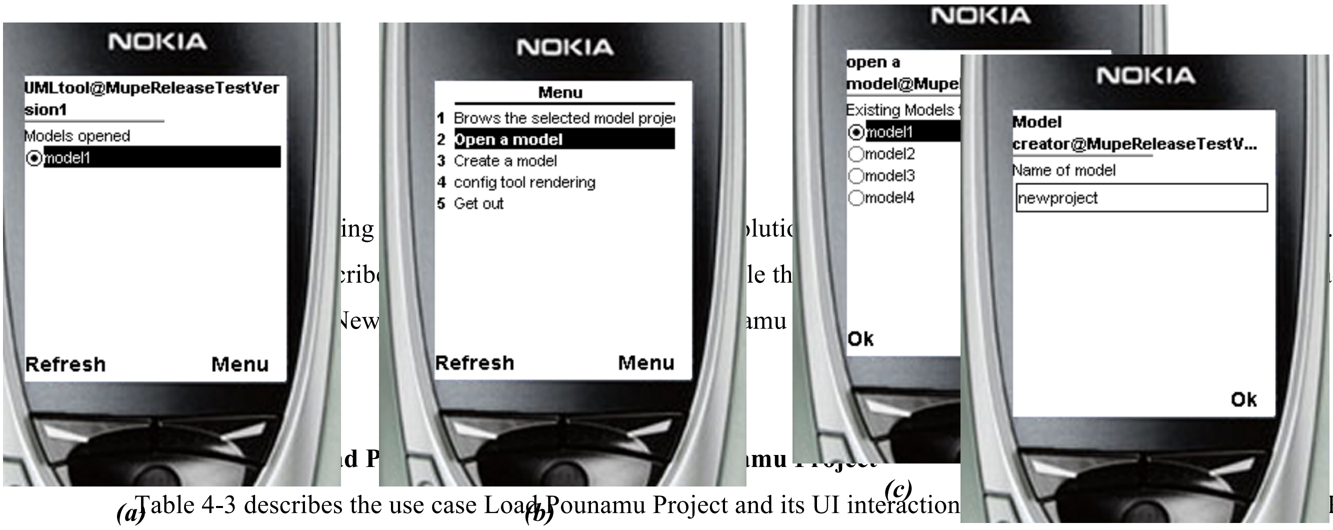


Figure 4-3: UI Examples of the use case "Load Pounamu Tool"



UMLtool@MupeReleaseTestVersion1
Models opened
◉model1
Refresh Menu

Menu
1 Brows the selected model proje
2 Open a model
3 Create a model
4 config tool rendering
5 Get out
Refresh Menu

open a model@Mupe
Existing Models
◉model1
○model2
○model3
○model4
Ok

Model creator@MupeReleaseTestV...
Name of model
newproject
Ok

ing lutio . To save

ribe le th d

New amu

d P mu Project

Table 4-3 describes the use case Load Pounamu Project and its UI interaction Pounamu Project and its UI interaction designs are described in the table 4-4. Figure 4-4 shows an UI example of the two use cases, where (a) is the page of the Pounamu Tool state showing all loaded projects using the tool, (b) is the menu page of the Pounamu Tool state showing all events handled by the state, (c) is the page of the task state "Load a Project" that displays all existing projects defined using the tool, and (d) is the page of the task state "Create a Project" that requires users to input the project name.

Table 4-3: Use Case Description "Load Pounamu Project"

Description	Load a Pounamu Project from existing projects defined using the tool
Step	1. Load up the page of the Pounamu Tool state 2. Display the menu using a menu list in a separate page 3. Select the menu command "Load a Tool" using up and down direction keys. 4. Load up the page of the task state "Load a Project", which use a radio button group displaying all projects defined using the tool 5. Select a project using direction keys 6. Press Ok key to trigger the command and load up the page of the Pounamu Tool state

Table 4-4: Use Case Description "New Pounamu Project"

Description	Create a Pounamu Project
Step	1. Load up the page of the Pounamu Tool state 2. Display the menu using a menu list in a separate page 3. Select the menu command "Create a Model" using up and down direction keys. 4. Load up the page of the task state "Create a Project", where users input the name for the new project 5. Press Ok key to trigger the command and load up the page of the Pounamu Tool state

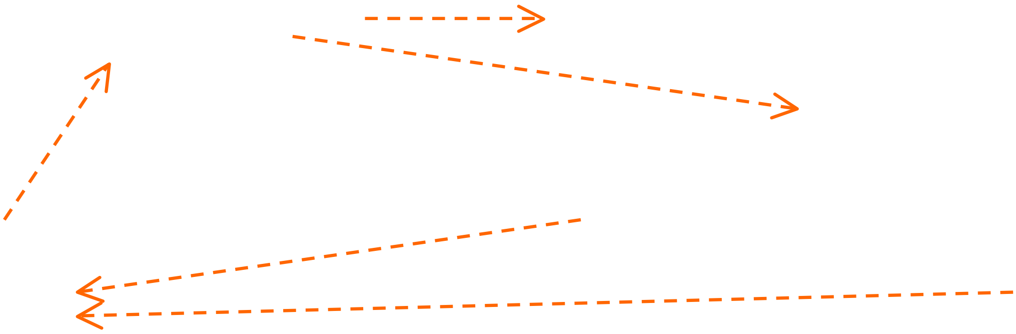
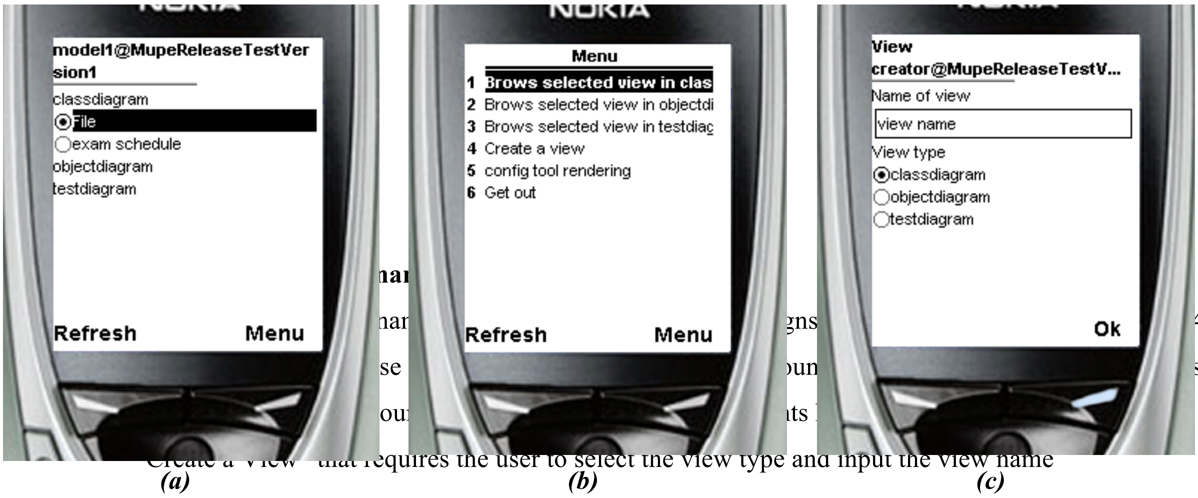


Figure 4-4: UI Examples of the use cases "Load Pounamu Project" and "New Pounamu Project"



model1@MupeReleaseTestVersion1

classdiagram

◉File

○exam schedule

objectdiagram

testdiagram

Refresh Menu

Menu

1 Brows selected view in clas

2 Brows selected view in objectdi

3 Brows selected view in testdiag

4 Create a view

5 config tool rendering

6 Get out

Refresh Menu

View creator@MupeReleaseTestV...

Name of view

view name

View type

◉classdiagram

○objectdiagram

○testdiagram

Ok

nar

nar ... gns ... 4-5. Figure 4-5 shows

se ... un ... s of the project, (b) is

ou ... ts ... of the task state

"Create a View" that requires the user to select the view type and input the view name

(a) *(b)* *(c)*

Table 4-5 Use Case Description "Load Pounamu Project"

Description	Load a Pounamu Project from existing projects defined using the tool
Step	1. Load up the page of the Pounamu Project state 2. Display the menu using a menu list in a separate page 3. Select the menu command "Create a View" using up and down direction keys. 4. Load up the page of the task state "Create a View", which requires users to input the view name and select the view type from a radio button group displaying all view types defined in the tool 5. Input the name and select a type using direction keys 6. Press Ok key to trigger the command and load up the page of the Pounamu Project state

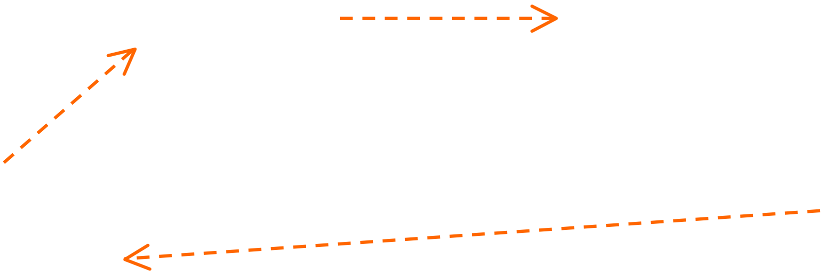
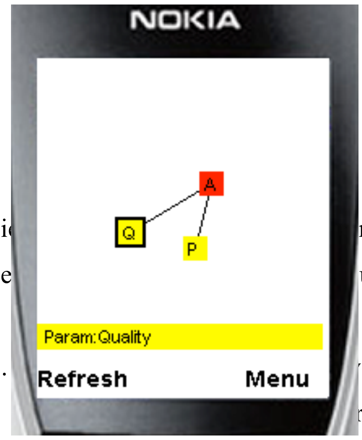


Figure 4-5: UI Examples of the use case "New Pounamu View"

Use Case 5, 6: Load/Refresh Pounamu View

As you may have noticed, for this use case we can't easily get a UI solution by realizing the abstract UI designs using facilities supplied on mobile phones. The basic problem is that data information of an original Pounamu can't be fitted into a single mobile page. A big Pounamu view may contain many elements represented using complex diagram icons defined in Pounamu. One mobile page may be only big enough to display one of these icons.

Although we are able to display a Pounamu view as originally defined in the Pounamu due to the screen size constraints on mobile phones, we still need to satisfy the functional requirement in this stage to allow basically browsing models and performing diagrammatic editing operations. To achieve this for a modeling tool, firstly it is essential to provide users a



Show the type and the name (key) of the selected entity

type of "Param". "Q", the first letter of the element key "Quality", is to differentiate it from other "Param" entities

overvi reen. Secondly, elements are differentiable from their appearances. Finally, details of an e ugh additional steps. We proposed several possible strategies:

1. as an image, in which the whole view is fitted in a single mobile page: Nothing rd to interact with an image without the necessary hardware facilities such as mouse or pen.

2. A part of readable zoomed-out Pounamu View as an image: lack of overview of a big view, and again hard interaction

3. A whole readable Pounamu View rendered using icons different from ones defined in Pounamu, but with many information cut off.

Apparently, as we discussed above, we chose the third one and extended it as the simple UI design solution for the use case Load Pounamu View. The solution is detailed in the table 4-6, and an UI example is shown in the Figure 4-6

Table 4-6: Use Case Description: "Load Pounamu View"

Description	Display a Pounamu View in the page of the state "Diagram Pounamu View"
UI Solution	• Use a square as the shape for all types of entities, • Use unique color to differentiate various entity types, • Use the first letter of an entity key to identify entities with the same type. • Elements are selectable using direction keys. When an element is selected, its outline will be highlighted • A status bar on the bottom of page displays more information of the current selected element, such as element type and key name.

Figure 4-6: UI Example of the original use case Load Pounamu View

User Case 7: Add Entity

Abstract the interaction design: In Pounamu, adding an entity is easily done by a typical mouse interaction drag-and-drop of an entity type from the left hand-side project tree to the diagrammatic view. In contrast, on a mobile phone, the size constraint makes it hard to put tree view and diagrammatic view side by side. Even if listing the element types available for the current Pounamu view at the top of the page, there is no directly intuitive interaction solution equivalent to the drag-and-drop by only using keys. In this use case, abstracting interaction designs is very important and convenient for us to work out what the concrete steps they exactly perform behind the mouse interaction. Table 4-7 shows the abstract interaction steps extracted from the thick client UIs

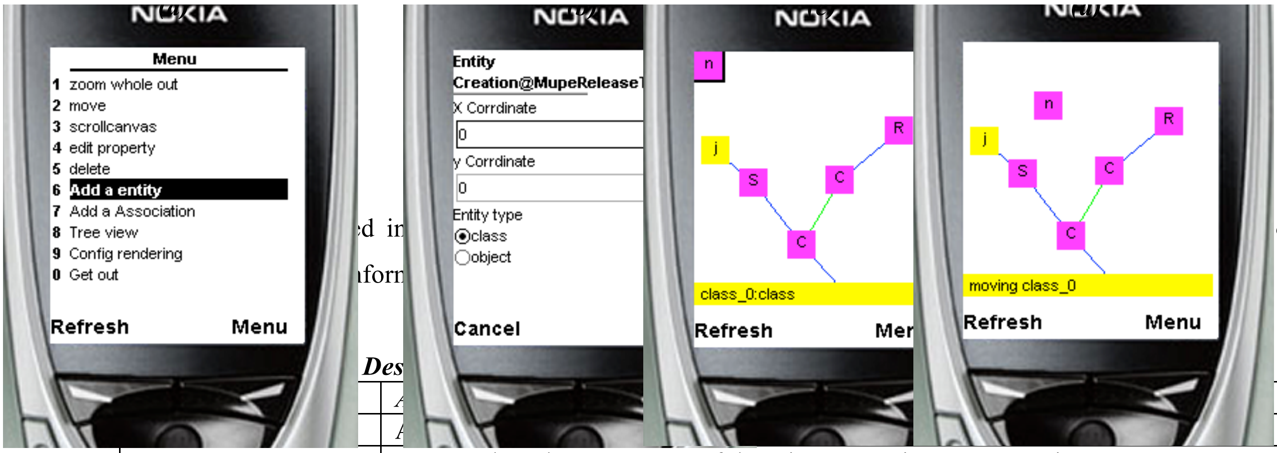
Table 4-7: UI Interaction Abstracting: "Add Entity"

	Add Entity	
Step	Thick client interaction designs	Abstract interaction designs
1	Mouse left button press down on an entity type node	Switch the system to the state Pounamu View (either diagram or tree)
2	Start dragging the clicked entity type node to a diagrammatic view	Switch the system to the state "Add Entity"
		Select an entity type for the new entity
3	Release the left button at a position in the diagrammatic view	Choose the position of the new entity
		Trigger the back-end command to add an entity with the selected type, chosen position, and a default name.
4	Highlight the name of the new created entity	Ask user to input the name of the new entity
5	Mouse clicking on another place indicates that user finishes modification of the name	The event switches the system back to the previous normal state

Experience 3: The state concept helps extract and describe abstract interaction designs. The above is a typical interaction design for window-based thick client UIs. As we can see, many mouse events are generated during the above process. In a complex system such as Pounamu, at runtime the user interactions generate many similar events, but are meant for performing different operations or tasks. For example, the drag-and-drop action, which is identified by a sequence of mouse events, is used to perform many operations in the Pounamu thick client, including creating an entity, creating an association, moving an entity, moving an association, and resizing an entity shape. From this point of view, we really appreciate the benefits that the FSM offers us, especially for describing and implementing an event-based system such as the User Interface system of an application.

Experience 5: Abstracting the interaction designs to a device-independent level helps understanding the real steps that the interactions perform and designing device-specific interactions and UIs to realize them on different devices. As we discussed above, almost all of use cases involved in diagram editing use the same interaction, drag-and-drop, to perform their tasks in the thick client version. In the long term, it is easy for transforming the abstract UI interaction designs to device-dependent specific ones, if rules for mapping are developed.

UI design solution: We use a wizard to perform the same sequence of steps of adding a new entity as detailed in the following table. Wizard is one of easiest ways of guiding users to complete a task, and especially suitable for the case in which it is hard for the user to collect and input all information required to perform a complex task once. With limited



Entity Creation@MupeRelease1

X Corrdinate

0

y Corrdinate

0

Entity type

● class

○ object

d in … out steps while

nforn

Des

Step:	1. Load up the menu page of the Diagrammatic Pounamu View state 2. Select "Add an entity" from the menu list to trigger the event for switching the system to the state "Add Entity" 3. Load up the page of the state "Add Entity", which displays the first page of the task wizard, contains a name edit box, initial position edit boxes, and a radio button group display available entity type in the Pounamu View 4. Input name 5. Input initial position x and y 6. Select entity type using up and down direction keys 7. Click Ok to trigger the back-end command 8. Load up the page of the Diagrammatic Pounamu View state which displays the second page of the task wizard, where the new entity is displayed and highlighted at the initial position as specified in the first wizard page, and the status bar displays a message asking the user to move the new entity to a desired position. 9. The user moves the entity and ends up with pressing the confirm key to complete the task "Add Entity"

Experience 6: In the above, we described a high quality UI interaction design with the task wizard and status bar to realize the abstract interaction design on mobile phones. However, it is hard to directly map the abstract one to our mobile one. In future, there is much left for us to understand about this mapping process and to develop a set of rules for it.

Figure 4-7 shows an UI example of the use case, where (a) is the menu page of the state "Diagrammatic Pounamu View", (b) is the page of the state "Add Entity", which is the first page of the task wizard, (c) is the page of the state "Diagrammatic Pounamu View", which is the second page of the task wizard, and (d) shows the new entity having been moved to a new position.

Figure 4-7: UI Examples of the use case "Add Entity"

Use Case 8: "Move Entity"

Abstract the Interaction Design: In the thick client Pounamu, moving an entity is easily done by a drag-and-drop using a mouse. We abstract this sequence of mouse activities out to a set of device-independent steps as described in the following table

Table 4-9: UI Interaction Abstracting: "Move Entity"

	Move an entity in a diagrammatic view	
Step	Thick client interaction designs	Abstract interaction designs
1	Mouse left button press on an entity shape in the diagrammatic Pounamu View	The triggered event switches the system to the state Diagrammatic Pounamu View
		Select an entity
2	Start dragging the clicked entity shape in the diagrammatic view	The event sequence switches the system to the state "Move Entity"
		Move the selected entity
3	Release the left button at a position in the diagrammatic view	Decide on the new position for the entity
		Trigger the back-end command to update the entity with the new position.
		Switch the system back to the previous normal state

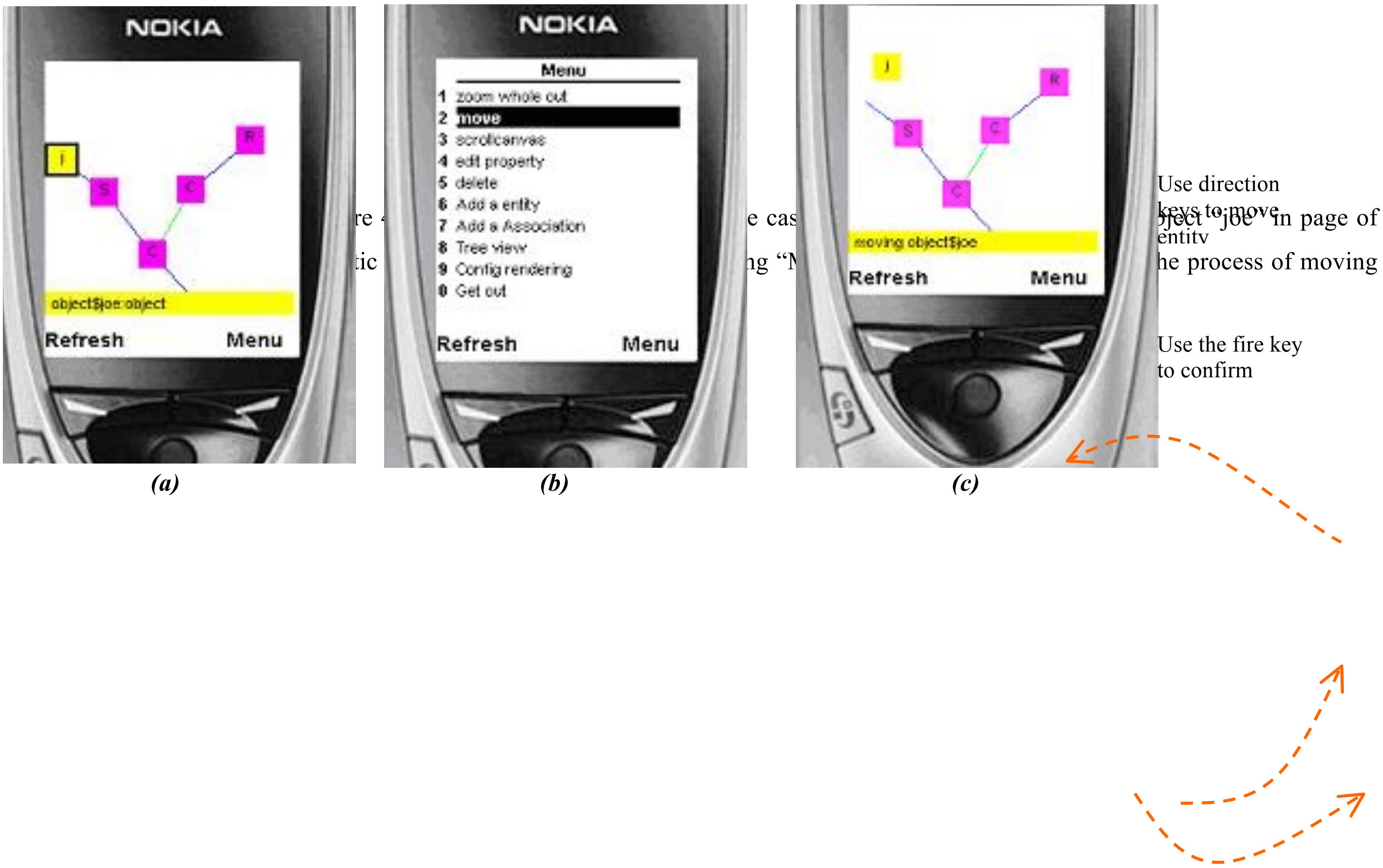
UI design solution: In this abstraction design, most abstract steps, which we have seen and solved in other use cases, can be easily solved here as well. The only problem is how to realize the step of moving a selected entity on mobile phones. Straightforwardly, we chose direction keys to replace a mouse. But the direction key is also used to select elements in a view by moving the highlighter. Therefore, we need to lock the highlighter on the selected entity when the system moves in the "Move Entity" state and release it when the system moves out from the state. The following Table 4-10 describes the interaction design for the mobile thin client version in detail.

Table 4-10: Use Case Description: "Move Entity"

Use Case Name	*Move Entity*
Description:	Move an entity to a new position in a diagrammatic Pounamu view
Step by Step:	1. Load up the page of the state "Diagrammatic Pounamu View" 2. Select an entity shape in the view using up and down direction keys 3. Select "move entity" in the menu list to trigger the event to switch the system to the state "Move Entity". 4. Return to the page of the view. The state "Move Entity" shares the UI page with the state "Diagrammatic Pounamu View", locks the selected entity so that it can be moved using direction keys, and shows status messages for moving a entity in the status bar 5. The status bar firstly shows the message indicating the moving the selected entity, and then it shows another message indicating about pressing the confirmation key to confirm the position, when a user starts moving the entity using direction keys 6. Press the confirmation key to complete the task and switch the system back to the previous state.

re 4 e cas bject "joe" in page of

tic ng "M he process of moving



(a) *(b)* *(c)*

Figure 4-8: UI Examples of the use case "Move Entity"

Use case 9, 10: Add/Move Association:

Abstract Interaction Design: Table 4-11 describes the abstract UI interaction steps of the use case Add Association, which is mapped from the UI designs of the thick client version.

Table 4-11: UI Interaction Abstracting: "Add Association"

	Add an association	
Step	Thick client interaction designs	Abstract interaction designs
1	Mouse left button click on an association type node in a Pounamu View sub tree of the tree	Firstly switch the system to the "Diagram Pounamu View"state
		Then switch the system to the task state "Add Association"
		Select the type for the new association
2	Mouse left button down on a shape handler of an entity in the diagram Pounamu view, and then drag and drop on	Choose the first entity shape and the handler on it
		Choose the second entity shape and the handler on it
3	Release the mouse left button on an handler of an entity shape	Trigger the back-end command with selected type, first shape and handler, second shape and handler, and the default name
4	Highlight the name of the new create association	Ask the user to input the name for the new association
5	Modify the name and end up with a mouse clicking on anywhere	Switch the system back to the previous normal state

UI design solution: Though the use case is performed by the same interaction drag-and-drop as "Move Entity", we can't use the same interaction design to realize them on mobile phones without understanding the tasks they actually represent and try to achieve. Here we feel again the benefits from abstracting interaction designs help us to better understand the detail steps that device-specific UI interaction represent. In this use case, we choose to use a task wizard to realize steps 2-5, which described in detail in the following table. We use the same UI design for the use case "Move Association", and, to save space, we don't repeat it again.

Table 4-12: Use Case Description: "Add Association"

Use Case Name	*Add Association*
Description:	Add an association to in a Diagrammatic Pounamu View
Step by Step:	1. Load up the page of the state "Diagrammatic Pounamu View" 2. Select the "Add association" in the menu list to trigger the event that switches the system to the task state "Add Association" 3. Load up the wizard page of the state "Add Association" 4. Choose the first entity from a radio button group. (Please note that we can allow user to choose the handler on the entity shape at this stage, because the entity shape is not equivalent to the one specified in Pounamu. So the handler information can't be stored in the application server Pounamu.) 5. Choose the second entity from a radio button group 6. Input the name 7. Press OK key to trigger the command and load up the updated diagrammatic Pounamu view to the client

Experience 7: It is hard to automate the transformation from an abstract to a device-dependent UI interaction designs. The above UI designs actually could be done better with other UI designs. For example, use the status bar to guide users to select the first and second shape respectively using direction and confirmation keys. However this helped us realize that as it is not easy to transform an abstract UI design to a high quality device-specific UI design, so it is even harder for a machine to do automatic transformation.

Use Case 11: Remove Element

It is straightforward to work out the interaction designs for this use case. We directly describe our mobile thin UI designs in the Table 4-13. Figure 4-9 shows an UI example of the use case, where (a) the user selects the element to be removed, (b) selects the menu command “delete”, and (c) shows the updated diagram page with the element removed.

Table 4-13: Use Case Description: Remove Element

Use Case Name	*Remove Element*
Description:	Remove an entity or association from a Pounamu View
Step by Step:	1. Load up the page of the state “Diagrammatic Pounamu View” 2. Select an element using direction keys 3. Choose the “delete” in the menu list using the confirmation key 4. Trigger the command and reload the updated page of the Diagrammatic Pounamu View state.

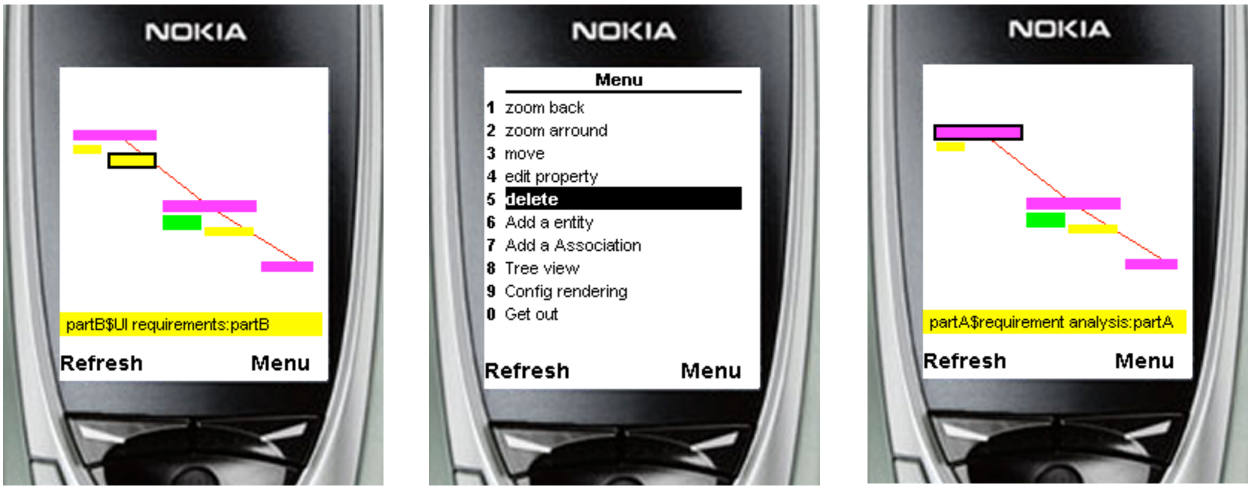


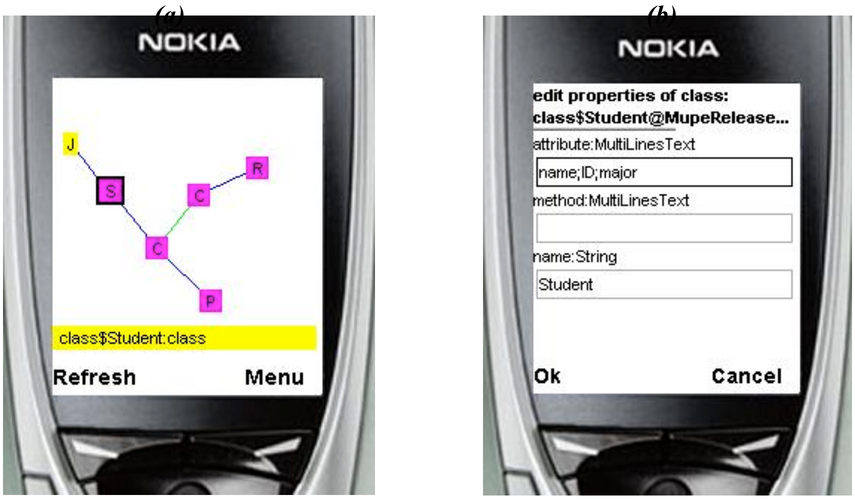
Figure 4-9: UI Examples of the use case “Remove Element”

Use Case 12: Edit Element Properties

It is simple to work out the interaction designs for this use case. We directly describe our mobile thin UI designs in the Table 4-14. Figure 4-10 shows an UI example of the use case, where (a) the user selects the element to edit, (b) the user edits property values in the element’s property sheet.

Table 4-14: Use case Edit Element Properties

Use Case Name	Edit Element Properties
Description:	Edit element properties
Step by Step:	1. Load up the page of the state “Diagrammatic Pounamu View” 2. Select an element using direction keys 3. Choose the “Edit properties” in the menu list using the confirmation key 4. Load up the page of the state “Edit Element Properties”, which displays all properties name and their values. The widgets are chosen to render the properties according to the property type. 5. Change property values and end up with pressing OK key 6. Trigger the command to commit the changes back to server and reload the updated page of the Diagrammatic Pounamu View state.



edit properties of class:
class$Student@MupeRelease...

attribute:MultiLinesText
name;ID;major

method:MultiLinesText

name:String
Student

Figure 4-10: UI Examples of the use case "Edit Element Properties"

Use Case 13: Resize Entity

This is going to conflict with our new zooming facilities presented in the next chapter, since elements could be in different detail levels inside a view. Size has been used as one visual property to represent entity in different detail levels. For instance, an entity in a more detailed level would appear bigger than one in lower detail levels. So in this mobile version of Pounamu, the size of entity doesn't mean its relative size in the original Pounamu View.

4.5 Summary

In this chapter, we presented the Analysis and UI Design work flow of our MUI Development Process with Pounamu/Mobile. We firstly mapped the functional requirements to the system FSM, and then we abstracted the UI interaction design of each task state (use case) from the thick client version to the device-independent version using the system state concept, finally we realized the abstract version, based on the constraint analysis, as the mobile thin-client UI designs, which satisfy all functional requirements and partial usability requirements except the use cases relevant to diagramming. Usage of task wizards and status bar makes complex systems such as visual modelling tools easy to learn and use through the single webpage-based UIs in small screens.

Based on our experience from this chapter, it is evident that using a FSM helps to clearly represent a user's conceptual functional requirements as an event-based system, such as a UI system, and describe the UI interaction steps in detail. Abstracting UI interaction designs helps clearly explain the substantial meanings of UI interaction steps that achieve a particular task as well as realizing them as the concrete device-specific UI interaction steps to achieve the same task.

However as you may see in the above analysis process, it is hard even for the human beings to design high quality UIs for realizing the abstract interaction designs on specific types of devices. We did so by fully understanding the meanings of the abstract interaction designs and constraints of the mobile phones, though much remains to be done in this chapter to improve the usability of the UIs. So as we can see, there is still a lot to be understand about the mapping process of transforming abstract UIs to device-dependent UIs, to either automate or provide support for the process.

Chapter 5 - Usability Analysis and Our Approach

5.1 Introduction

Generally UI and interaction designs from the last chapter support almost all operations supplied in Pounamu thick client and have reasonably quality and usability except for the tasks relevant to diagrammatic browsing. The simple solution presented in the last chapter provides a very low usability, for the end-users, though it allows basic diagrammatic browsing and editing.

In this chapter, we present our innovative approaches, including multi-user runtime tool rendering configuration, multi-level zooming, and a set of navigation approaches, which significantly improve the usability of diagram-based modeling tools and allows users to manipulate with large models effectively though mobile thin-client UIs. In the reminder of the chapter, firstly we present a usability analysis by discussing the existing problems to solve based on the usability requirements and mobile phone constraints. Then we present our solutions to these problems using the new introduced use cases and example usages. Finally we present the completed system state diagrams with the new states added to represent the additional use cases for our approaches.

5.2 Usability Analysis

We present the usability analysis for our current simple solution of diagrammatic browsing and outline the potential problems to be solved to help end-user diagramming effectively.

Readable overview of a diagrammatic Pounamu View: This requirement has been matched in the current UI designs by shrinking entities into a small square so that all contents of a Pounamu view can be fitted in one page displayed in a mobile screen.

Elements differentiable from their appearances: However, fitting a whole Pounamu view inside a small screen by cutting off many details of elements' visual representations potentially causes another serious problem, while applied to this usability requirement. The visual icons are too simple to differentiate represented elements in a complex model. For example, no distinctive difference is evident between two entities with the same type and the same first key letter.

Diagram-based modeling tools closely represent their problem domains: The common solution makes the same appearances for all tools specified by Pounamu. Pounamu is a meta-case tool, which is used to create other visual designing tools by specifying visual notations specific to the problem domains. Using this common simple solution, we lose the ability to tell differences from various tools through mobile UIs.

Diagrams adaptable to users' preference: If we stay with the solution of redefining element representations for the mobile version, then we should allow users to play this role. As in Pounamu, users redefine tool rendering on mobile phone. In addition, unlike single user-based Pounamu, the Pounamu/Mobile server is a multi-user system. Different users may think differently about how the same tool should be rendered on their mobile devices. The same user may have different tool representations specific for his various mobile devices as well, to make the tool more adaptable to different device constraints.

Diagram details easily visible: As we satisfy the requirement of fitting a Pounamu View into a single mobile screen to provide a readable overview, it is impossible to display every detail of an element at the same time. But users still desire to get a deep insight understanding of the model view to help them modeling effectively. Although a property sheet can show the detailed data information using the whole screen, this is not very helpful for end-users from the usability point of view. It is very hard for users to get a deep understanding of the whole model by switching the screen among property sheets of elements, because human beings can't concentrate on too many things once. According to Miller's Law, it is even hard for a person to remember more than seven digits at the same time.

Navigation: The current simple solution doesn't need navigation facilities though. However, in the case that we have to match the above requirements using a potential approach, it naturally has trade-offs between providing an overview to a Pounamu View and displaying diagram in detail. Therefore, this indicates that it is necessary to allow users to locate their positions in a large Pounamu View and quickly switching to any detailed part of a view.

5.3 Our Approaches

In this section, addressing each of these above problems, we present our approaches, which significantly improve the usability of mobile thin-client diagrammatic UIs for Pounamu specified modeling tools. These approaches include multi-user runtime tool rendering configuration, multi-level configurable zooming, button panning, and a floating window for quick zooming and locating. These new facilities introduced new use cases and system states into the system. We describe the UI interaction designs for the new introduced use cases in detail using use case description table and demonstrate the mobile UIs generated by our Pounamu/Mobile server using two example usages: the UML tool and the Project Management tool defined by Pounamu

5.3 1 Multi-user Runtime Tool Rendering Configuration

To better differentiate elements in a Pounamu view, make the diagrams better represent their problem domains, and allow diagrams rendered as users desire, under the natural limitation of screen size, 2D graphic rendering ability, and user interactions of mobile phones, we proposed the multi-user runtime tool configuration facility. This allows users to specify different diagram content configurations via mobile device UIs. User configurations are stored in XML format in Pounamu/Mobile Server. In the runtime, as shown in the figure 5-1, the system gets model data from the Pounamu application, and generates diagram view according to user configurations.

(a) (b)

The original use case “Load Pounamu View” has to be modified to allow rendering elements in a Pounamu View according to the user’s personal configuration. We describe the updated steps in detail for this change in the use case description Table 5-1. Figure 5-1 shows two example UI of the use case Load Pounamu View based on the users’ tool configuration, where (a) shows a class diagram using the UML tool in a small color phone, (b) shows a ganchart chart using the Project Management tool in Nokia 7650.

Table 5-1: Use Case Description: Load Pounamu View (Advanced)

Use Case Name	*Load Pounamu View*
Description:	Rendering diagrams of a Pounamu View according to the element representations redefined by users in the their personal tool configuration file
Step by Step:	7. Load data information of the selected view from Pounamu. 8. Load the user tool configuration. 9. Create entity and association icons according to: (1) data from Pounamu application server, (2) the elements’ visual icon specification in the user’s tool configuration from Pounamu/Mobile server database, and (3) the elements’ current detail levels, zoom levels, which are explained in our next approach multi-level configurable zooming. 10. Calculate the layout of the generated diagram view, according to the elements the spatial relations in the original Pounamu View, zoom levels, and screen size of the requesting device. 11. Generate and return page of the view

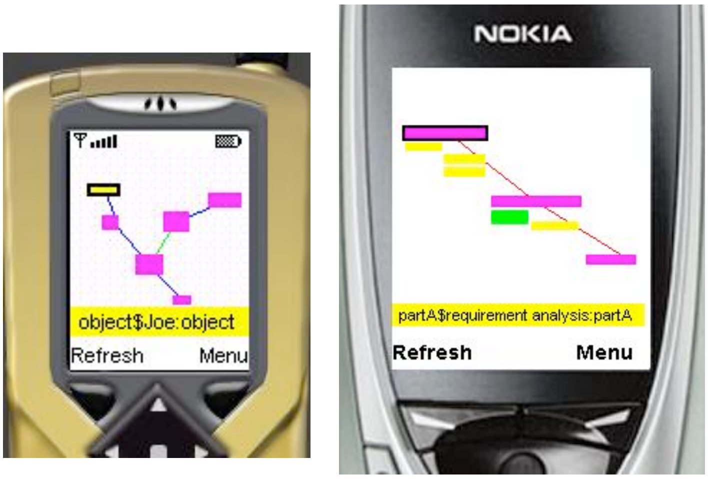
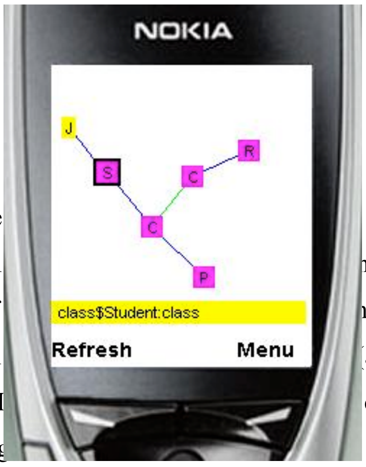


Figure 5-1 UI Examples of the use case “Load Pounamu View”

Apparently this new facility introduces an additional use case “Multi-user Runtime Tool Rendering Configuration” to the system for cofiguring tool representation through mobile UIs at runtime. One disadvantage of single webpage-based UIs is that users have to move multiple levels up or down to get access to a desired functionality, such as setting pages in most of 3D computer games. It apparently reduces the visibility of UIs. To provide an easy access to this facility for the end-users of modeling tools, we proposed multiple entries to it from the location states. Users get different configuration UIs according to the locations where they request configuring tool representation. For example, the mobile page shows the view types can be chosen to configure if a user requests rendering configuration from a “Pounamu Tool” location state. This multi-access to the users’ tool configurations introduced several new task states to represent them in the system state diagram, including the “Tool Rendering Configuation”, “View Type “Element Type Rendering Configuration” state. The detail UI interaction steps of tool he following use case description Table 5-2, and the tool configuration UIs of each of nstrated with the example usage of the UML tool in the Figure 5-2. Figure 5-2 shows an (a) shows a class diagram before configuration (b) shows the multi-level configuration UI class diagram of the UML tool, (c) shows the class diagram displayed after the tool config

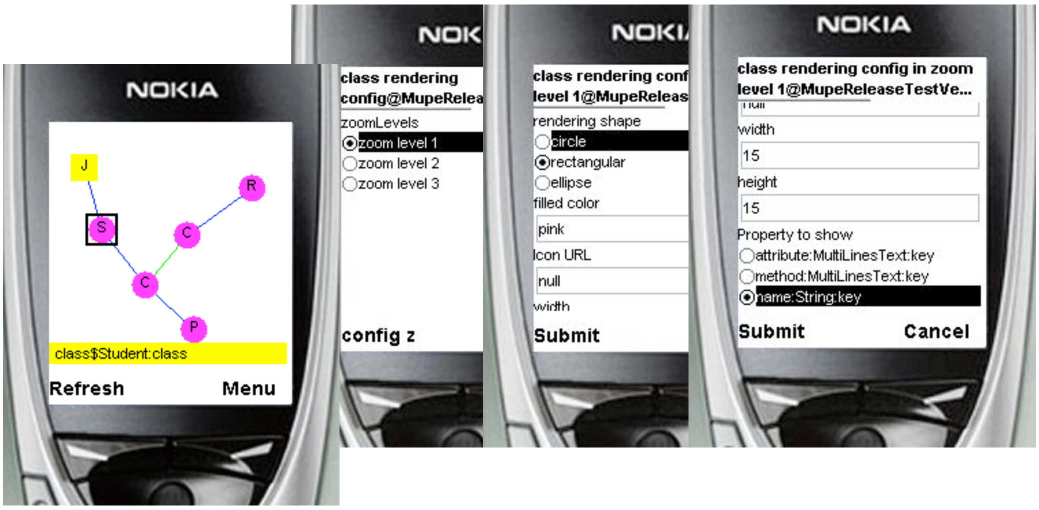


class$Student:class

(a) *(b)* *(c)*

Table 5-2: Use Case Description: Tool Rendering Configuration

Use Case Name	*Tool Rendering Configuration*
Description:	Load and change users’ tool configuration
Step by Step:	1) User can choose menu command of configuration in several level locations, including tool, project, and view 2) The system loads the user’s representation configuration of the current tool and display configuration as a tree structure. But different levels of the configuration tree split to separate windows because of the constraint of screen size. It shows configuration tree level according to the user current location. As describe in the following figure, if user is in tool or project level, then it shows view types to choose for configuring; if user is view level, then it shows element types to configure. 3) As shown in the following figure, user can select an element type and then choose to browse and edit its multiple representations in different detail levels. 4) Based on the mobile phone rendering ability, currently user can specify a diagram representation by using different shapes, colors, and size, and selecting properties to display inside icons. 5) The system store changes back to the user configuration database



class$Student:class

class rendering config@MupeRelea
zoomLevels
● zoom level 1
○ zoom level 2
○ zoom level 3

class rendering conf level 1@MupeReleas
rendering shape
○ circle
● rectangular
○ ellipse
filled color
pink
Icon URL
null
width

class rendering config in zoom level 1@MupeReleaseTestVe...
null
width
15
height
15
Property to show
○ attribute:MultiLinesText:key
○ method:MultiLinesText:key
● name:String:key

Figure 5-2: UI Examples of the use case Tool Rendering Configuration

5.3.2 Multi-Level Configurable Zooming

To allow users to get an in-depth understanding of a Pounamu View and manipulate big model views effectively in small screens, we chose a readable zooming approach. As we mentioned in the Background and Related Work chapter, the existing mobile webpage-based zooming approaches include zoomed-out copies of pages [Wobbrock 02, Milic 02], summarized versions of the page [Buyukkokten 01, Chen, L.Q. 02], page outlines by transforming pages into sets of tiles [Björk 99, Buyukkoten 00, Wobbrock 02], combinations of tiling with zoom-out pages that allow users access individual tiles from an overview [Milic 02, Chen, L.Q. 02, Chen, Y. 03], and collapsing page contents that allows users to zoom into relevant areas by collapsing areas that are less relevant [Baudisch 04]. However, to date, most of these techniques have been limited to text and form-based data and not suitable for richer diagrammatic content.

In our case, although it is impossible to provide both overview and detailed diagrams together, it would very be very useful, if the system could flexibly render and quickly switch between different partial details of an element as users desire. This would dramatically improve the efficiency of diagramming modeling by allowing users to focus on their interested details at different designing stages. To achieve this, we proposed the Multi-level Configurable Zooming approach to provide the facility for users to define multiple representations for a single diagram in different detail levels. While manipulating model views, as shown in the Figure 5-3, each diagram element can be separately selected and zoomed among multiple levels. Automatic zooming of elements is supported while users navigate a large view.

This new facility introduces two additional use cases “Multi-Level Zooming” and “Multi-Level Element Rendering Configuration”. The use case “Multi-level Zooming” introduces a new task state “Multi-level Zooming” to represent the new functionality in the system diagram for zooming a selected element to one of the multiple detail levels. The following use case description Table 5-3 describes the interaction steps involved for performing the task. The use case “Multi-level Element Rendering Configuration” introduces additional UIs for the original task state “Element Type Rendering Configuration” and the UI interactions involved has been discussed previously together with the use case “Multi-User Runtime Tool Rendering Configuration”. Figure 5-3 shows the UI examples of the use case, where the entities are zoomed in various levels for a same gantt-chart view of the Project Management tool.

Table 5-3: Use Case Description: Multi-level Zooming

Use Case Name	*Multi-level Zooming*
Description:	Zoom a selected entity to one of user specified detail levels
Step by Step:	1. User selects an entity in a view under the canvas view, 2. Then select the command menus about zoom, or select hot-keys to zoom the selected entity to different detail levels. Such as level 1 is the simplest and smallest one, and level 2 is the most detailed one. All zooming levels are pre-defined in the user’s tool representation configuration as described in the use case above. 3. After selecting zoom command or hot key, client page is switched to multi-level zooming state and display about going to zoom the selected entity to the desired level and waiting for confirmation from user 4. User confirms the zooming activity by pressing the fire key. 5. The client page receives the command and sends the request back to the server. Before server sends back a returning page, the status bar displays message that the selected entity is being zoomed to the level. 6. The server then sets the flag about level to the entity and regenerate the view page as steps of use case of Display a Pounamu View in Mobile Canvas View

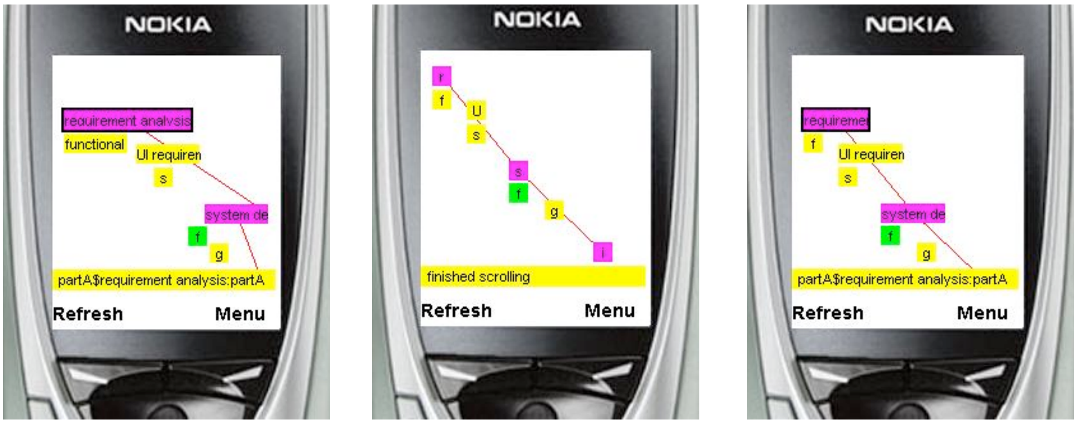


Figure 5-3: UI Examples of the use case Multi-level Zooming

5.3.3 Navigation

We use several navigation approaches to improve user interaction with diagramming tools on mobile devices. Button panning lets users quickly move around in a big diagram by scrolling screens using mobile phone buttons. A floating zooming window allows users easily and quickly to zoom to interesting areas from an overview of a large diagrammatic Pounamu view, as shown in Figure 5-4.

These new facilities introduce two additional use cases: “Locate and Zoom” and “Scroll Screen”, and two new task states to represent them in the system state diagram. These two states share the UI page with the location state “Diagrammatic Pounamu View”. The first one displays an additional floating window on top of a zoom-out diagram view, with the UI interaction steps described in the Table 5-4. Figure 5-4 shows the UI examples from the UML tool with modeling a class diagram, where (a) the class diagram view with all diagrams zoomed in level 1; (b) the same class diagram view with diagrams zoomed in different levels; (c) the floating window indicates the use’s current position in the class diagram view, and the user can be move it and quick zoom to another position. The use case “Scroll Screen” only needs to enable the button panning facility embedded in a page to allow scrolling of a diagram view horizontally and vertically by pressing using the direction keys

Table 5-4: Use Case Description: Locate and Zoom

Use Case Name	*Locate and Zoom*
Description:	Zoom out the whole view, a floating window shows user’s current location in a view, and user can move the window around to zoom to another location
Step by Step:	1. In the canvas view, user selects zooming out command 2. The client page receives and sends the command back to server. 3. The server generates the returning page with the whole view fitted in one screen as our first solution, “Fit a view in one mobile page”. Taking our multi-level zooming facility into account, we can think about this as zoom all elements to level 0, default representation of elements. Element’s default representations are defined, as shown in the following figure 5(c), by taking color and shape from element’s level 1 configuration, and calculating element’s size by zooming out the original Pounamu View proportionately according to the Pounamu View size and Canvas View size on the requesting mobile device. 4. The Server calculates the size and location the floating location window, and display it as shown in the figure 5(c)

(a) (b) (c)

	5. User can move around the floating location window by pressing the direction keys on mobile phones and then zoom into a new location by pressing the confirm button.

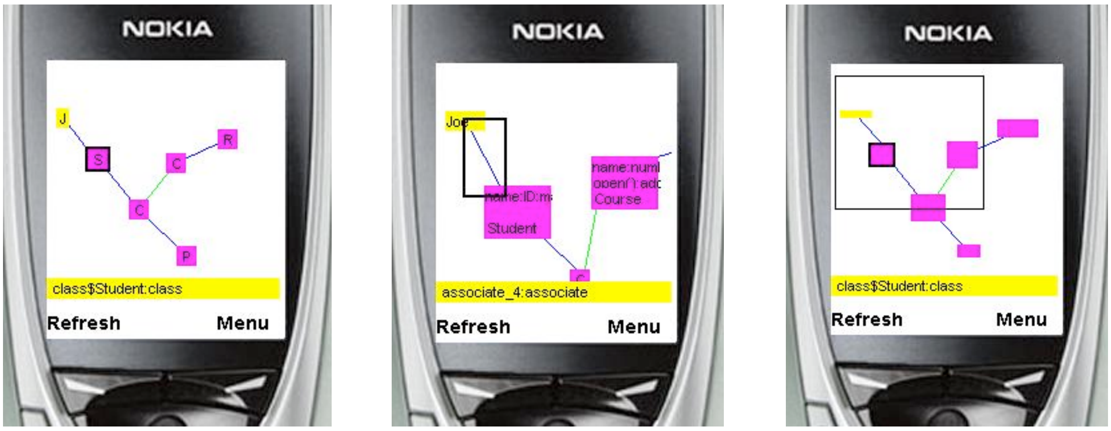


Figure 5-4: UI Examples of the use case "Locate and Zoom"

5.3 The completed System State Diagrams

As may have been noticed, that our approaches introduced many additional task states into the system. In the following, we present the completed System State Diagrams with these new states added in. Figure 5-5 shows the Pounamu/Mobile location state and Configuration state, which is composed of three new task states introduced by the use case "Tool Rendering Configuration". Events between these two big states represent that a user can get the access to different setting level of his tool configuration by requesting from different location states. Figure 5-6 shows the completed View Editing state with three new task states "Multi-level Zooming", "Locate and Zoom", and "Scroll Screen", which are introduced by the three new use cases respectively. It also contains another information, handling annotation that indicates the states need to be handled in client site, in which "C" means the task performed by the state can be handled only in the client side, and "C&S" means the state needs both client and server to complete its task. As you will see, this annotation is useful for extracting classes from the system state diagram in the next chapter.

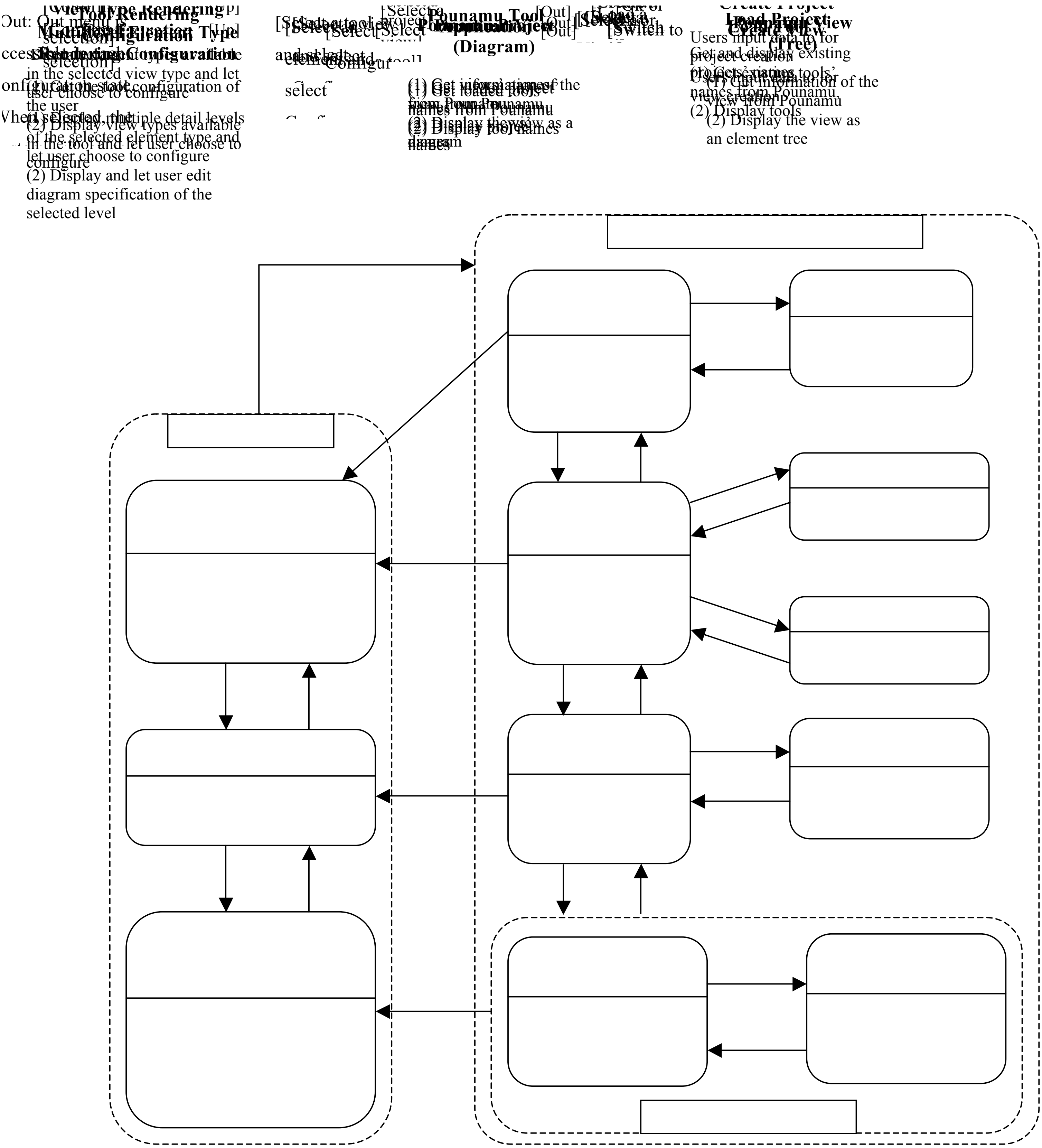

Tool Rendering
Pounamu Tool
(Diagram)
Pounamu View
(Tree)
Create View
Load Project
Create Project
Users input data to for project creation
Get and display existing projects' names
(2) Display tools
(1) Get information of the view from Pounamu
(2) Display the view as an element tree
(1) Get loaded tools names from Pounamu
(2) Display tool names
(2) Display view types available in the tool and let user choose to configure
in the selected view type and let user choose to configure
of the selected element type and let user choose to configure
(2) Display and let user edit diagram specification of the selected level
[Select a tool]
[Switch to

Figure 5-5: System State Diagram: Location State and Configuration State

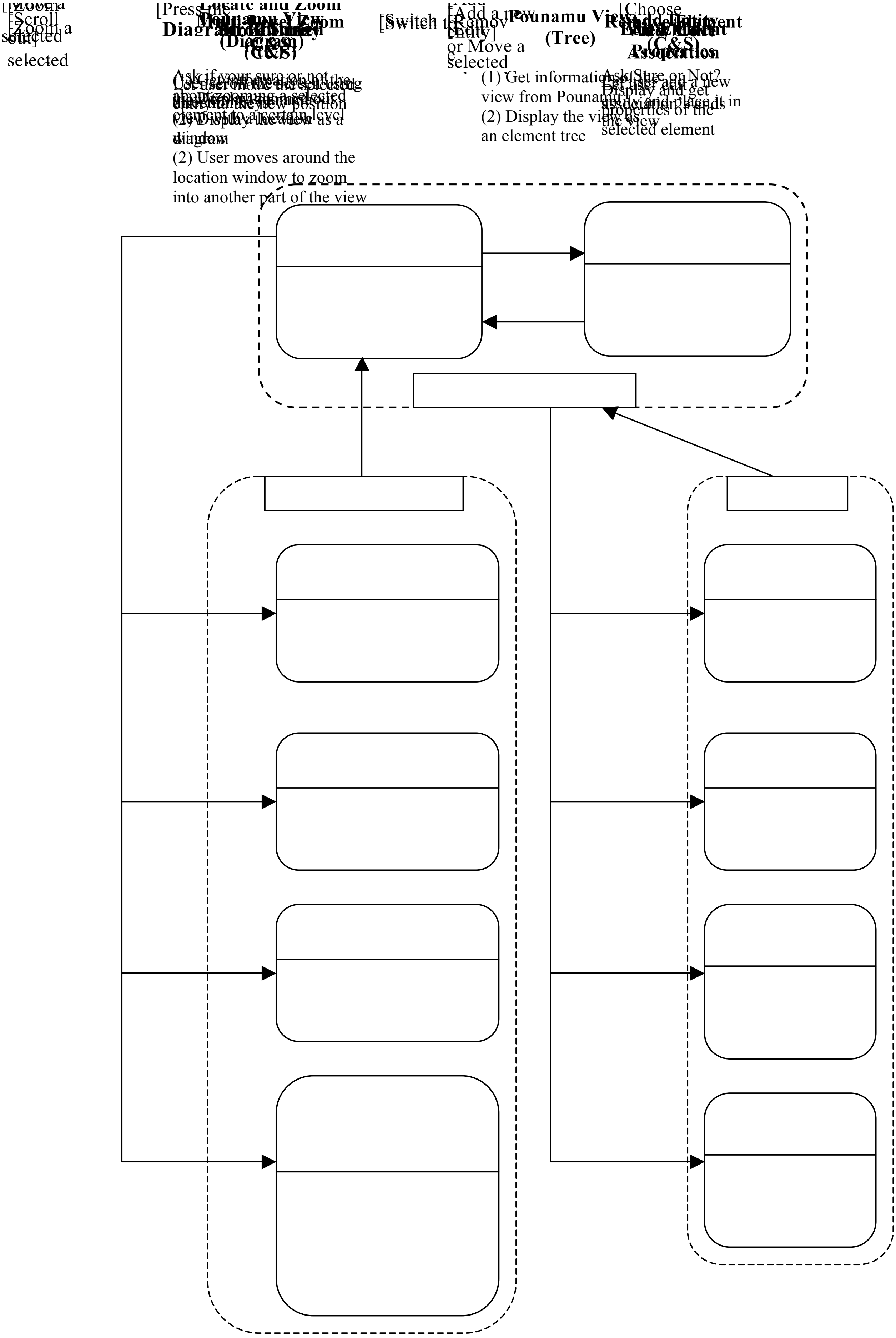


Figure5-6: System State Diagram: Completed View Editing State

5.5 Summary and Discussion

Summary of our contributions

We presented our approach to providing advanced mobile thin-client UIs with high quality and usability for diagramming tools. These include multi-user runtime tool rendering configuration, multi-level configurable zooming, and two navigation facilities. Any Pounamu specified tool can be realized by our Pounamu/Mobile as a mobile thin-client modeling tool with diagrams closely mapped to the targeted problem domain. These new facilities significantly improve the ability of diagrammatic browsing and editing in small screen on mobile devices.

Discussion and future work suggestion

In this chapter, we again experienced the difficulty of supplying UI and interaction design solutions with high quality and usability based on the application domains, diagramming applications for this thesis. We doubt the possibility as a machine able to cognitively analyze usability issues and then propose solutions for realizing abstract UI designs based on specific device constraints, just like what we did in this chapter. However we would like to see whether an intelligent machine could automatically work out the different situations and choose proper solutions pre-proposed in a solution database. Or an advisory development environment support creating high quality multi-device UIs by suggesting, applying, and integrating proper solutions to their UI system.

Chapter 6 – UI Realization and Approach Generalization

6.1 Introduction

In the last two chapters, we described the process of analyzing requirements and device constraints, and designing device specific UIs. The results which have now been gained for developing the Pounamu/Mobile, include the FSM representing the UI system, which includes a set of location states and task states; the device-independent UI and interaction designs for each state; and the mobile thin-client UI and interaction designs for each state.

In this chapter, we describe the process of turning this specific UI design into the concrete implementation. Firstly we present the high-level system architecture design based on the MUPE architecture. Then we describe how to map the states in the system FSM onto objects in our Pounamu/Mobile MUPE server. After extracting the main classes of the system, we present the generalization of our main approaches and discuss the possible reusable artifacts that could be integrated with the MUPE platform, while presenting each use case realization. Finally we present the completed class model of the Pounamu/Mobile system and summarize our main contributions and suggestions to future work from this chapter.

6.2 System Component Architecture

The architecture design is based on the MUPE Architecture (refer to Figure 2-5). As we mentioned in the Chapter 2, we chose MUPE as the implementation and deployment platform, because its XML-based MUPE client script language supports not only the traditional form-based but also canvas-based UI features, which helps us rapidly prototype our ideas and approaches. Also the MUPE client is based on JAVA MIDP, which is enabled on most of latest mobile phones and PDAs. So our mobile thin client of Pounamu can be simply deployed on a wide range of mobile devices.

Our Pounamu/Mobile is a set of components hosted on the customized MUPE server. These components work together to provide diagram view rendering and manipulation, diagram shape and connector property editing facilities, and multi-user tool configuring facilities. Figure 6-1 shows the component architecture of our Pounamu/Mobile system. The diagramming tool runs on the Pounamu Host and behaves as the application server/database manager. It supplies data information to the Pounamu/Mobile server, including tool specifications, models, and multiple views of models. The Pounamu/Mobile server generates mobile thin-client UIs for the diagramming tool at runtime according to these data information and user configurations. The generated UIs are rendered in the MUPE Client browser running on end-users' mobile devices.

The Pounamu/Mobile contains a set of components including Location, Task Handler, Element Property Editing Render, Diagram Visual Property Configuring Render, and User's Tool Rendering Configuration Database. The Location component represents all the location states in the system state diagram. It is responsible for generating UIs for each location state, such as the UI of the Diagrammatic Pounamu View state for diagram rendering and manipulation, and handling all user requests, which are events generated from the location states. The Task Handler component represents the task states and is responsible for generating UIs for users to perform each task. It is also in charge of all communication between the Pounamu/Mobile server and the Pounamu Host (Pounamu Application Server). The Element Property Editing Render is in charge of rendering various types of element properties for generating the element property sheet. The Diagram Visual Property Configuring Render is in charge of rendering various types of diagram visual properties for generating the element-rendering configuration UIs. User's Tool Rendering Configuration Database then stores all users' tool configuration XML files.

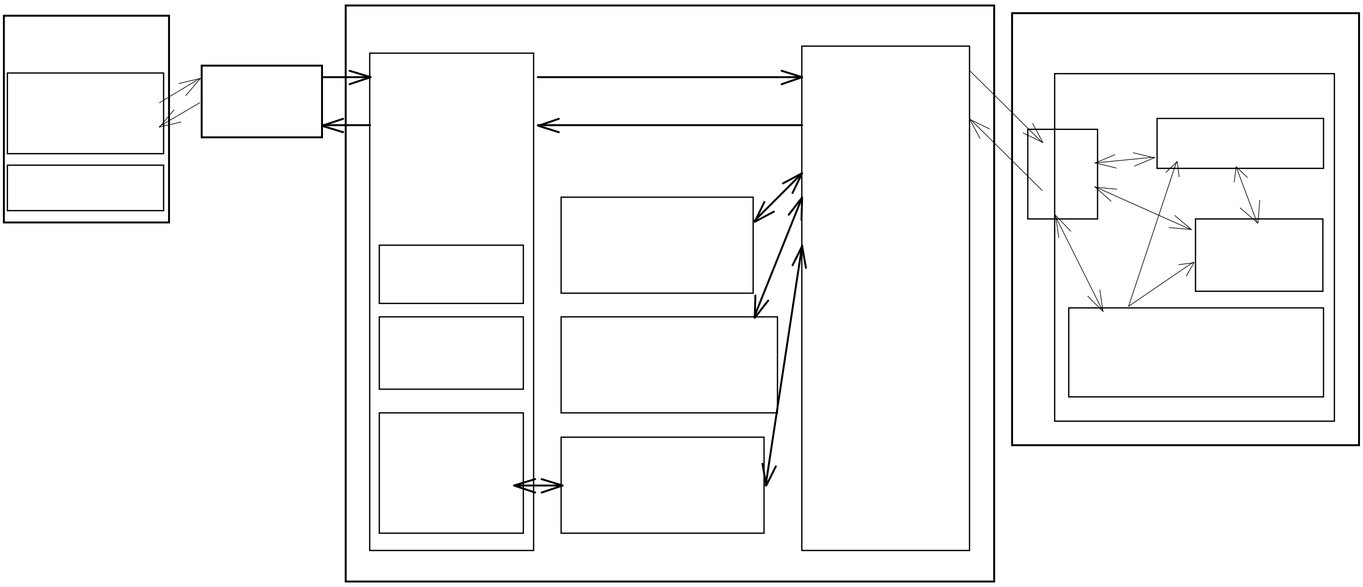

User devices
MUPE Client Browser
JAVA MIDP
XML
MUPE Core
XML
Pounamu/Mobile MUPE Server
(responsible for handling all client requests and generating UIs for the location states)
Pounamu Tool View (diagram and tree view)
Pounamu Project
Element Property Editing Render
Diagram Visual Property Configuring Render
Users' tool rendering configuration database (XML)
(resposible for performing all tasks and generating task wizards, such as add, move, and remove an element; edit element properties; multi-level zooming; tool rendering configuration …)
XML
Pounamu Host
Pounamu Tool
RMI API
Model views
Projects using tools
Modeling Tool Specs (such as UML tool, project management tool.)

Figure 6-1 Architecture of Pounamu Mobile Thin CLient

6.3 Class Extraction

The object-oriented analysis and design of the system is based on object architecture of the MUPE Server. So before mapping the states in the system state diagram to objects, we introduce the primary objects in the MUPE server. Then we describe the Class Extraction from the location states and task states respectively.

6.3.1 Primary Objects in MUPE Server

A typical MUPE application is composed of three basic building blocks: rooms, users and items. The room object is a container that stores other MUPE objects inside it. The user object is an instance of the end-user in the application and an item is a single piece of information in the application. The users move in the rooms and interact with other users, rooms, and different items in a room. For the details of how an MUPE application works, please refer to the MUPE section in the Chapter Background Knowledge and Related Work.

The Figure 6-2 shows the basic class model of the MUPE server. Base class contains three main attributes: name, ID, and contents, and a primary function getDescription. ID is the object ID, which is unique and used to identify an object. Contents contain a set of other Base objects. The function getDescription is to get UI description of the object in the file description.xml. Other three classes all derive from Bass class. The User object contains another field to store user's password. The Room object has another two main operations, which are add and remove. They make adding and ase objects into its cont Item object can't co

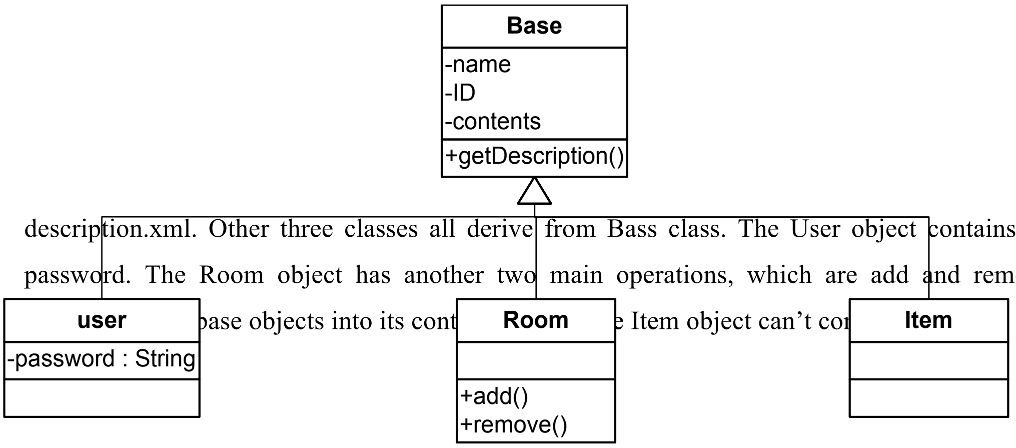


Figure 6-2: Class Diagram: Primary Object Model of MUPE Server

6.3.2 Class Extraction from States

In this section, we will see how to use objects to represent states, which we designed during the analysis and UI design workflow in our MUI Development Process. As you have known, the FSM is composed of two types of states, location and task state. Location states include the Application, Pounamu Tool, Pounamu Project, and Pounamu View state. The rest are task states which are mapped from the system tasks (use cases). In the following, we perform OOA for these Location States and Task States respectively

Map Location States to Classes:

As described early, a location state is responsible for visualizing its contents and handling events generated in it. Then how to visualize the location states and handle their events using objects. In MUPE, a Room object accordingly can be used to realize a location state. Visual representation of a state can be described in the file description.xml using MUPE Client script language in XML. For example, a Diagram Pounamu View state is mapped to the View class that derives from the Room class. Data contents of a Pounamu View include entities and associations. Based on the MUPE data visualization mechanism, it is convenient to use the visual description XML file attached to each object. So we defined the Shape and Connector class derived from the Item Class, which are responsible for rendering entities and associations.

All events defined in a state can be simply mapped to event handling functions defined in the corresponding class. At runtime, the MUPE Server provides the mechanism to handle a client request by calling the corresponding function defined in the object. For example consider a user in a Pounamu View state, the user request of Add Entity invokes the function cmdAddEntity( ) in the View object. However, to the events, which can be handled only in the client side, such as the event Scroll Window in the Diagram Pounamu View state, there is no need to generate event-handling functions for them.

The Figure 6-3 shows the initial class model of the system, which is composed of only the objects mapped from the location states. The classes and their event handling functions are described in detail in Table 6-1. Note: all events about [illegible] he system into [illegible] are handled directly b[illegible] function of the global World object in the [illegible]er.

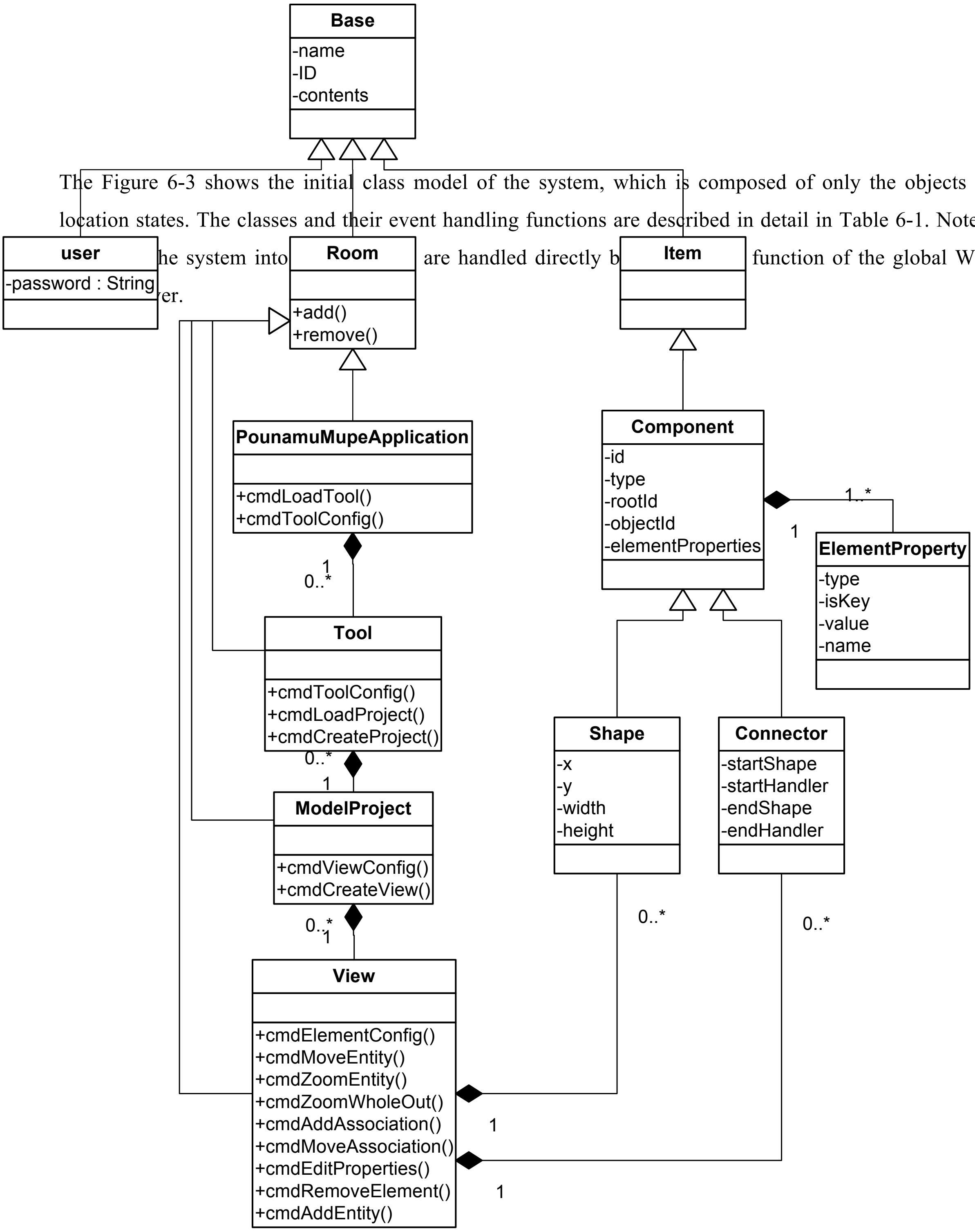


Figure 6-3: Initial Class Model of the Pounamu MUPE

Table 6-1: Class Description: Location Classes

Object	State	Event handing functions function name (event name)
PounamuMupeApplication	Application	cmdLoadTool handles: "load a tool" cmdToolConfig: "tool configuring"
Tool	Pounamu Tool	cmdToolConfig: "configure Tool Rendering", cmdLoadProject: "Load a Project", cmdCreateProject: "Create a Project"
ModelProject	Pounamu Project	cmdConfigView: "Configure View Rendering", cmdCreateProject: "Create a View"
View	Pounamu View	cmdElementConfig: "configure element rendering" cmdAddEntity: "add an entity" cmdAddAssociation: "add an association" cmdMoveEntity: "move the selected entity" cmdMoveAssociation: "move the selected association" cmdEditProperties: "edit the selected element properties" cmdRemoveElement: "remove the selected element" cmdZoomEntity(level): "zoom the selected entity to level.." cmdZoomWholeOut: "Locate and Zoom"
Component		The base class of shapes and connectors and contains a set of ElementProperty objects
Shape		Derives from the Component class and is responsible for rendering an entity
Connector		Derives from the Component class and is responsible for rendering a association
ElementProperty		Represents a property of an element. It contains property name, type, and value.

Class Extraction of Task States:

Task states are responsible for generating task wizard UIs for end-users to complete the tasks. In the Pounamu/Mobile server, it is also in charge of sending commands back to and getting updated data from the Pounamu Host. We defined the base class, called TaskHandler, which derives from the Room class. Similar solution to the location states, we map each task state to a class derived from the base class TaskHandler. To the task states, which don't need its own UI and communication with the Pounamu Application Server for completing their task, we simply handle them in the corresponding event handling functions defined in the location states. This kind of task states includes Multi-level Zoom Element and Locate and Zoom.

Figure 6-4 shows all task handler Class mapped from the task states and the event handling functions mapped from their events. This is concerning which the mapping for the System State Diagrams to the Class Diagram is quite simple and clear. However there two main objects, we need to give the detailed explanation in the following table 6-2:

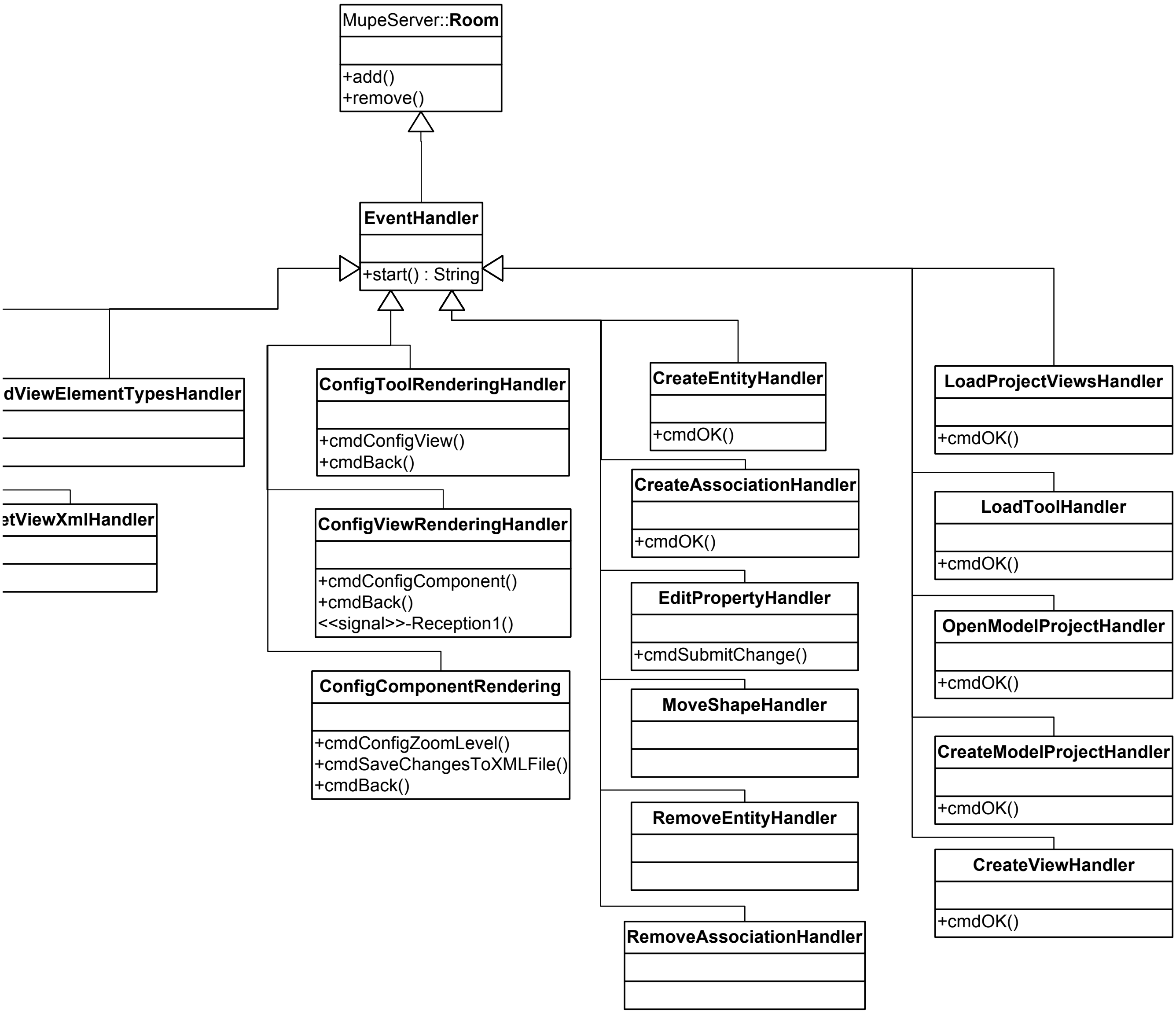


Figure 6-4: Class Model of Task States

Table: 6-2 Class Description: TaskHandler Classes

Class Name	Description
TaskHandler	An abstract class, which derives from the class Room. It is the base class of all task handler classes. Virtual function start( ) is to be defined by the deriving task handler class and invoked by the global TaskHandlerManager object to start the task.
TaskHandlerManager	The interface between the location objects (User, Room, and Item objects) and the task handler objects. It is in charge of organizing and creating corresponding task handler objects to complete a particular task.

Reusable Artifacts

As we have seen, in this section, the class extraction can be quite simply and clearly done by mapping states from a System State Diagram to objects in a customized MUPE Server. The rules are described as follows:

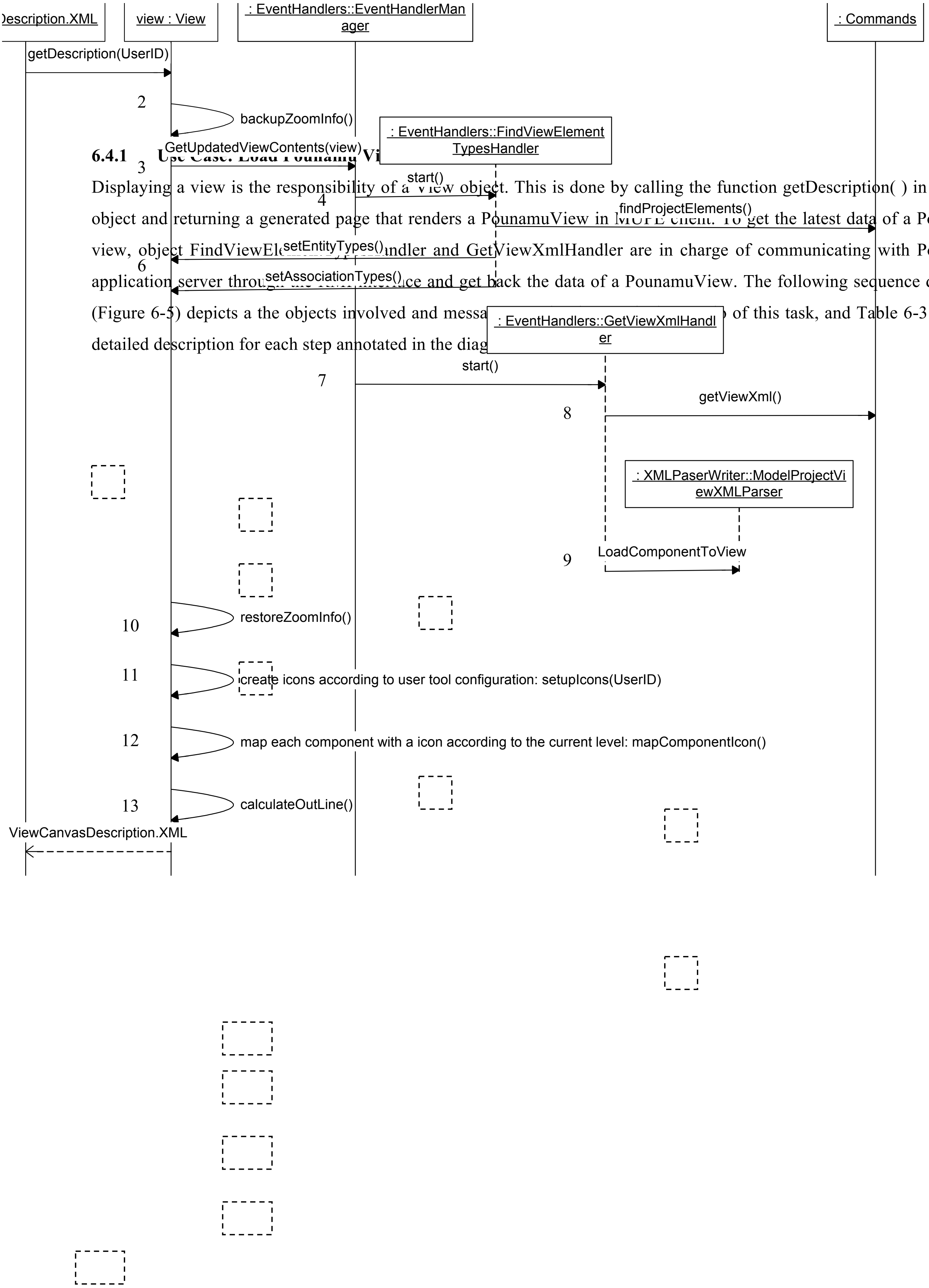
- Map a location state to a class that derives from the Room class and
- Map a task state to a class that derives from the, if the task state needs its own UI or complex process to perform the task, such as communicating with the application server or database.
- Define an event handling function in the class for each event generated in the state.
- Specify the UI in the description.xml file attached to the class.

6.4 User Case Realization and Approach Generalization

In this section, we go through several main use cases relevant to diagrammatic browsing and editing, including our approaches that we presented in the last chapter, and see how these use cases are realized using objects that we extracted from the last section. At the end of each use case, we will discuss the possible reusable artefacts including the generalization of our approaches, for building other similar mobile thin-client applications on the MUPE platform.

6.4.1 U[illegible] Pounamu Vi[illegible]

Displaying a view is the responsibility of a view object. This is done by calling the function getDescription( ) in a View object and returning a generated page that renders a PounamuView in MUPE client. To get the latest data of a Pounamu view, object FindViewEl[illegible]Types[illegible]andler and GetViewXmlHandler are in charge of communicating with Pounamu application server throu[illegible]ce and get back the data of a PounamuView. The following sequence diagram (Figure 6-5) depicts a the objects involved and messa[illegible] of this task, and Table 6-3 gives a detailed description for each step annotated in the diag[illegible]



)escription.XML
view : View
: EventHandlers::EventHandlerManager
: Commands
getDescription(UserID)
2
backupZoomInfo()
: EventHandlers::FindViewElementTypesHandler
GetUpdatedViewContents(view)
3
start()
4
findProjectElements()
setEntityTypes()
6
setAssociationTypes()
: EventHandlers::GetViewXmlHandler
start()
7
getViewXml()
8
: XMLPaserWriter::ModelProjectViewXMLParser
9
LoadComponentToView
10
restoreZoomInfo()
11
create icons according to user tool configuration: setupIcons(UserID)
12
map each component with a icon according to the current level: mapComponentIcon()
13
calculateOutLine()
ViewCanvasDescription.XML

Figure 6-5: Sequence Diagram of the use case – Load Pounamu View

Table 6-3: Use Case Event Description: Load Pounamu View

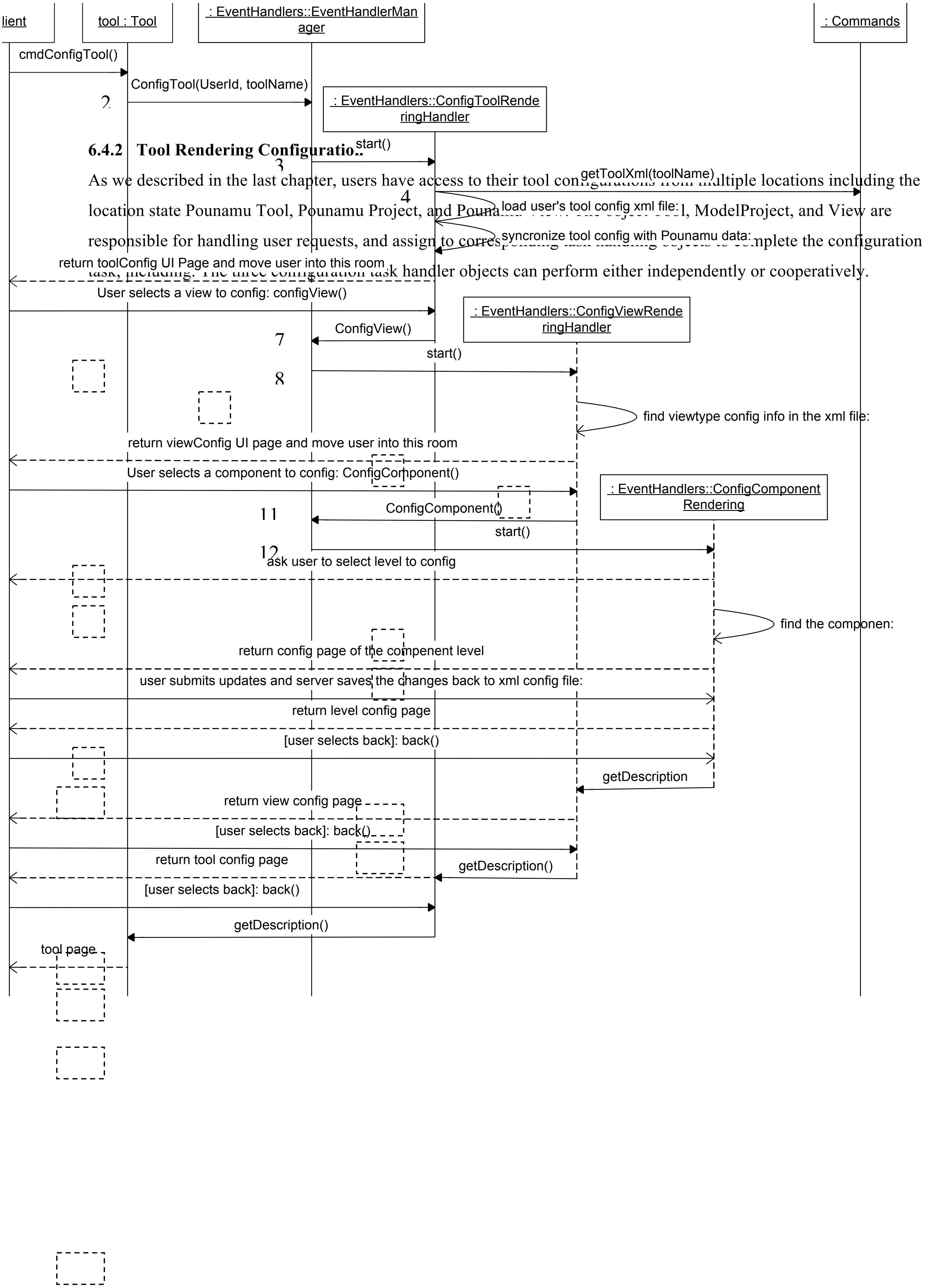
Step	Description
1	Function getDescription( ) of a corresponding View object is invoked when a user asks for browsing a selected View in a Project page.
2	The activated View object is in charge of updating and displaying its contents (entity and association objects). First it needs to backup all components' current zoom levels.
3	Invoked by step 1, to get the latest view data from Pounamu application
4	TaskHandlerManager creates and launches a FindViewElementTypesHandler to get available element types in the view type
5	Invoked by step 4, to get new element types through RMI
6	a Store entity types in the view object
	b Store association types in the view object
7	Create and launch another task handler object to get latest view contents
8	Invoked by step 7, to get the view data in XML through RMI
9	Create a ModelProjectViewXMLParser object to load contents of the view to the view object to replace the old Shape and Connector objects
10	Now back to the view object, restore previous zoom level information to the new component objects
11	Retrieve tool configuration of the current user and create a rendering icon object for each element type in the view
12	Map each Component object in the view with a rendering icon object according to its element type
13	Calculate layout of Shapes in various zoom levels according to their spatial relationships
14	Render the view contents by mapping data of each element to its rendering icon and place it in the canvas according to the result of layout calculation. Finally return the generate view page back to client on the user's device.

Reusable Artifacts

- Classes involved in Steps 1-9 could be automatically generated by a tool as we have discussed in the last section
- Mechanism of Step 10-14 could be integrated into MUPE Server to allow other MUPE applications to render their Items, Users, and Rooms according to the users' configurations and use our multi-level zooming facility. We discuss the approach generalization of multi-user runtime UI configuration and multi-level zooming in the next two Use Cases.

6.4.2 Tool Rendering Configuration

As we described in the last chapter, users have access to their tool configurations from multiple locations including the location state Pounamu Tool, Pounamu Project, and Pounamu View. The object Tool, ModelProject, and View are responsible for handling user requests, and assign to corresponding task handling objects to complete the configuration task, including: The three configuration task handler objects can perform either independently or cooperatively.



lient
tool : Tool
: EventHandlers::EventHandlerManager
: Commands
cmdConfigTool()
ConfigTool(UserId, toolName)
2
: EventHandlers::ConfigToolRenderingHandler
start()
3
getToolXml(toolName)
4
load user's tool config xml file:
syncronize tool config with Pounamu data:
return toolConfig UI Page and move user into this room
User selects a view to config: configView()
: EventHandlers::ConfigViewRenderingHandler
ConfigView()
7
start()
8
find viewtype config info in the xml file:
return viewConfig UI page and move user into this room
User selects a component to config: ConfigComponent()
: EventHandlers::ConfigComponentRendering
ConfigComponent()
11
start()
12
ask user to select level to config
find the componen:
return config page of the compenent level
user submits updates and server saves the changes back to xml config file:
return level config page
[user selects back]: back()
getDescription
return view config page
[user selects back]: back()
return tool config page
getDescription()
[user selects back]: back()
getDescription()
tool page

Figure 6-6: Sequence Diagram of the use case: Tool Rendering Configuration

Table 6-4: Use Case Event Description: Tool Rendering Configuration

Steps	Description
1	Invoked when an user requests access his/her tool configuration from a Tool state
2	Ask TaskHandlerManager to handle the particular task
3	ConfigToolRenderingHandler is created and launched to perform the task.
4	Retrieve and load the user's tool configuration from user tool configuration XML DB, using the user ID and the tool name as keys. Synchronize the extracted user tool configuration file, with latest tool information from Pounamu Application.
5	Generate and return description page of Configure Tool Rendering
6	Invoked when user asks to configure a selected view type
7	Ask TaskHandlerManager to handle the particular task
8	ConfigViewRenderingHandler is created and launched to perform the task.
9	Load all element type names in the selected view type and return ConfigView page
10	Invoked when user asks to configure a selected element type
11	Ask TaskHandlerManager to handle the particular task
12	ConfigComponentRenderingHandler is created and launched to perform the task.
13	Create and return configuration page of the element type in the selected zoom level
14	Invoke when user finishes configuring and submits changes. The function stores updates back to the configuration file and return ConfigComponentLevel Page
15	Function cmdBack is invoked when user asks to return back to the previous configuration level. Here it returns ConfigView state by invoking getDescription() function of the object ConfigViewRenderingHandler
16	Finally the user returns back to the location state where he came from. Here he comes back to the Tool state by invoking getDescription function of the Tool Object

Approach Generalization:

Multi-user Runtime Configurable UI Configuration: generalizes our approach multi-user runtime tool-rendering configuration to a two-level runtime UI configuration. In the following we give our solution, which could be integrated in the MUPE server in the future. At runtime, as the first level, end-users specify which features of data visual representation are configurable and define the value range of each configurable feature. As the second level, end-users configure the UIs of a MUPE application based on the configurable visual features they have specified. Both level configurations are done though the MUPE client.

In the MUPE server, each Room and Item object has its own multi-level description files, a configuration file, and a configuration editor file in XML. A description file exists in the original MUPE server to describe the UI of a room or item object. Here we extend it as multiple ones to describe the object UIs in different detail levels. This idea supports not only multi-level zooming, but also displays objects in different sizes of screens. A configuration file lists all visual properties of an object with the configurable tag, value range, and values of each zoom level. A configuration editor file is used for runtime configurations by the end-users through the MUPE client. At the runtime, to render an item/room object, the server firstly determines its detail level, and then loads its configuration of the level, and finally fills the user specified property values into the object visual description.

But the above solution is only suitable for a single user application, which means all users share the same UI configuration of an application. To make it feasible for multi-user, just like what we did for the Pounamu/Mobile, an MUPE application stores a UI configuration file for each user. Each user configuration file stores visual configuration for all item/room objects of the application.

6.4.3 *Use Case: Multi-level Zoom Element*

The View object is responsible for handling multi-level [illegible] formed in the UI page of the state Diagram Pounamu View, and doesn't require updated data from the Pounamu application. Therefore the task can be easily handled by an event handling function in the View object instead of using a request handler object. The following sequence diagram (Figure 6-7) depicts the objects involved and messages passing in a typical scenario of this task, and Table 6-5 gives a [illegible] each step annotated in the diagram.

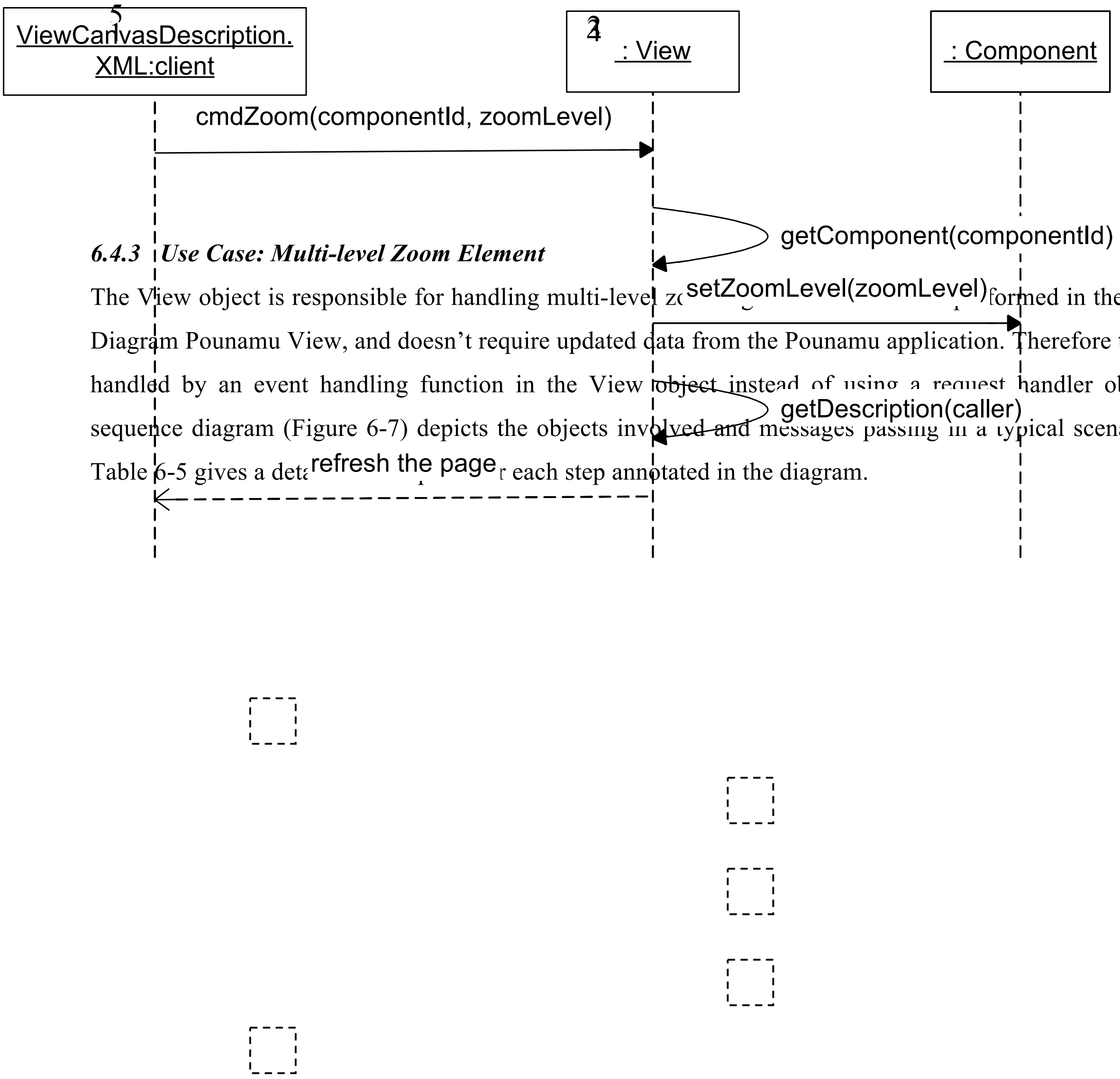


Figure 6-7: Sequence Diagram of the use case: Multi-level Zoom Element

Table 6-5: Use Case Event Description: Multi-level Zoom Element

Steps	Description
1	Invoked when user requests to zoom an entity to a predefined detail level, here say level 2
2	Invoked by step 1, to get the component according to the argument parameterID sent from client
3	Set the zoom level of the entity to level 2
4	Refresh the view by invoking the function getDescription of the view object
5	Return the new generated View Page

Reusable Artefacts:

As discussed in the last two use cases, the mechanism that allows multi-zooming has been integrated in the MUPE server. To future work, the facility could be wrapped in the MUPE client script language and enabled in the MUPE client.

6.4.4 *Use Case: Locate and Zoom*

Similar to the multi-level zooming, this task is handled only using an event handling function in the View object. Instead of rendering elements according to their zoom levels, all elements are displayed, so that a whole Pounamu View can be fitted inside a screen. sequence diagram (Figure 6-8) depicts the objects involved and messages passing in a typical scenario o d Table 6-6 gives a detailed description for each step annotated in the diagram.

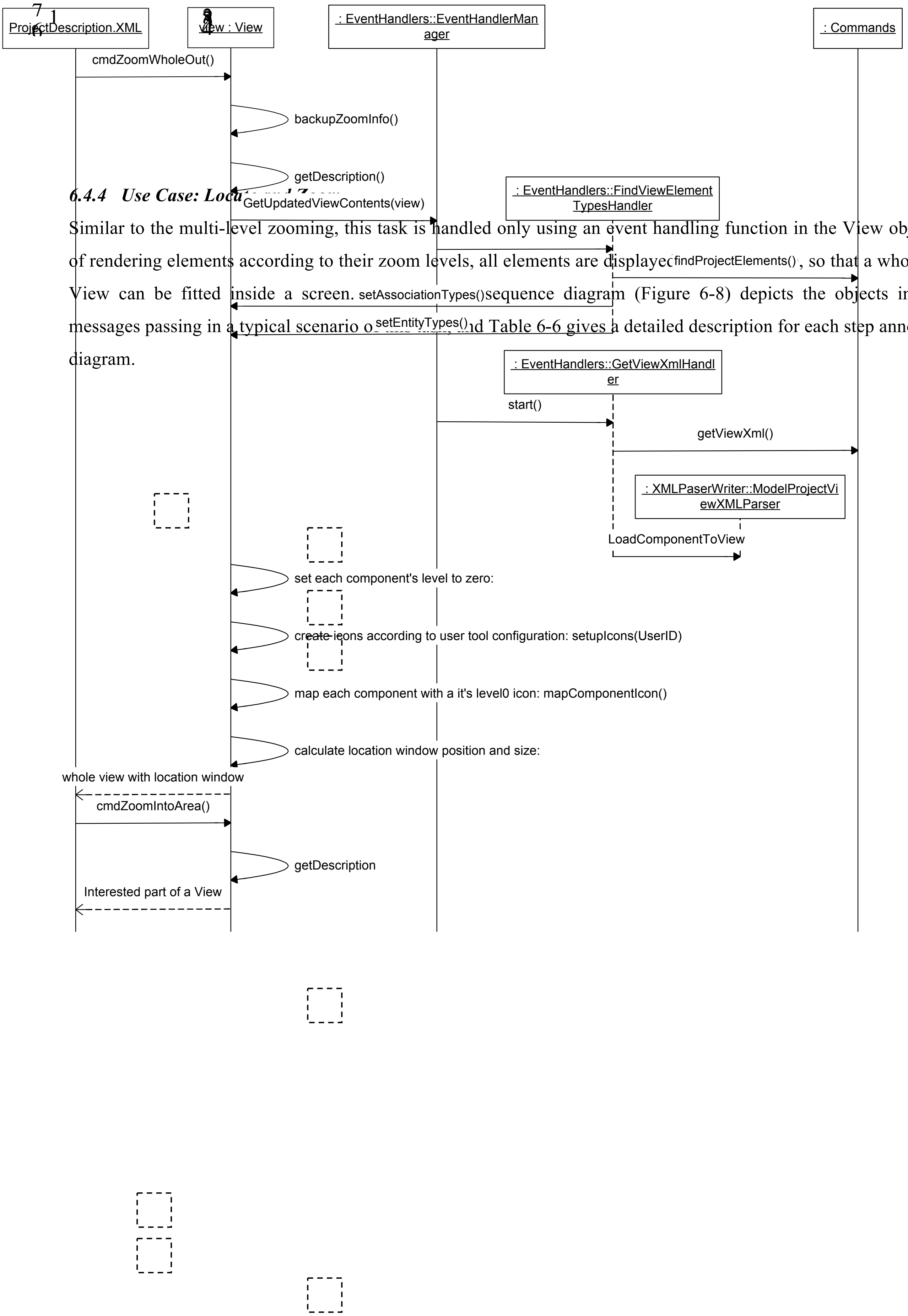

7
1
ProjectDescription.XML
8
4
view : View
: EventHandlers::EventHandlerManager
: Commands
cmdZoomWholeOut()
backupZoomInfo()
getDescription()
GetUpdatedViewContents(view)
: EventHandlers::FindViewElementTypesHandler
findProjectElements()
setAssociationTypes()
setEntityTypes()
: EventHandlers::GetViewXmlHandler
start()
getViewXml()
: XMLPaserWriter::ModelProjectViewXMLParser
LoadComponentToView
set each component's level to zero:
create icons according to user tool configuration: setupIcons(UserID)
map each component with a it's level0 icon: mapComponentIcon()
calculate location window position and size:
whole view with location window
cmdZoomIntoArea()
getDescription
Interested part of a View

Figure 6-8: Sequence Diagram of the use case Locate and Zoom

Table 6-6: Use Case Event Description: Locate and Zoom

Steps	Description
1	Invoked when user requests to zoom the whole view out
2	Invoked by step 1, store current zoom level information of all view components, which are necessary when zooming to another interesting location.
3	Generate view page by invoking getDescription function
4	Call TaskHandlerManager to get the latest view data information and replace the current component objects in the view. This data updating is not necessary if users only get access to Pounamu application through Pounamu MUPE Server. It is necessary if users access Pounamu application from multi-pass, such as Web server and Pounamu itself.
5	Set each component's zoom level to zero
6	Generate and return the view page by going through the same process as displaying view, but also create a navigation window
7	Invoked when user asks to zoom the view into a new interested location
8	Refresh the view and scroll the screen to the new location

Reusable Artefacts:

The facilities of scrolling a page and navigating with a small locator window could be integrated in the client script language and the MUPE client in the future.

6.4.5 Use Case: Add Entity

It is handled by the function cmdAddEntity( ) in the View object. The task Add Entity needs both its own UI and communication with the Pounamu application server. So it is implemented using a request handler object, CreateEntityHandler. The task “Add entity” is executed by a Command object in the Pounamu application. Similar to the other tasks including “Add Association”, “Move Entity”, “Move Association”, “Remove Element”, and “Edit Element Propert[illegible] mu/Mobile ser[illegible] Command object in the Pounamu application server. The following sequence diagram (Figure 6-[illegible]picts the objects involved and messages passing in a typical scenario of this task, and Table 6-7 gives a detailed descripti[illegible] step annotated in the diagram.

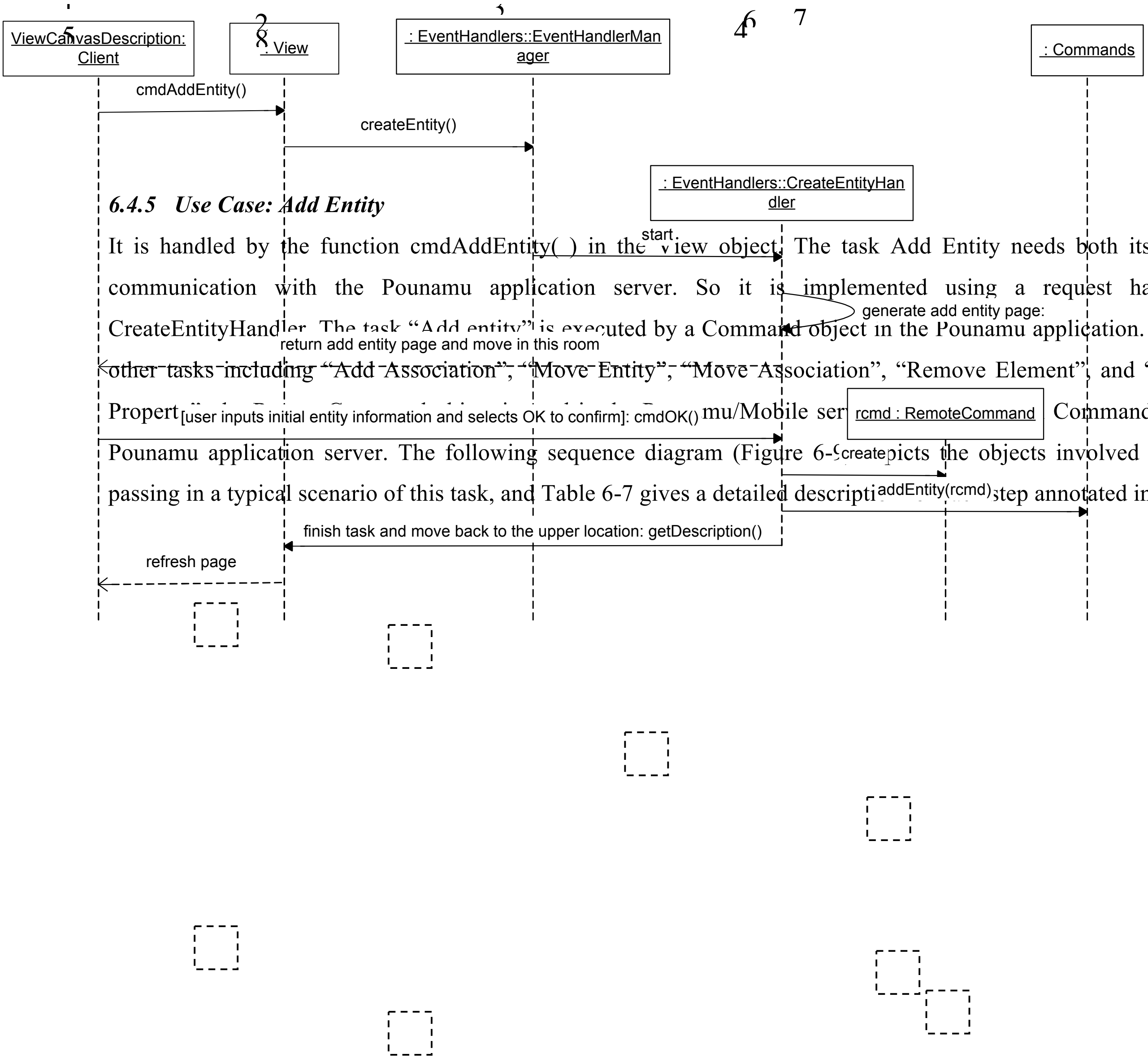

ViewCanvasDescription: Client
: View
: EventHandlers::EventHandlerManager
: Commands
: EventHandlers::CreateEntityHandler
rcmd : RemoteCommand
cmdAddEntity()
createEntity()
start
generate add entity page:
return add entity page and move in this room
[user inputs initial entity information and selects OK to confirm]: cmdOK()
create
addEntity(rcmd)
finish task and move back to the upper location: getDescription()
refresh page
5
2
8
3
4
6
7

Figure6-9: Sequence Diagram of the use case Add Entity

Table 6-7: Use Case Event Description: Add Entity

Steps	Description
1	Invoked when user asks to create a new entity
2	Invoked by step 1, to perform the task
3	Create and start task handling object CreateEntityHandler
4	Invoke the function getDescription in the object CreateEntityHandler to generate and return task wizard page of Create An Entity
5	Invoked when user inputs and necessary information for creating
6	Create an RemoteCommand object storing new entity data information
7	Invoke addEntity function in the remote editing object Commands, taking the new created RemoteCommand as argument
8	If the command executes successfully in the application server, then refresh the view by invoking view object’s getDescription function

Approach Generalization:

Task Annotation: This approach is generalized from our experiences of using UI features such as task wizards and status bar when we designed the UI interactions for the tasks of Pounamu/Mobile, such as the Add Entity. These UI features have been found very effective in making applications easy to learn and use, especially for webpage-based mobile UIs.

This generalized approach allows designers to specify a status annotation for each state and its sub-states (task steps) when designing application UIs using our MUI Development Process. When mapping tasks to UIs and interactions, these annotations could be displayed anywhere they are needed, such as in the title of a page, the label of a control, or text messages in a status bar.

For example, consider the state and its sub-states/steps of the task "Add Entity", the annotations of its sub-states include "Add Entity Wizard"(the title of entity creator page), "select type for the new entity" (the label of the radio button group of entity types), "input the name" (the label of the name edit-box), "give initial position" (the label of the position edit-box group), "move the new entity to desired place" (displayed in the status bar), and "press the confirm key after moving" (displayed in the status bar).

6.4.6 Use Case: Move Entity

The function cmdMoveShape( ) of the View object handles the event of Moving an Entity. Although this task doesn't have its own UI, completion of the task requires sending the command bac [illegible] pplication to execute. The TaskHandlerManager object is in charge of creating and starting the corresponding request handler object MoveShapHandler to perform the task. The following sequence diagram (Figure 6-10) depicts the objects involved and messages passing in a typical scenario of this task, and Table 6-8 gives a description for each step annotated in the diagram.

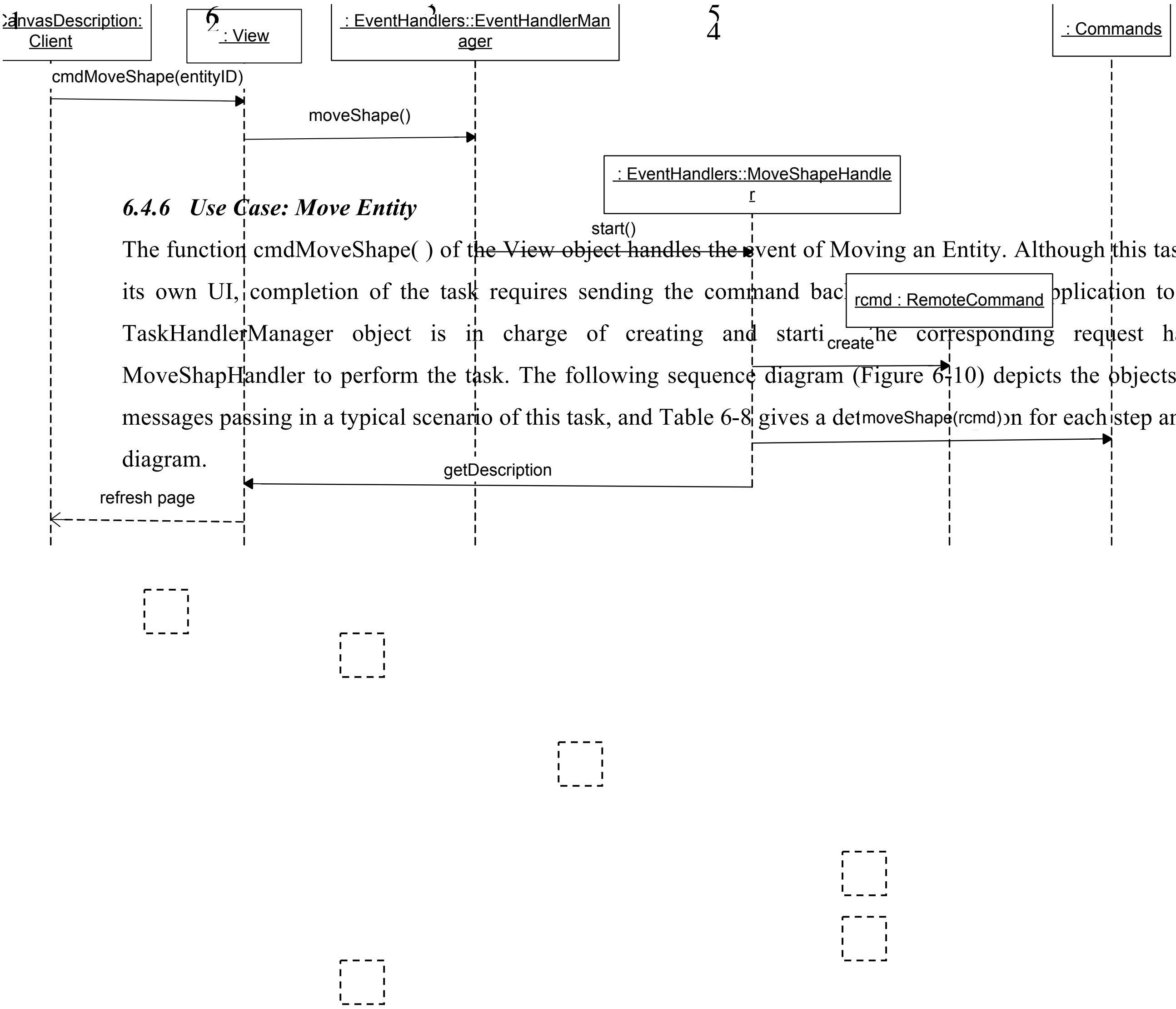


Figure6-10: Sequence Diagram of the use case Move Entity

Table 6-8: Use Case Event Description: Move Entity

Steps	Description
1	Invoked when user asks to move an entity
2	Invoked by step 1, to perform the task
3	Create and start task handling object MoveShapeHandler
4	Create an RemoteCommand object storing new position of the Shape.
5	Invoke moveShape function in the remote editing object Commands, taking the new created RemoteCommand as argument
6	If the command executes successfully in the application server, then refresh the view by invoking view object's getDescription function. The entity will be rendered in the new position.

6.4.7 Use Case: Edit Element Properties

The event is generated in the Pounamu View state and handled by the function cmdEditProperty( ) in the View object. The task Edit Element Properties needs its own UI and communication with application server, so it was implemented using the request handler object, EditPropertyHandler. The Pounamu/Mobile server doesn't the elements' properties. The EditPropertyHandler gets properties of an element from Pounamu application, and then, after the user submits changes, it commits the changes back to the Pounamu application server through a RemoteCommand object. The following sequence diagram (Figure 6-11) depicts a the objects involved and messages passing in a typical scenario of this task, and Table 6-9 gives a detailed description for each step annotated in the diagram.

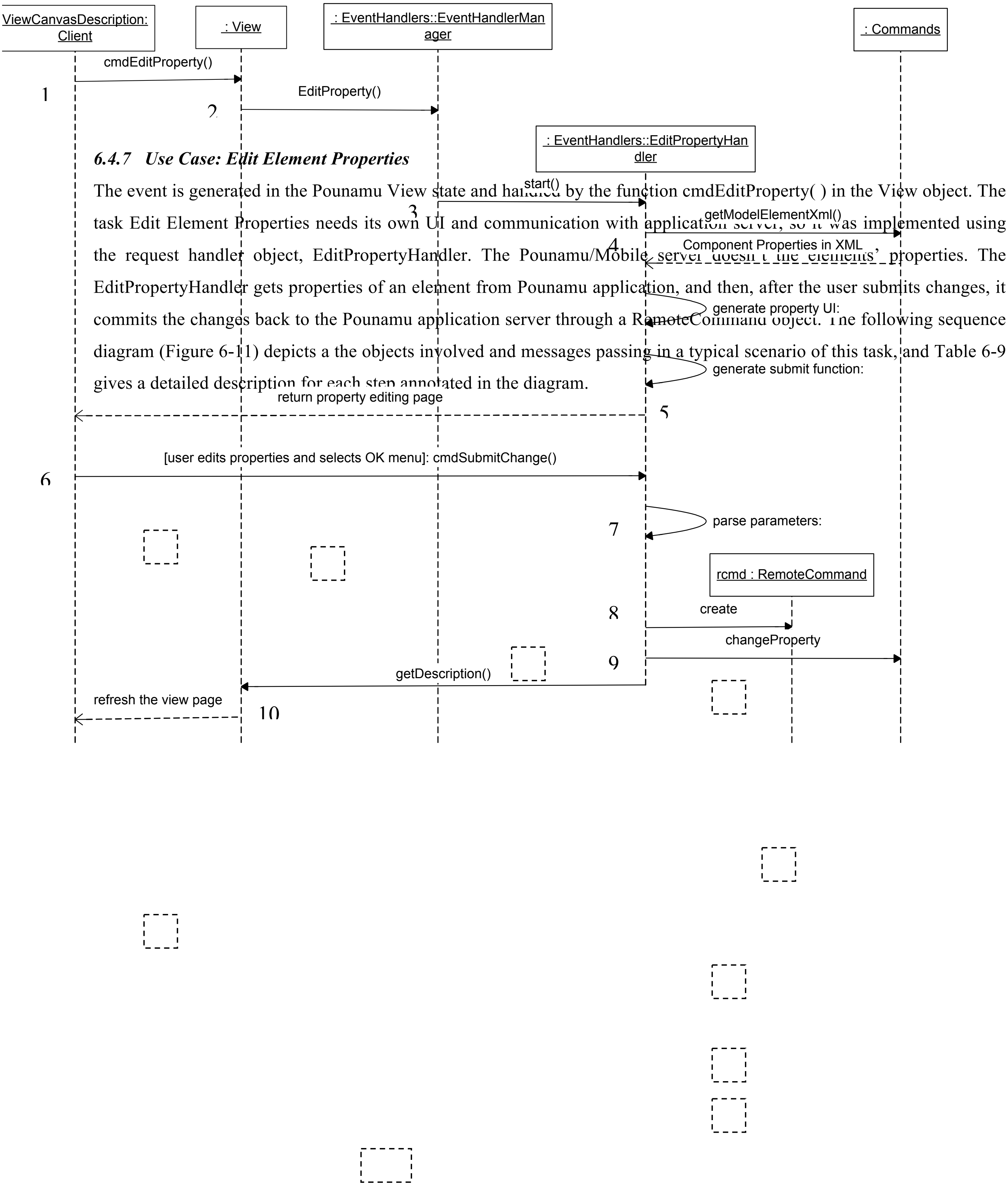

ViewCanvasDescription: Client
: View
: EventHandlers::EventHandlerManager
: Commands
cmdEditProperty()
1
EditProperty()
2
: EventHandlers::EditPropertyHandler
start()
3
getModelElementXml()
4
Component Properties in XML
generate property UI:
generate submit function:
return property editing page
5
[user edits properties and selects OK menu]: cmdSubmitChange()
6
parse parameters:
7
rcmd : RemoteCommand
create
8
changeProperty
9
getDescription()
refresh the view page
10

Figure 6-11:Sequence Diagram of the use case Edit Element Properties

Table 6-9: Use Case Event Description: Edit Element Properties

Steps	Description
1	Invoked when user asks to edit properties of a selected element
2	Invoked by step 1, to perform the task
3	Create and start task handling object EditPropertyHandler
4	Get element properties in XML by a remote method invocation through the remote editing object Commands
5	Return Property Editing Page by Invoking the function getDescription in the object EditPropertyHandler. This not only generates UIs for editing properties, but also the function cmdSubmitChanges. The function is quite special. Different element types have different numbers and types of properties to be transferred between client and server. Because MUPE client doesn't support passing objects, it is hard to store properties into an object in client side and send it back to the server. So the function is required to take different parameters according to element type.
6	Invoked when user finishes editing and submits changes back
7	Event handling function in the EditPropertyHandler is in change of parsing out the arguments, which are property values.
8	Create an RemoteCommand object and store element property values
9	Invoke changeProperty function in the remote editing object Commands, taking the new created RemoteCommand as argument
10	If the command executes successfully in the application server, then refresh the view by invoking view object's getDescription function

6.5 Completed Object Model

Figure 6-12 shows the completed object model of the Pounamu/Mobile server. Compared with the original object model, the new objects involved during the use case realization are added in and their details are described in the following table 6-10. Please note that the View object implements not only the location states Diagram Pounamu View and Tree Pounamu View, but also task states that share the UI with it, including Move Entity, Remove Element, Multi-level Zoom Element, Scroll Screen, and Locate and Zoom.

Table 6-10: Class Description Additional

Class	Description
ComponentIcon	Responsible for visual configuration of elements. Because different types of elements require different rendering configuration. So we need to create an object stores configuration for each type of element, rather than share a single configuration xml file for all element objects.
ShapeIcon	Derived from ComponentIcon and represents rendering configuration for an entity type.
ConnectorIcon	Derived from ComponentIcon and represents rendering configuration for a association type.
XMLParserWriter	It is a package, which contains a set of specific XMLParserWriter Classes. Each class is responsible for loading a specific data object from xml file and writing it back to xml file.
Commands	Responsible remote editing. The server has one global Commands object
RemoteCommand	Refers to a real Command object in the Pounamu application

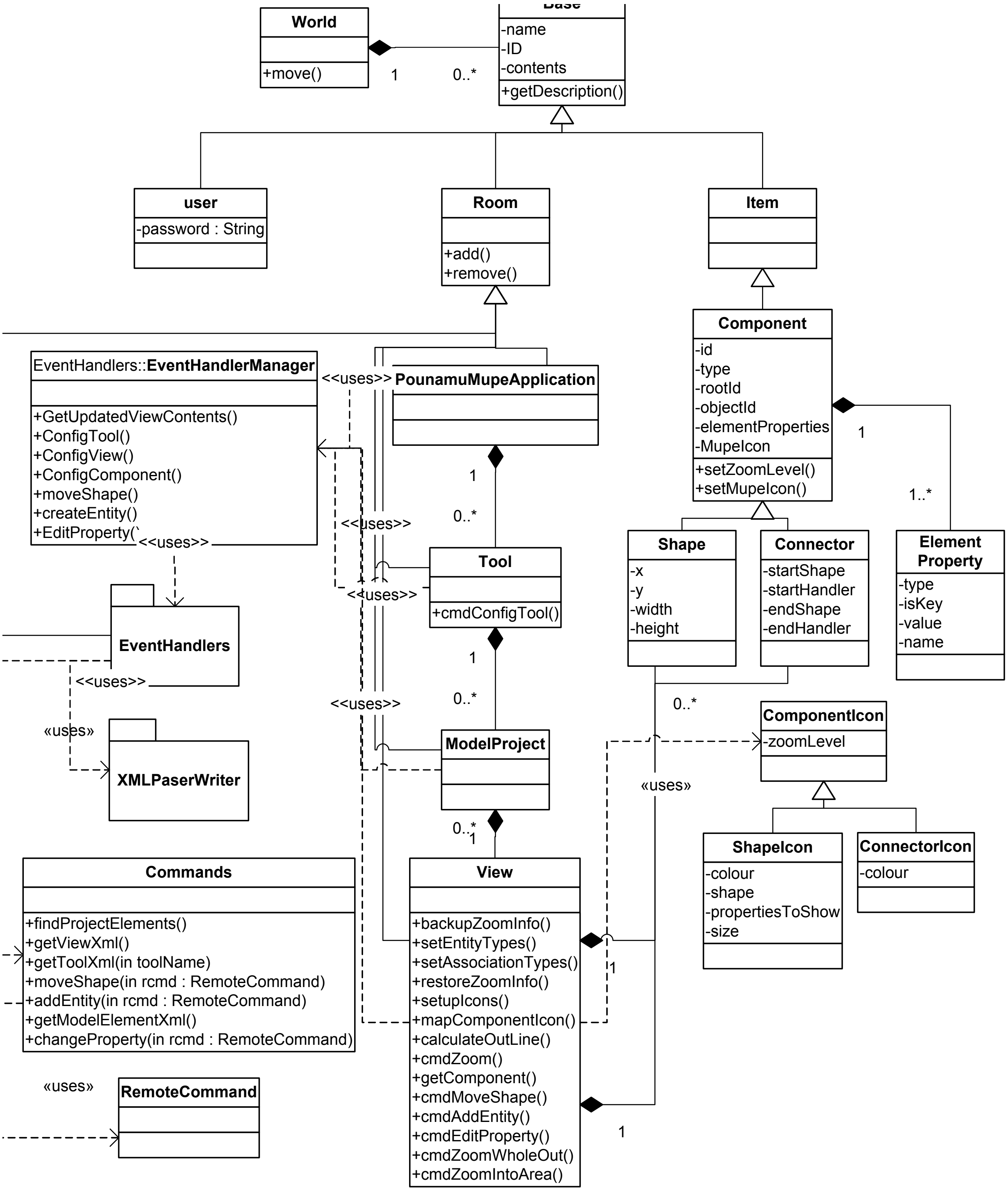


Figure 6-12: Class Diagram: The completed conceptual object model of the Pounamu/Mobile

6.6 Summary

In this chapter, we have again explored the benefits of using the FSM to describe the UI system. It is fairly simple and clear to map the system states and their events to the classes and the event handling functions, though implementing the device specific UI interaction designs is quite complicated. We also generalized our main approaches to be integrated in either the MUPE client or the MUPE server to support creating advanced mobile thin-client UIs for other MUPE applications. These generalized approaches include multi-user runtime configurable UI configuration, multi-level zooming, and task annotation.

To future work, based on our progress so far, we believe that a visual designing tool could be created using Pounamu to support developing mobile UI systems. At the beginning, the tool will support description of a UI system using the state-event concept in system state diagrams and generating main classes and event handling functions in them as a customized MUPE server. Later on, after our approaches have integrated into MUPE, the tool could allow users to choose various facilities to design high quality mobile thin client UIs. However to describe our MUI Development Process using a tool and support automatic ready-to-run code generation, we still need to study more about the process of mapping between abstract UI interaction designs and device specific UI designs.

Figure: Completed Class Object Model of Pounamu MUPE Server

Chapter 7 - Evaluation

7.1 Introduction

This chapter presents an evaluation of the Pounamu/Mobile prototype. Our aim was to build mobile thin client UIs that support an equivalent capability and usability to the original application running on PCs. We evaluate Pounamu/Mobile using both the cognitive dimension framework and by a small survey. In this chapter, firstly we present an evaluation of the Pounamu/Mobile prototy`pe using Green and Petre's cognitive dimensions framework [Green 96], comparing this with the thick-client and thin-client versions of Pounamu on a PC. Then we describe feedback from a small survey, where several users shared their impressions of our mobile thin client environment for diagramming tools. Finally we summarize the strengths and weakness of our system, and recommendations for future work.

7.2 Cognitive Dimensions Evaluation

The cognitive dimension framework is a tool to aid non-HCI specialists in evaluating usability of information-based artifacts. The cognitive dimensions approach consciously aims for a broad-brush treatment rather than a lengthy, detailed analysis. It helps to ensure that serious problems are not overlooked. The framework was introduced by Green, who emphasized the generality of this approach. It is applicable over all domains where information representation is used. Green and Petre (1996) presented a lengthy analysis of one particular domain, visual programming languages. It is the most detailed presentation of the framework currently published.

The approach is called cognitive because it focuses on usability aspects concerning the mental difficulty of learning and using a system. For example, the degree to which the system makes users translate one operation in the head into a sequence of individual actions is cognitive, since it relates to the user's conception of an operation. In the followings, we evaluate the usability aspects of Pounamu/Mobile using those cognitive dimensions that are of our interest. We apply these dimensions to our system from three different points of view, which are meta-tool, modelling tool, and application UI.

Viscosity: Resistance to change

The viscosity dimension is used to measure the cost of making small changes. It means how much work the user has to put in, to implement a small logic or structural change in a local model. Abstraction can reduce the viscosity of a system. In the following, we discuss the viscosity of Pounamu/Mobile from three points of view; meta-tool, visual modelling tool, and application UIs.

From a meta-tool point of view:

In Pounamu, changes in tool specification automatically update all model projects using the tool. For example, a user modifies the visual appearance of an entity type, and then the system applies changes to all entities with the type in all project models. This abstraction facility of Pouanmu as a meta-model tool reduces the viscosity of the system. Thin-client

Pounamu has the same viscosity from this point of view, since it displays Pounamu views exactly the same as in thick-client Pounamu. Pounamu/Mobile can reflect all changes on a tool specification from Pounamu in runtime, except for visual diagrams. As we know, Pounamu/Mobile doesn't render entities as being defined in Pounamu. Instead Pounamu/Mobile supplies facilities for users to specify their own visual configuration of tools. Similar to the abstraction facility of Pounamu, changes of an entity type in configuration automatically apply to all entities with the same type. However, though this tool rendering configuration in Pounamu/Mobile supplies a much simpler icon definer with limited configurable visual properties compared with the Shape Designer in the thick-client version, it allows multi-user to specify their own tool configuration and define multiple diagram icons for one element type in different detail levels. There is no equivalent in the thick-client version. All these mean different visual forms for the mobile version and thick-client version, so there may be consistency issues when users migrate between two versions.

From a visual modelling tool point of view:

The viscosity of a tool apparently depends on how a user defines the tool. User event handling facility in Pounamu helps to reduce tool viscosity by allowing user to create refactoring functionalities for the tool. The real program codes of user-defined event handlers execute in the Pounamu application itself. Both thin client and Pounamu/Mobile take advantage of this facility. For example, in the Pounamu UML tool, a user-defined event handler is responsible for removing all associations related to an entity when a user removes the entity.

Doc/View approach, which is applied to Multi-view, reduces the viscosity of tools as well. For example, changes concerning an entity in one view actually modify the shared model, which is common for multiple views. Users need not repetitively make changes for the same entity in each view containing it. Both Pounamu/Thin and Pounamu/Mobile implemented the operation of adding an existing entity to another view, so that they take advantages of this facility too.

From a UI point of view:

The thin client makes some operations harder to perform for users than with the thick client Pounamu. This happens in Pounamu/Mobile too. For example, consider the operation "Moving Entity", which is a simple drag-and-drop from the tree view to the diagramming view in Pounamu thick client. An alternative complex process completes the same task in Pounamu/Mobile, by selecting an entity in a view, and then choosing the menu "Move Entity", then moving the shape, and at last clicking the confirmation key. However this increase in viscosity is balanced with the increase in mobility gained. For example, driving back to your office to make a change to a diagram, which you have had to point out for discussion with a client is far more of an imposition than a few button clicks.

Hidden Dependency

Hidden dependencies occur in programs when one part is affected by another in a way that is not overtly signalled on the page. Applied to Pounamu, it means that two Pounamu elements are dependent on each other, but the in-between relationship is not fully visible to a user. With hidden dependencies, while modelling using a tool, it is easy for a user to create unexpected bugs

From a modelling tool point of view:
Hidden dependencies in models and views occur more frequently in Pounamu/Mobile than in Pounamu. Dependencies between entities and associations are displayed using shapes and connectors inside a Pounamu View. The pop-up menu of an element shows all its related information, such as multiple views, which contain the same element. Users can therefore easily view dependencies of an element before making changes to it. However in the other two versions, the ways of displaying these dependencies are quite different. Pounamu thin client requires users to look up similar information via buttons and non-contextual option lists.

In Pounamu/Mobile, due to the small screen interface, users either get the whole of a Pounamu view but with less information displayed, or a part of a Pounamu view with detailed information. Some dependency information may be missed out in the whole view, such as roles of an association and attributes of an entity. The detailed view may not have all dependent shapes and connectors displayed in one screen. Also due to the small screen, users have to browse element related information such as properties and multiple views containing the same the larger screens of Pounamu thick client and thin client than in Pounamu MUPE.

Consistency
Similar semantics are expressed in similar syntactic forms. Consistency usually affects learnability rather than usability of a language or software. Consistent notations or information structure tend to minimize user knowledge required to use a system by letting users generalize from existing experience of the system or other systems. Generally consistency of an application can be evaluated by asking questions, such as how easy users feel about learning the system, how much of the rest can be successfully guessed?

From a meta-tool point of view:
Pounamu is quite easy to learn. All tools specified in Pounamu use the same meta-meta model, which a project model is composed of entities and associations, and a Pounamu view is composed of shapes and connectors. As a user learned how to use one tool, the user knows a major part of other tools. The same meta-meta model has been used in Pounamu thin client and Pounamu/Mobile.

From a UI point of view:
We used similar design styles to other mobile applications, to make UIs of Pounamu/Mobile consistent and easy to learn. All location states share similar UIs. For example, if a user learned how to load a tool in the Pounamu/Mobile Application state, then he/she knows how to load a model project in the Tool state. For the diagramming view, the same keys are used to select elements, move entities, scroll screens, and move the navigation window. The same process is designed for various tasks, which start with selecting from the command menu and ends with pressing confirm key. All these designs make our Pounamu/Mobile easy to learn. In addition, UIs of the three Pounamu versions implement the same set of tasks/operations, though their appearances are quite different from each other. If a user is already familiar with one of them, it requires little extra for the user to study the other two versions, except that Pounamu/Mobile has three additional new features, multi-user tool rendering configuration, multi-level zooming, and view navigation.

Closeness of mapping

This dimension measures closeness of representation to the problem domain. Applied to visual programming language, it means that ideally, the problem entities in the user's task domain could be mapped directly onto program entities, and operations on those problem entities would likewise be mapped directly onto program operations. Applied to UIs, it means that ideally, the application data could be mapped directly onto the visual representations, and UIs and interactions for completing tasks would map intuitively how users think they should be.

From a modeling tool point of view:

Pounamu/Mobile has less ability than Pounamu and Pounamu/Thin to map user-defined diagrams to actual domains. Pounamu allows users to design complex diagrams to closely map the actual domain. For example, the electronic tool mentioned in the chapter 2 is closely modeled on an actual circuit diagram, minimizing the number of new concepts that need be learnt. However these complex diagrams can't be drawn using 2D graphic library in the current MIDP version, which MUPE client is based on. One possible solution to solve this is transferring a Pounamu view as an image and displaying the whole or part of the image on mobile devices. But diagrammatic editing with an image is hard on mobile phones without a mouse or stylus.

From a UI point of view:

We feel that UIs in three versions of Pounamu have same closeness of representation to the task domains. Although UIs and interactions are different in three versions and some tasks are performed more complexly in Pounamu/Mobile than in Pounamu thick client, they are all derived from the same people mental model of how to complete Pounamu operations. For example, consider the operation "Add Entity", which a person's mental process of completing this task is firstly telling the system that he wants to create a new entity, then giving the necessary property values including type, name, and position, and finally asking the system to create it. This process is performed in the Pounamu thick-client by clicking on an entity type in the left hand tree view, then dragging and dropping into the right hand side diagram view, and inputting the name in the created entity. Alternatively, several steps in Pounamu/Mobile include firstly selecting the menu of Add Entity, and then following the wizard to select entity type, give name, select OK to launch the command, and at last move the created entity to a desired location. Survey participants feel the process with wizard match their mental imagination closely. It is even more intuitive than Pounamu thick client for users who are not familiar with the facility of drag-and-drop on PCs.

Hard mental operations

This dimension is intended to refer to combinations of primitives and means high demands on Cognitive Resources. Generally speaking, hard mental operations can cause user fatigue and mistakes. When applied to application UIs, it overlaps with closeness of mapping and consistency. They are all about how easy the UIs are for users to learn and use.

From a UI point of view:

We feel Pounamu/Mobile requires a similar amount of hard mental operations to the other two versions. We designed the UIs to reduce user mental operations. All tasks are menu-driven, followed by task wizards. Each wizard was designed to match people mental process of performing the specific task, so that it is easy for users to use the system. In addition, the learning curve of the thin-client and Pounamu/Mobile is less in many respects to the thick client tool, as system status and interaction options are presented more explicitly in the single webpage-based UIs. However some editing actions, such as moving, require complex sequences of steps.

Visibility and Juxtaposability

The visibility dimension denotes simply whether required material is accessible without cognitive work. Long or intricate search trails make poor visibility. Contrast with hidden dependencies: visibility measures the number of steps needed to make a given item visible, hidden dependencies describe whether relationships are manifest. Juxtaposability is the ability to place any two components side by side. For example, trying to compare two statistical graphs on different pages of a book. Systems that present information in a single window lack Juxtaposability.

From a UI point of view:

Pounamu/Mobile has less visibility than the other two versions caused by tiny windows. Theoretically, most of the data of an element could be visible in the diagram, however usually only some of them are specified to be visible according to user's interests. Pounamu thick client and thin client display diagrams as user defined. However the same set of data information can't be displayed all in a tiny screen on mobile devices. We proposed many approaches to improve the visibility, including multi-level zooming, status bar, screen scrolling, and navigation window. Therefore most of the data for views and elements are accessible within one or two additional steps. The element management tree in both Pounamu and Pounamu/Mobile increases accessibility of an element in a large model.

Thick-client Pounamu has high Juxtaposability. Multiple views display side-by-side, visual and model properties of model elements are simply available via the right-hand-side property panel. In comparison, Pounamu/Mobile runs in tiny screen, in which Juxtaposability has been sacrificed.

7.3 User Survey

To assist us in evaluating our work, we carried out a small user survey where participants were asked to complete a list of tasks and answer a number of open-ended questions. All participants were chosen from those who have experience with the thick-client Pounamu. No special tutorial was provided for the Pounamu/Mobile on purpose, so we could evaluate how easy it is for users to learn and use our mobile thin-client version of Pounamu.

Tasks

We proposed two tasks for participants to perform. These tasks covered the basic functionalities of Pounamu/Mobile and its new facilities. Task 1 requires evaluators to model a class diagram using the UML diagramming tool specified in Pounamu on the mobile thin client. In this task, evaluators have to learn how to use the tool by themselves through our intuitive webpage-based UIs and complete the task using functionalities supplied on Pounamu/Mobile. In task 2, evaluators were asked to specify their personal tool rendering configurations and define multiple icons for elements through the runtime tool configuring UIs, and the redo task 1 with new features such as multi-level zooming and view locating. This task was designed to test whether our new features are beneficial to users of diagramming tools on mobile devices, and to obtain comments for further improvement.

Open-ended questions

The survey questions were designed to cover two general topics. The first questions focused on how the users felt about the functionality and usability of the prototype. The survey asked questions like "Did you find it easy or hard to figure out how to do modeling in Pounamu/Mobile?" and "Does the mobile version supply all necessary functionalities for you to perform modeling?" The questions were designed to find the good and weak points of the UIs, to find whether the facilities such as status bar and wizards help users to learn and use the system, to discover which operations were easy to carry out and which were not and how we could improve them. The second group of questions focused on the new concepts of multi-user personal tool configuration and multi-level zooming. Questions asked included "Did you find the new features are helpful for using diagramming tools on mobile devices?" These questions were designed to give us feedback on our approaches, whether they are useful to the users and how they might be improved. We also wanted feedback on what additional functionalities the system should have to make mobile diagramming more effective.

Survey results

The general response to the survey was very good. The majority of the respondents did find the system easy to learn based on their concepts of diagrammatic modeling from Pounamu. Although the mobile version supplies dramatically different UIs from its original thick client ones, users found it very easy to figure out how to perform modeling by following the command menus, task wizards, and status messages in the low-complexity single webpage-based UIs. Users generally agreed that the prototype has all necessary modeling functionalities to complete task 1, compared with the thick-client version. The almost universally mentioned suggestions were reducing the number of interactions in performing certain operations such as moving an entity.

The responses to the second set of questions are very positive. While surprised by the innovative features, users felt it more effective to manipulate models with the support of these new facilities. The personal tool rendering configuration and multi-level zooming were appreciated, although they were found to suffer from some shortcomings. Particularly, as we already knew, users also desired to have the same icon displayed on mobile phones as specified in Pounamu and being able to define more advanced icons more easily and flexibly. However everyone had different ideas on ways that these could be improved. There were some interesting issues raised and suggestions made, the ones we felt had the most relevance and potential are included in the future work section of the Conclusion and Future Work chapter.

7.4 General Comments

Strengths

Generally we are very happy with the Pounamu/Mobile we created. We feel that the innovative approaches that we have proposed and implemented in the prototype make it possible and feasible for diagrammatic browsing and editing on the mobile devices. The approaches we have taken for designing UIs make the mobile version easy to learn and fit well with people mental concept of diagrammatic modeling operations.

From the survey feedback and our own experiences, we feel that the ability to manipulating with models through both a detailed view and overview, as a user desires, is a major benefit. The multi-user personal tool rendering configuration through the runtime mobile configuration UIs allows users to define icons to be rendered and details to be displayed as they desire. With multi-level zooming and navigation, users felt it distinctly easier and more effective to manipulate models. This was primarily, we believe, because these new features improve the visibility and reduce hidden dependency of models.

We found the UI designs that we applied to the prototype are very helpful for improving the usability and quality of mobile UIs of Pounamu. Compared with the thick-client counterpart, most users felt that Pounamu/Mobile supplies all the essential functionalities through less sophisticated single webpage-based UIs. Command menus list all events that drive the system to the next possible state. To users, it clearly shows all possible operation options that can be performed at the current state. This could be improved to only show possible options for the user's current status, which is explained in more detail in the future work section of the next chapter. Task wizards, the UIs of task states, make it easy for users to follow the steps and complete operations. The status bar helps users have a clear mind at any time about where they are, what they are doing currently, and what the next step should be performed.

Weaknesses

Weaknesses and comments about the prototype discovered both from a cognitive dimensions evaluation and survey are very important, because they give us insight into potential future work. Many of them actually indicate shortcomings of the current device type we chose for creating UIs. Until now the major issue raised by survey is that multiple interactions are required to perform editing operations such as moving an entity. We doubt that there will be major breakthroughs on this aspect with regard to mobile phones without necessary facilities such as stylus pen and touchable screen. Although this increases the viscosity of the system, it is balanced by the mobility obtained by users. From users' responses, they hope to have the ability to render the same icons of shapes and connectors on mobile phones as in the Pounamu thick client. As mentioned earlier, there is not too much that can be done to improve this, due to the current 2D graphical rendering ability of the MUPE client. Shape handlers used to attach connectors therefore can't be matched on the simpler shapes in the mobile version. The simple diagrams finally cause poor closeness mapping from real problems to models. However, similar to some other issues include response time of repainting a view page and ability to render a big model, these issues will be solved with the rapid development of mobile hardware and platforms.

Recommendation on Future Work

Many of other issues raised by the survey and most of our own reservations come squarely in the category of future work on Pounamu/Mobile. In other words there are a number of concerns that we simply did not have time to address in this prototype, but are certainly important issues for the future. These include:

- More flexible tool rendering configuration as we discussed in the design and implementation chapter
- Advanced shape designer to define more complex icons
- The ability to change a tool specification
- Group awareness to improve collaborative working.
- Undo/redo facilities to roll back and repeat the last operation.
- Resize operation in the zoomed-out view

Our suggestions for extensions to these issues that we regard as potential future work are discussed in the next chapter.

7.5 Summary

In this chapter we carried out an evaluation of our Pounamu/Mobile on several diagramming tools. We applied the cognitive dimensions framework in this evaluation to analysis usability aspects of our Pounamu/Mobile. We described the feedback we received from a survey we carried out on our system and generally discussed the strengths and weaknesses of our approach with a eye toward the potential future work.

Chapter 8 - Conclusion and Future Work

8.1 Introduction

This chapter is the concluding chapter of the thesis. It presents a summary of the main contributions of this thesis to the research field and outlines the areas we see having the most potential for future work. Finally there is a general summary for this thesis.

8.2 Contributions of this thesis

This thesis makes several contributions to the area of MUI at three different levels. The presented development process, which we called MUI Development Process, provide the first step towards future work on providing tool support for developing and auto-generating advanced MUI with high quality and usability. Our innovative approach and proof-concept prototype demonstrate the possibility of supporting advanced diagramming UIs on mobile devices. We generalized the approach to support for generation and construction of advanced mobile thin-client UIs with high quality and usability for diagrammatic modelling tools and other general applications.

Our MUI Development Process

The analysis and deign process of Pounamu/Mobile that we described in the previous chapters gave us many ideas and experiences for future work in providing tool support for creating mobile UIs and MUI. In the thesis, we mapped conceptual tasks to event-driven UI system states, which brings people's mental model of designing UIs one step closer to the real UI designs. We then mapped the abstract states onto mobile thin-client pages through the following steps. We designed data visual representations for each state by splitting big states to several smaller ones, choosing widgets for different types of data, and designing the layout of each page. We raised device-dependent user interactions for performing tasks to a device-independent abstract level, and then mapped them to the context (facilities provided on mobile phones). Then we proposed our approach solving the problems caused by the device constraints to improve the usability of the device specific UIs. For our long-term goal, however, there are much more to be understood about this process before a promising MUI Development Framework/Tool is able to support effect development of advanced thin-client MUI system.

Our Approach to provide effective dynamic diagramming on mobile devices

We have proposed a solution using a set of approaches to provide the ability to display and edit diagrams on a wide range of mobile phones. We have tested our approaches in three visual modeling tools provided by the meta-CASE tool Pounamu, including a UML CASE tool with class, collaboration, use case, sequence diagram and deployment diagram views, a project management tool with Gantt chart and work breakdown views, and a business strategy modeling tool with process modelling views. The main contributions in this solution are:

- *Multi-user runtime configuration.* To address screen size and rendering limitations on mobile devices, we proposed a multi-user runtime tool configuration facility. This allows users to specify and modify an item's visual representation though mobile UIs as they desire at any time while using the diagramming tools. It provides diagram-based UIs the ability to be adaptable to both users' preferences and the current task being performed.

- *Multi-level zooming.* To help manipulate large model views in a tiny screen, we proposed a multi-level rendering and zooming approach. This facility allows users to define multiple representations for a single diagram at different levels of detail. While manipulating model views, each diagram element can be separately selected and zoomed among multiple levels. We feel this approach together with the following navigation facility improves dramatically the efficiency of diagrammatic modelling on a tiny screen.

- *A set of navigation approaches.* We use several navigation approaches to improve user interaction with diagramming tools on mobile devices. Button-panning allows users to quickly move around in a big diagram using mobile phone buttons. A floating zooming window allows users to easily and quickly zoom to interesting areas from an overview of a large diagram.

- *The thin client technology.* To address the requirement of deploying UIs on a wide range of mobile devices, we chose thin client as the underlying deployment technology, because it has several advantages over the traditional thick clients. There is no need to install and keep updating software on each end-user's machine. It provides simple, easy-to-use and consistent user interface design. It seamlessly supports collaborative work via the client-server approach inherent to web applications.

Combined these approaches extend accessibility of diagramming applications and allow group collaboration at different locations to be carried out in a straightforward manner.

Generalization of the approaches

We have generalized and implemented these approaches to proof-of-concept level in order to help generate advanced mobile UIs for any diagramming tools specified in Pounamu. We have prototyped Pounamu/Mobile, a mobile thin client plug-in component for Pounamu, using MUPE as the underlying thin-client platform. Any Pounamu specified visual designing tool can now be accessed from multiple devices via thick client, web-based thin client, and mobile thin client (MUPE client). Reflecting on the evaluation results and survey feedback, we feel that the mobile thin-client UIs generated by Pounamu/Mobile supplies similar capabilities to the previous thick-client Pounamu and has reasonably high quality and usability. In addition, this work supports further extensions to Pounamu for ubiquitous collaborative editing.

The successfully experience we had from building Pounamu/Mobile has driven us to further generalize our approach to a more abstract level to support creation of advanced mobile UIs for applications in a wider range of domains. For our long-term goal, this approach and experience have great potential to contribute as part of development libraries for a future MUI construction framework. Currently we have integrated some of these in the MUPE platform.

- *Multi UI description files:* The idea of multi-level zooming has been generalized and integrated into the MUPE server. Users can define multiple UI descriptions for a room or item, and at runtime, end-users can select any UI description to render a room or item. The runtime UI description selection can also be optimized for different size screens.

- *Multi-user runtime configurable UI configuration:* This generalizes our approach multi-user runtime tool rendering configuration. It contains two level configurations, which means that users can specify which visual properties are configurable and configure values of configurable visual properties both at runtime. Compared with the tool rendering configuration in our Pounamu/Mobile, it provides a more flexible UI configuration facility.

- *Task annotation:* UI features such as task wizards and status bar have been found very effective to make applications easy to learn and use, especially in tiny screens. We have generalized these features into our task annotation approach, which means that designers specify a status annotation for each task and sub-task when designing application UIs using task-based mental model. When mapping tasks to UIs and interactions, these annotations could be displayed anywhere they are needed, such as in the title of a page, the label of a control, or text messages in a status bar.

Many other approaches used in the Pounamu/Mobile are hoped to be integrated into either the MUPE server or the MUPE client in the future. Those with highest potential are outlined in the future work section.

8.3 Future work

Despite our progress, much remains to be done as outlined in the following.

Approach:

- A better layout algorithm for the multi-level zooming facility. We want to improve the current algorithm to better produce layouts for those diagram types relying heavily on spatial relationships, such as the work breakdown diagram in the project management tool. A better layout algorithm should calculate accurate positions for all elements to satisfy all spatial relationships among them, especially when various elements are zoomed to different levels in a view. Potential approaches may include spatial reasoning and other related approaches for mobile devices [AAAI 04, Evaluation of visual balance for automated layout IUI 04]

- Explore the limits of our approach. We need to extend our approaches to a wider range of devices, such as PDAs, PCs, tablet PCs, and even wall-sized displays. We are keen to see how diagram-based visual design tools benefit from these alternative interaction mechanisms.

- Apply our approach to other development platforms. We would like to apply our approach to mobile technologies other than MUPE, such as Mobile SVB, Symbian development framework in C++, and Microsoft.Net Mobile version.

Pounamu/Mobile:

- Further support for ubiquitous collaborative design: We would like to integrate our mobile thin-client version with the previous extensions to Pounamu including thick client-based collaborative editing support [Mehra 04] and the web-based thin client version. The integrated system would provide richer support for group awareness and synchronisation between multiple device users of a Pounamu specified tool.

- A more flexible and advanced shape designer. As we explore our approaches on other devices such as PDA and tablet PC, specifying representations of diagrams will definitely be improved by using interaction facilities such as stylus pens and touchable screens.

- The ability to change tool specifications through mobile UIs. Currently changing a tool specification can only be done with the swing-based thick-client tool designer. This has been seen as a disadvantage for some users if thin-client editing of the resultant tool is desired. Adding this extension to our Pounamu/Mobile and Pounamu/Thin should not be very difficult, since the hardest component of this function, the shape designer, has already been achieved on mobile phones.

- Context-aware command menu. The current command menu of a state is static. This could be improved using context-aware menu, where the command menu changes according to what a user is currently doing. For example, if a user selects an element in the canvas view state, then only those common commands and commands related to the selected element type are displayed in the state page.

- Other features. Other functionalities missed out in the current Pounamu/Mobile include undo/redo to roll back and repeat the last operation, and resizing shapes in the zoomed-out view.

Generalization of the Approaches:

We believe that MUPE has potential to act as a development platform for MUI applications. It is a great platform for us to rapid prototype and test our ideas on. We would like to realize and integrate our research results including approaches, architectures, and methodologies into MUPE, so that they could become reusable building blocks for other applications. In this thesis, the reusable artefacts that have occurred to us include:

- Task annotation: provide a mechanism for specifying annotations to tasks and mapping annotations to UI descriptions of Item and Room objects.

- Multi-user runtime UI configuration: create a mechanism of saving and loading UI configurations for Item, Room, and User objects, mapping configurations to object UI descriptions, and providing UIs for runtime configuration.

- Multi-level zooming: add a facility to the client script language and enable it in the MUPE client

- Navigation facilities: add scroll window and location window with their functionality to the client script language.

MUI Research:

Towards our long-term goal, we propose several future research projects on providing tool support for construction of advanced thin-client MUI. As we mentioned above, we will continue to explore experiences of building advanced MUI on a wider range of devices from our exemplars, generalize and integrate our results into a promising MUI development platform such as MUPE, and understand the process of mapping people's conceptual UI designs to the real ready-to-run implementation code. The potential directions and tasks we have in mind for future research include:

- Create a visual language tool using Pounamu. The tool will represent the process we used for analyzing and designing UIs for Pounamu/Mobile. It will support designing mobile thin-client UIs for MUPE applications and generate a Java program framework specification based on user-designed models concerning UI states and their task events. This generated code framework could be further extended to a MUPE server.

- Describe our Pounamu/Mobile using a task model-based MUI specification language such as XIML. In this thesis, we feel that task-based design [Johnson and Wilson 1993] is a human mental model that can closely describe users' conceptual UI designs. After exploring limits of the exiting MUI description languages, we can set our future work based on one of them.

- Perform a detailed survey on how users would design web-based MUI for applications. Users should be chosen from stage one or two computer science students who represent the end-users of a MUI construction tool. A human mental model for building thin client MUI for advanced applications could potentially be derived, by combining the results from the user survey with concrete engineering processes obtained from our previous experiences of creating exemplars.

- A possible research direction on separating UIs from data and logic program. This potentially supports building MUI for applications, we believe, mainly because UI designers would not worry about integration of MUI with the back-end programs. The potential technologies include XSL[W3C XSL] and XAML[Microsoft XAML]

- An idea of creating an advisory system. Such a system would suggest possible widgets and interaction means that are suitable for perform a particular task on a particular type of device. The suggestions provided by the system wrap the knowledge of several factors including the current task being designed, the targeted device constraints and facilities, and usability of UIs. It is better than automatic generating UIs that bypass user desires and easier than the existing MUI development framework that expects users to specify all the details.

8.4 General Summary

In this thesis, we began with an overview of the background technology and a comprehensive review of the related research into support for constructing MUI of ubiquitous applications. From this review, we formulated the key requirements for providing effective tool support for creating high quality advanced MUI and set up our long-term goal and the more specific goal for this thesis. Based on this context, we then discussed the concrete requirements for Pounamu/Mobile for providing advanced mobile thin-client UIs for any Pounamu specified diagramming tools. Next in the Analysis Chapter, we presented our approaches addressing these requirements respectively and described the requirement analysis and UI designs performed by us, to indicate the process of mapping people's mental designs to the real UI designs and the possibility to automate it. In the design and implementation chapter, we discussed in detail the issues involved in implementing the Pounamu/Mobile and generalized our approaches to be integrated in MUPE platform. Later we presented an evaluation of Pounamu/Mobile, where we felt our approaches have improved the quality and usability of mobile UIs. Finally we concluded our contributions of the thesis and outlined potential future work.

Bibliography

1. AAAI-04 Workshop on Spatial and Temporal Reasoning

2. Abrams, M., Phanouriou, C., Batongbacal, A.L., Williams, S.M., and Shuster, J.E. "UIML: An Appliance-Independent XML User Interface Language," in *The Eighth International World Wide WebConference.* 1999. Toronto, Canada: http://www8.org/ and http://www.uiml.org/.

3. Ali, M. F. and Pérez-Quinõnez, M., *Using Task Models to Generate Multi-Platform User Interfaces while Ensuring Usability*, accepted as interactive poster for CHI'2002, Minnesota, USA

4. Baudisch, P., Xie, X., Wang C., and Ma, W.Y.: Collapse-to-Zoom: Viewing Web Pages on Small Screen Devices by Interactively Removing Irrelevant Content. In Proc.UIST '04.

5. Björk, S., Holmquist, L.E., Redström, J., Bretan, I., Danielsson, R., Karlgren, J., and Franzén, K. WEST: a web browser for small terminals. In *Proc. UIST'99*, pp. 187–196.

6. Burnett M, Goldberg A, Lewis T (eds) *Visual Object-Oriented Programming,* Manning Publications,Greenwich, CT, USA, 1995.

7. Buyukkoten, O., Garcia-Molina, H., Paepcke, A., and Winograd, T. Power browser: efficient web browsing for PDAs. In *Proc. CHI'00*, pp. 430–437.

8. Buyukkokten, O., Garcia-Molina, H., Paepcke, A. Text Summarization for Web Browsing on Handheld Devices. *Transactions on Inf. Sys. 20*(1):82-115.

9. Chen, L.Q., Xie, X., Ma, W.Y., Zhang, H.J., Zhou H.Q., Feng H.Q., DRESS: A Slicing Tree Based Web Representation for Various Display Sizes. *Microsoft Research Technical Report MSR-TR-2002-126.*

10. Chen, Y., Ma, W.Y., and Zhang, H.J. Detecting Webpage Structure for Adaptive Viewing on Small Form Factor Devices. In *Proc. WWW'03*, pp 225–233.

11. Cao, S. Grundy, J.C., Hosking, J.G., Stoeckle, H. and Tempero, E. An architecture for generating web-based, thin-client diagramming tools, In Proceedings of the 2004 IEEE International Conference on Automated Software Engineering, Linz, Austria, September 20-24, IEEE CS Press, pp. 270-273.

12. Cypher, A. and Smith, D.C., KidSim: End User Programming of Simulations, In *Proceedings of CHI'95*, Denver, May 1995, ACM, pp. 27-34.

13. Eisenstein, J. and Puerta, A. (2000): Adaptation in automated user-interface design, Proc. 2000 Conference on Intelligent User Interfaces, New Orleans, 9-12 January 2000, ACM Press, pp. 74-81.

14. Evaluation of visual balance for automated layout, IUI2004

15. Ferguson R, Parrington N, Dunne P, Archibald J, Thompson J, MetaMOOSE-an object-oriented framework for the construction of CASE tools: Proc Int Symp on Constructing Soft. Eng. Tools (CoSET'99) LA, May 1999.

16. Gajos, K, and Weld, D., "SUPPLE: Automatically Generating User Interfaces" in Proceedings of IUI'04. Funchal, Portugal, 2004

17. Gordon, D., Biddle, R., Noble, J. and Tempero, E. A technology for lightweight web-based visual applications, In Proceedings of the 2003 IEEE Conference on Human-Centric Computing, Auckland, New Zealand, 28-31 October 2003, IEEE CS Press.

18. Grundy, J.C. and Zou, W. (2002): An architecture for building multi-device thin-client web user interfaces, *Proc. 14th Conference on Advanced Information Systems Engineering*, Toronto, Canada, May 29-31 2002, Lecture Notes in Computer Science

19. IBM, AUIML(Abstract User Interface Markup Language), 2004, http://www.alphaworks.ibm.com/tech/auiml

20. IBM Corp, Rational Rose XDE Modeler, http://www-306.ibm.com/software/awdtools/developer/modeler/

21. IBM Corp, IBM Transcoding Technology, www.research.ibm.com/networked_ data_systems/transcoding/

22. Kelly, S., Lyytinen, K., and Rossi, M., Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, in *Proceedings of CAiSE'96*, LNCS 1080, Springer-Verlag, Crete, Greece, May 1996.

23. Khaled, R., McKay, D., Biddle, R. Noble, J. and Tempero, E., A lightweight web-based case tool for sequence diagrams, In Proceedings of SIGCHI-NZ Symposium On Computer-Human Interaction, Hamilton, New Zealand, 2002.

24. J. Lin and J. A. Landay. Damask: A tool for early-stage design and prototyping of multi-device user interfaces. In Proceedings of The 8th International Conference on Distributed Multimedia Systems (2002 International Workshop on Visual Computing), pages 573–580, 2002.

25. Johnson, P., and Wilson, S. 1993 A framework for task-based design, in Proceedings of VAMMS 93

26. Mackay, D., Biddle, R. and Noble, J. A lightweight web based case tool for UML class diagrams, In Proceedings of the 4th Australasian User Interface Conference, Adelaide, South Australia, 2003, Conferences in Research and Practice in Information Technology, Vol 18, Australian Computer Society.

27. McWhirter, J.D. and Nutt, G.J., Escalante: An Environment for the Rapid Construction of Visual Language Applications, *Proc. VL '94*, pp. 15-22, Oct. 1994.

28. Mehra, A. Grundy, J.C. and Hosking, J.G. Supporting Collaborative Software Design with a Plug-in, Web Services-based Architecture, In Proceedings of the ICSE 2004 Workshop on Directions in Software Engineering Environments, May 25 2004, Edingurgh, Scotland, IEE Press

29. Microsoft, XAML, 2004, http://xml.coverpages.org/ms-xaml.html

30. Milic-Frayling, N. and Sommerer, R. SmartView: Enhanced Document Viewer for Mobile Devices. *Microsoft Research Technical Report MSR-TR-2002-114*.

31. MicroMedia www.micromedia.com

32. M. Minas and G. Viehstaedt, DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams, *Proc. VL '95,* 203-210 Sept. 1995.

33. Multiple User Interfaces: Cross Platform Applications and Context-Aware Interfaces, Ahmed Seffah and Homa Javahery (Eds), John Wiley & Sons, Chichester, England; 2004.

34. Myers, B.A., "The Amulet Environment: New Models for Effective User Interface Software Development*," IEEE TSE*, vol. 23, no. 6, 347-365, June 1997.

35. Meyer, N., "Finite State Machine Tutorial" in Generation5, http://www.generation5.org/content/2003/FSM_Tutorial.asp

36. Nichols, J., B. A. Myers, M. Higgins, J. Hughes, T. K. Harris, R. Rosenfeld, and M. Pignol. Generating remote control interfaces for complex appliances. In CHI Letters: ACM Symposium on User Interface Software and Technology, UIST'02, Paris, France, 2002.

37. Nichols, J., "Automatically Generating User Interfaces for Appliances" in *Advances of Pervasive Computing* edited by Alois Ferscha, Horst Hortner, and Gabriele Kotsis. April 18, 2004.

38. Nokia Corp, MUPE, www.mupe.net

39. Olsen Jr., D.R. "A Programming Language Basis for User Interface Management," in *Proceedings SIGCHI'89: Human Factors in Computing Systems.* 1989. Austin, TX

40. Oracle Corp. 1999, Oracle Protal-to-go

41. Palm Corp 2001, Web Clipping services, www.palm.com

42. S. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A service framework for ubiquitous computing environments. In Proceedings of Ubicomp 2001, pages 56–75, 2001.

43. Puerta, A., Eisenstein, J. "XIML: A Common Representation for Interaction Data," in *7th International Conference on Intelligent User Interfaces.* 2002. San Francisco: pp. 214-215.

44. Puerta, A., Eisenstein, J. "XIML: A Universal Language for User Interfaces", RedWhale Software 2003, www.ximl.org

45. Quasar Electronics, Visual Spice http://www.quasarelectronics.com/product-files/vs/visual_spice_support.htm

46. Sukaviriya, P., Foley, J.D., and Griffith, T. "A Second Generation User Interface Design Environment: The Model and The Runtime Architecture," in *Proceedings INTERCHI'93: Human Factors in Computing Systems.* 1993. Amsterdam, The Netherlands: pp. 375-382.

47. Sipser, M., Introduction to the theory of computation, MIT, Cambridge, PWS Pub. Co., c1997

48. Szekely, P., Luo, P., and Neches, R. "Beyond Interface Builders: Model-Based Interface Tools," in *Proceedings INTERCHI'93: Human Factors in Computing Systems.* 1993. Amsterdam, The Netherlands: pp. 383-390.

49. Vander Zanden, B. and Myers, B.A. "Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces," in *Proceedings SIGCHI'90: Human Factors in Computing Systems.* 1990. Seattle, WA: pp. 27-34.

50. W3C, Device Independence Group, http://www.w3.org/2001/di/Activity.html

51. W3C, XHTML, www.w3.org/TR/xhtml1/

52. W3C, XSL, http://www.w3.org/Style/XSL/

53. W3C, SVG, http://www.w3.org/TR/SVG/

54. W3C, SVG MOBILE, www.w3.org/TR/SVGMobile/

55. *WAP:* *www.wapforum.org*

56. Weld, D., Anderson, C., Domingos, P., Etzioni, O., Gajos, K., Lau, T., Wolfman, S. “Automatically Personalizing User Interfaces,” in *Eighteenth International Joint Conference On Artificial Intelligence.* 2003. Acapulco, Mexico

57. Wiecha, C., Bennett, W., Boies, S., Gould, J., and Greene, S., “ITS: A Tool for Rapidly Developing Interactive Applications.” *ACM Transactions on Information Systems*, 1990.8(3): pp. 204-236.

58. Wobbrock, J., Forlizzi, J., Hudson, S., Myers, B. WebThumb: interaction techniques for small-screen browsers. In *Proc.UIST '02,* pp. 205–208.

59. Xamlon Icn, www.xamlon.com

60. XMLToAny, 2001, www.javazoom.net/jzservlets/xmltoany/xmltoany.html

61. XULPlanet: www.xulplanet.com

62. Zhang, K., Zhang, D-Q., and Cao J., Design, Construction, and Application of a Generic Visual Language Generation Environment", *IEEE TSE*, 27,4, April 2001, 289-307.

63. Zhu, N., Grundy, J.C. and Hosking, J.G., Pounamu: a meta-tool for multi-view visual language environment construction, In Proceedings of the 2004 International Conference on Visual Languages and Human-Centric Computing, Rome, Italy, 25-29 September 2004, IEEE CS Press, pp. 254-256.

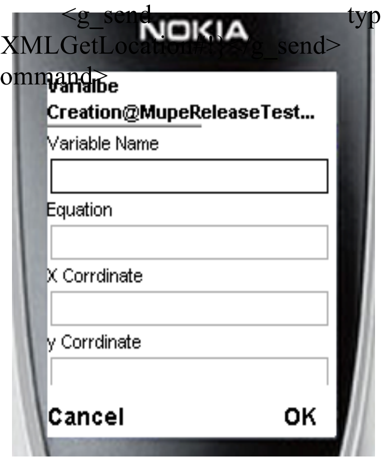
i) Element Management Tree

Figure 2-6 (1) A simple Equation-based Business Modelling Tool, composed of three types of elements: Variables, Parameters, and Connectors. Selected UIs show (1) creating a variable element, (2) element management tree, (3) equation graph view. *(2)* *(3)*

```
template type='form' title='!#XMLGetName#!' id=''>
        <command type='screen' text='Back'><g_send type='request'>poll</g_send></command>

        !#XMLGetTreeContents#!

        <command type='screen' text='Create a entity Variable'>
                <g_send type='request'>create{Variable}</g_send>
        </command>

        <command type='screen' text='Create a entity Param'>
                <g_send type='request'>create {Param}</g_send>
        </command>

        <command type='screen' text='Create a connector'>
                <g_send type='request'>create {BaseConnector}</g_send>
        </command>

        <command type='screen' text='Get out'>
                <g_send                    type='request'>move                    {$(service.username)}
        {!#XMLGetLocation#!}</g_send>
        </command>
```

Appendix A: MUPE Application Example

Following shows the Equation-based Business Modelling Tool developed on MUPE platform. We show the same MUPE UIs again as in the chapter 2 for your convenience to compare the object visual appearances with the corresponding XML UI descriptions in the followings.



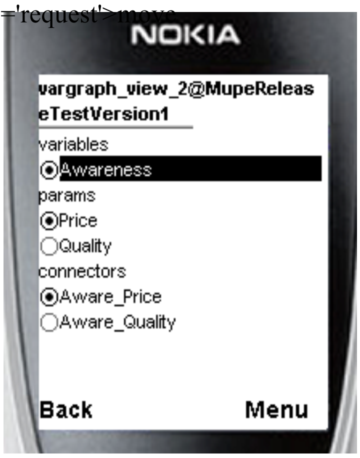
Varialbe Creation@MupeReleaseTest...

Variable Name

Equation

X Corrdinate

y Corrdinate



vargraph_view_2@MupeReleaseTestVersion1

variables

● Awareness

params

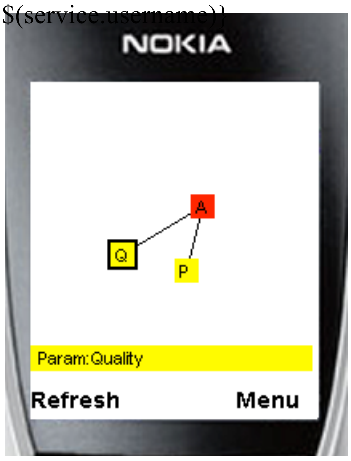
● Price

○ Quality

connectors

● Aware_Price

○ Aware_Quality



Param:Quality

(2) Variable Object Creator

```
<template type='form' id='' title='Variabe Creation'>
	<editfield title='Variable Name' id='varname' />
	<editfield title='Equation' id='equation' />
	<editfield title='X Corrdinate' id='x' />
	<editfield title='y Corrdinate' id='y' />
	<command type='cancel' text='Cancel'><g_send type='request'>poll</g_send></command>
	<command type='ok' text='OK'>
		<g_send type='request'>
			!#XMLGetID#!::answerCreator {$(varname.text)} {$(equation.text)} {$(x.text)} {$(y.text)}
		</g_send>
	</command>
</template>
```

(3) Equation Graph View

```
<template type='canvas' id='maincanvas' uigroup='rpg_game' bgcolor='white' style='color' active='true'>
	<command type='screen' text='Refresh'><g_send type='request'>poll</g_send></command>
	!#XMLGetContents#!
	!#/statusbar.xml#!
	<g_allowuserinput value='true' />
	<command type='screen' text='edit'>
		<g_send type='request'>$(statusbar_bg.selectedId)::getEditor </g_send>
	</command>

	<command type='screen' text='delete'>
		<g_send type='request'>removeObject {$(maincanvas.statusbar_bg.selectedId)} </g_send>
	</command>

	<command type='screen' text='Create a entity Variable'>
		<g_send type='request'>create {Variable}</g_send>
	</command>

	<command type='screen' text='Create a entity Param'>
		<g_send type='request'>create {Param}</g_send>
	</command>

	<command type='screen' text='Create a connector'>
		<g_send type='request'>create {BaseConnector}</g_send>
	</command>

	<command type='screen' text='View as a tree'>
		<g_send type='request'>!#XMLGetID#!::getTreeDescription </g_send>
	</command>

	<command type='screen' text='Get out'>
		<g_send type='request'>move {$(service.username)} {!#XMLGetLocation#!}</g_send>
	</command>
</template>
```