Thin-client user interface design for the Pounamu meta-CASE tool

Shuping Cao

2005

Abstract

Traditional thick-client diagramming tools have common problems such as installation and setup overhead, complex and difficult to learn user interfaces, and a heavy infrastructure to support multi-user collaborative work. These are some of the reasons why tools of this calibre fail to meet expectations.

Thin-client design tools are immune to the problems mentioned above. They provide a number of advantages over traditional thick-client tools, including a simple and easy to learn user interface, inherent support for collaborative work, and server-side integration with legacy systems or facilities. However, building thin-client diagramming tools is a challenging task. Instead of constructing a complete new tool and architecture for thin-client diagram editors, we propose an extension (namely, Pounamu/Thin) to a thick-client meta-CASE tool we have been developing which allows any specified diagram editor to be realized as a thin-client tool. A combination of a web services interface to the thick-client tool, a set of server-side components, diagram format converters and a conventional web browser are used to provide these thin-client diagramming applications.

In this thesis, we provide the motivation for our research on Pounamu/Thin. Related diagramming tool research is reviewed, which defines the basis for our work. Object-oriented analysis and design are carefully performed, which is the prerequisite for a good quality of the system. We also discuss issues related to implementing Pounamu/Thin and demonstrate examples of use of the prototype tool. Finally, we describe evaluations of the effectiveness of our approach and identify potential enhancements to our work.

Acknowledgements

First and Foremost I would like to express gratitude to my two supervisors, John Grundy and John Hosking, for their encouragement, support, and guidance. Throughout my work period, they provided me with great ideas, sound advice, and good teaching. Without their help my work would have been much the poorer.

I would like to thank all my student colleagues who helped and advised me during my work on this project.

Thanks also to those people who attended my evaluation survey. The time and energy they spent to provide their well-considered feedback is much appreciated.

Finally, special thanks to my husband, Jun Zhuang, who always cares and loves me for all these years. I am grateful for his encouragement during many difficult times.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

This chapter presents the rationale for this project and briefly describes the approaches taken to fulfill its goals. An overview of the structure of this thesis is also given in the last section.

1.1 Motivation

Most traditional diagramming tools, such as CASE tools, CAD tools and user interface design tools, use a thick-client, desktop interface and architecture. As they can leverage sophisticated thick-client interaction techniques and typically manage information on local PCs in a distributed fashion [5, 7,11], these tools generally work very well in terms of providing highly responsive diagram editing and viewing facilities. However, there are some disadvantages that keep these thick-client, standalone tools from becoming successful. First, they often provide complex and difficult to learn user interfaces. Second, they need heavyweight architectures to support collaborative editing. Most importantly, they have to be installed and periodically updated on every user's PC.

With the rise of the web and scripting languages, many researchers have tried to address the disadvantages of thick-client diagramming tools by developing web alternatives. However, due to the poor visual interaction supported by web browsers, most earlier work in this area resorted to augmentation technologies such as Java Applets or similar thick-client browser plug-ins [7]. Web-based diagramming tools built in this way are, in essence, indistinguishable from traditional thick-client tools.

Some recent efforts [6,12,14] prove that it is possible to build web-based diagramming tools without the help of JavaApplets or complex browser plug-ins. In these lightweight tools, users view diagrams as the contents of a "web page" along with buttons, links etc for modifying the diagram, moving to other diagrams and so on. Diagrams may be constructed with combinations of automatic and manual layout, resizing, highlighting and so on. Technologies to render the diagrams might include GIF images, SVG renderings, VRML visualizations and so on. Key potential advantages of this approach are a consistent look and feel across all web-based diagramming tools, use of web page design techniques to aid interaction and learning, no need to install and update copies of tools on PCs, and use of conventional web architectures to support collaborative work by multiple users.

We have been developing a new thick-client meta-CASE tool (called Pounamu) which provides very flexible diagram and meta-model specification techniques and a thick-client editing approach. Instead of building a complete new tool and architecture for thin-client diagram editors, we chose to build an extension to this meta-CASE tool which supports a thin-client diagramming infrastructure. One copy of the meta-tool is used as an application server, holding shared diagrams and model data. A set of web server components interact with this server via a web service-based interface. These web server components provide diagram format translation, editing, view management and other facilities to users. Web browsers deployed on each user's PC render the diagrams and capture user input for property editing, diagram editing, view management and so on.

1.2 Our Approach

We aimed at designing and developing a thin-client diagramming tool. The research approach that we took to achieve this goal is described below.

First, we performed a comprehensive literature review on this research field. In this review, we investigated the features and characteristics of a variety of web-based software engineering tools, and especially the approaches taken to develop web-based diagramming tools. Based on this investigation we formed our own ideas for providing a web component extension (called Pounamu/Thin) to the original Pounamu Meta-CASE tool. Many useful ideas and insights come from the existing approaches to support web-based diagramming facility.

After the literature review we defined more accurately the goals of our work, and identified system requirements for the thin-client visual tool. The essential requirements are:

Support for thin-clients: This system can be deployed on any client machine running a suitable web browser, so that users can access the modeling environment of Pounamu meta-CASE tool without the need to download/install software. Additionally, this allows developers to upgrade the server-resident systems more easily. Compared with the original Java Swing-based Pounamu tool, our system should provide an easy-to-use and easy-to-learn user interface, so that novice users can pick up the Pounamu tool fairly easily.
Consistency: In addition to supporting all the core functionalities enabled in the original Pounamu tool, our thin-client extension should adopt a design approach consistent to other web-based applications, i.e., using HTTP POST/GET page display metaphor for diagram viewing and interaction.
Friendly and intuitive user interfaces: The obstacle to implementing web-based visual tools is that web browser support for visual interaction is very poor. This means that a simple operation such as moving diagram components often involves multiple interactions to complete. In order to equip our system with comparatively friendly, intuitive user interfaces, special care should be given to avoid unnecessary user interactions.
Integration: Our Pounamu/Thin extension should be easily plugged into and off the original Pounamu tool given a user's choice to use either the thick-client or the thin-client user interface for their diagramming tasks. This should be achievable with the help of component-based solutions.

With the above key requirements in mind, we created a theoretical basis for our tool. This consists of a sound software architecture, a number of design objects and methods to perform the core functionalities, as well as graphical user interfaces via which users can interact with our system. A general idea of how different our thin-client visual tool is from its corresponding thick-client version can be obtained from Figure 1-1.

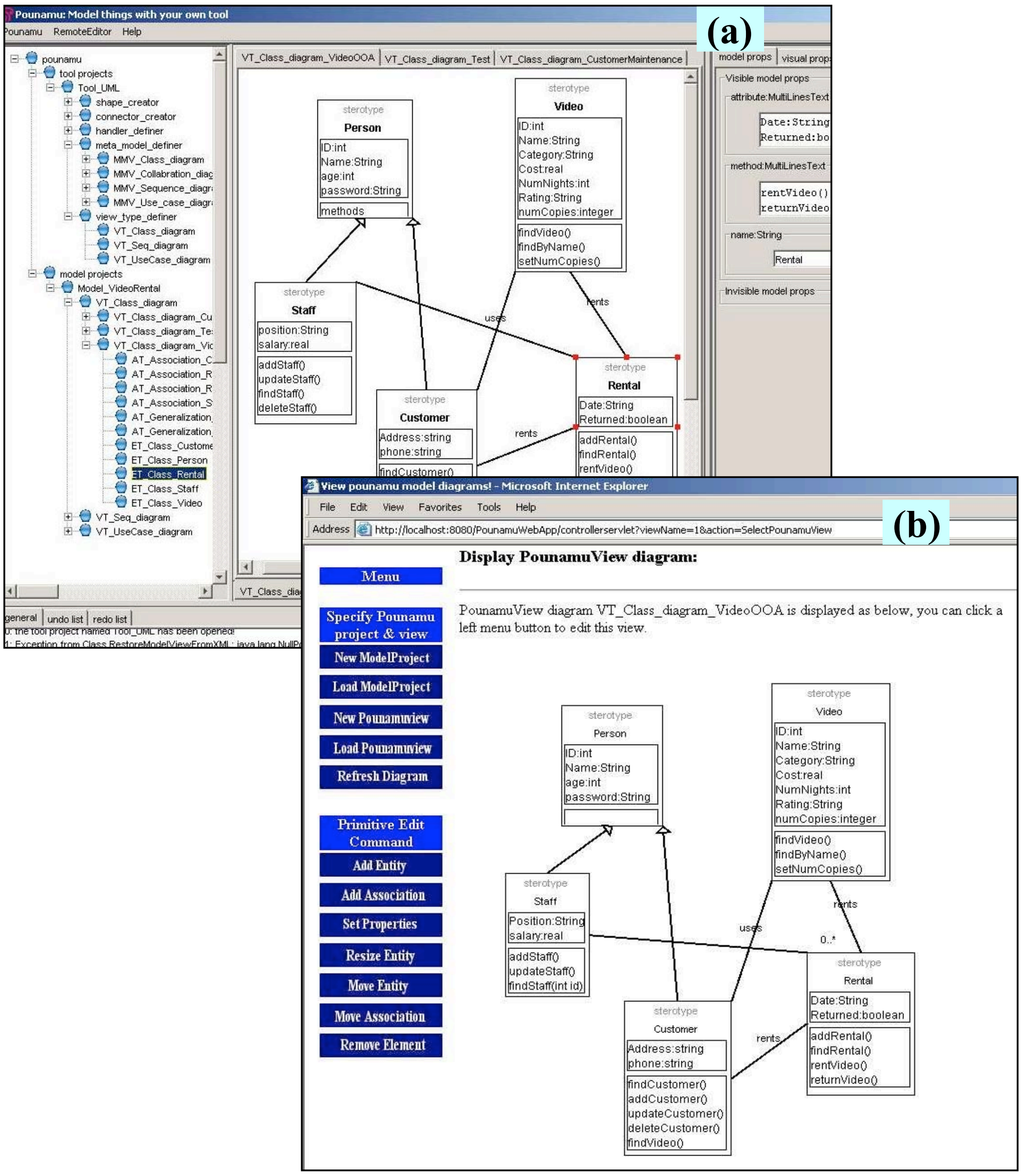


Figure 1-1 The main interface of the thick client Pounamu tool (a) vs. its thin-client counterpart (b)

The next step was to build prototype systems from the theoretical basis formed previously. Several components were implemented during this process including:

- Remote editing component plug-in

- SVG-version web component
- GIF-version web component

The final stage of our work was to carry out the evaluation of our prototype tools to investigate system usability issues. The tasks performed in this evaluation were:

- Describe goals of the evaluation
- Determine the features of the prototypes that we were interested to test
- Compare these features along a variety of cognitive dimensions
- Prepare survey tasks to test these features
- Recruit subjects to participate in the survey
- Analyze the results

1.3 Thesis Overview

This thesis is organized as follows:
Chapter 1: Introduction
Chapter 2: Background and Related Research--covers the background technology and examines some related work in this topic area.
Chapter 3: System Requirements--focuses on developing system functional requirements and non-functional constraints.
Chapter 4: System Design--constructs the underlying software architecture and presents a more detailed design for the main system components.
Chapter 5: Implementation of the GIF Prototype--describes implementation of our GIF version thin-client diagramming tool and the techniques we used in doing so. Discusses some of the basic implementation issues we had to consider in the development of this prototype.
Chapter 6: Implementation of the SVG Prototype--describes implementation of our SVG version thin-client diagramming tool and the techniques used to assist this implementation.
Chapter 7: Evaluation--evaluates our GIF and SVG prototypes. Discusses the strengths and weaknesses of our tools compared against the original Pounamu thick-client tool, as well as other thick-client and thin-client diagramming tools. Presents the results of applying cognitive dimensions framework to our prototypes.
Chapter 8: Conclusion and Future Work--Presents the conclusions and summarizes the avenues of future work indicated by this work.

Chapter 2: Background Knowledge and Related Research

The chapter presents some background knowledge and past work related to our research. The whole chapter has been organized as 4 sections. The first section describes what CASE and Meta-Case tools are, and the discussion is focused on our Pounamu Meta-CASE tool. In the second section, we examine the reasons why increasing numbers of researchers want to bring their software engineering tools to the web and explore work done in this area. Section 3 describes more recent efforts in developing web-based thin-client diagramming tools and comments on the strengths and weaknesses of approaches taken by these tools. The final section presents an overview of our approach employed to develop the Pounamu-based thin-client diagramming extension.

2.1 CASE and MetaCASE tools

2.1.1 General Information

Computer Aided software engineering (CASE) tools are software applications that support a variety of software engineering activities, including analysis, design-level modelling, implementation, maintenance, process modeling, management, testing and documentation etc. Examples of representative CASE tools include Rational Rose™ [16], Simply Objects™ [25], Argo/UML [17], MOOSE [24], and the JComposer CASE tool [9]. Originally designed to provide a variety of disparate analysis and design methods and notations, most CASE tools are now attempting to adopt standardized modeling methods due to the emergence and popularity of UML. However, despite all these efforts, CASE tools are generally not as successful as was expected. A major limitation is that these traditional tools only allow the use of a certain, fixed method for all problems which may or may not belong to the same domain. As a result, users may find the models difficult to understand because they are unfamiliar with the method, or they may feel it difficult to map the concepts and semantics used in the models to their application domain [23]. To address this limitation, Meta-CASE technologies have been developed.

Meta-CASE tools provide a means by which users are enabled to build their own customized CASE tools in a very low cost effective way. In order to do this, Meta-CASE tools contains an extra metamodel level compared with a typical two-level architecture used in the traditional CASE tools (illustrated in Figure 2-1). Thus, instead of hard-coding a method, as in a fixed CASE tool, Meta-CASE tools store all the method related information into a metamodel. The metamodel in turn can be modified to support the Meta-CASE tool's tailorability. Examples of Meta-CASE tools that have been developed in recent years include MetaEDIT+ [11], KOGGE [4], JViews [9], MetaMOOSE [5], Synthesizer Generator [26], and Dora [27]. In general, they all support facilities such as defining tool data structures and views on the data structure, providing input/output specification techniques, and generating some or all of the code to implement the specified tool [22].

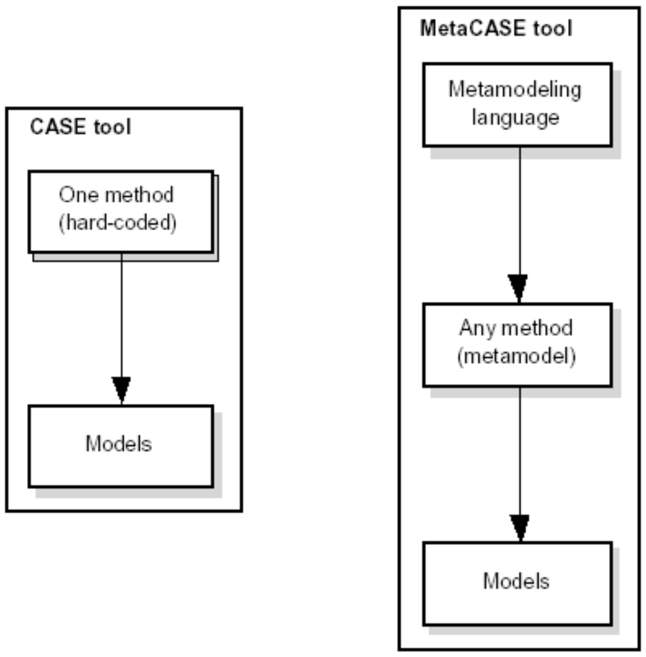
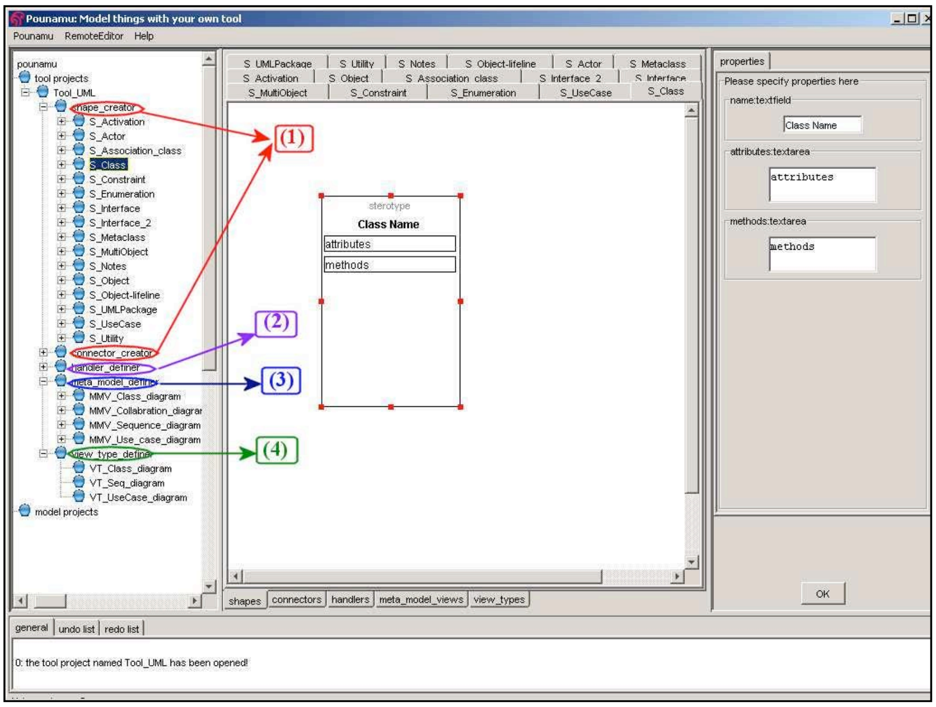


Figure 2-1 CASE Tool versus MetaCASE Tool (Adapted from [23])

2.1.2 The Pounamu Meta-CASE Tool

Pounamu is a metaCASE tool developed at the University of Auckland. Compared with other Meta-CASE tools named previously, Pounamu has three distinct features. First, the separation between meta-model definer and meta-model elements' view definer provides the basis for Pounamu's multi-view and multi-view type support. Secondly, the introduction of event handlers greatly improves the ability to specify behavioural aspects of Pounamu's visual languages. For example, with this facility users can define any piece of java code which they wish to be executed dynamically, and may also enforce constraints in modeling. Thirdly, the adoption of a universal file standard (XML) enables Pounamu to be easily integrated with some other third-party software tools. As the objective of this project is to design and implement the thin-client user interface for Pounamu, a detailed description of the internal structure of Pounamu, as well as examples of this tool in use, is presented below.

Pounamu is a Java Swing-based software engineering tool. It can be used to model software engineering projects, as well as build user-customized tools. Pounamu consists of five major sub tools, which are icon creator, meta-model definer, view type definer, event handler creator, and modeler. While the first four are used to specify a tool, the last one is used in the software modelling process with the defined tool. Figure 2-2 shows some of tool design facilities from Pounamu in use. These include a shape designer (1), event handler definer (2), meta-model designer (3) and view designer (4).



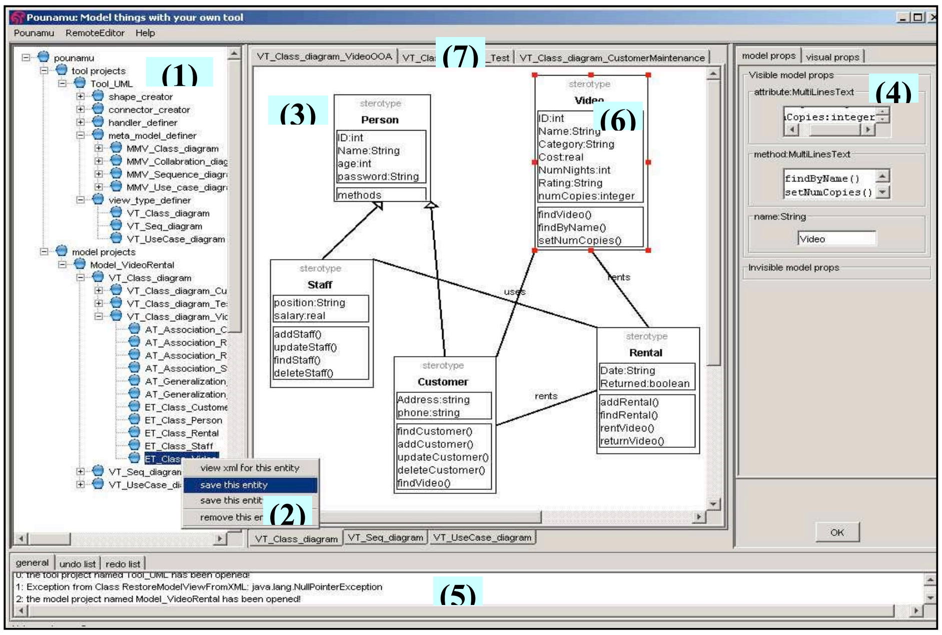
0: the tool project named Tool_UML has been opened!

Figure 2-2 the Pounamu Meta-tool Designer Tool

The functionalities of these components, with references to the above picture, are summarized as below:

1. Shape designer: can be used to design the appearance of shapes and connectors
2. Meta-model designer: can be used to define meta-model elements
3. View designer: can be used to map meta-model elements to their corresponding visual representation
4. Event Handlers: can be used to specify handlers to perform tasks.

Seeing that all the actual modeling work is done in the modeller, it is one more important sub-tool in the Pounamu environment. In this modeller tool, we create instances of the element type defined in a meta-model designer tool. These instances are visually represented as certain shapes or connectors following mapping operations defined in the view type definer tool. Additionally, event handlers are executed to perform certain tasks when the events specified in the handler creator take place. To wrap up our discussion, we present a screen shot (Figure 2-3) which shows an example diagramming tool, a Unified Modelling Language (UML) CASE tool, generated by Pounamu being used in the modeller.



```
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 A Pounamu-specified UML design tool in use

2.2 Web-based Software Engineering Tools

2.2.1 Taking software engineering tools to the Web

Like other diagramming tools, such as CASE tools, CAD tools and user interface design tools, the traditional thick-client program approach has been used to design and develop our Pounamu metaCASE tool. For example, in Figure 2-3, you can see thick-client user interface widgets, including elements management tree (1), pop-up or pull-down (2) menus, drawing canvas (3), property editor (4), status window (5), and directly-manipulable shapes (6) and tabbed-pane (7). It is only with these widgets that users are able to make direct interaction with a variety of tools and diagrams in Pounamu. There is no doubt that diagramming tools in this thick-client category provide key advantages such as highly responsive interaction when directly manipulating diagrams and the ability for tool developers to make use of all the graphics facilities available on the host PC. But we also notice that these traditional diagramming tools suffer from some of the common problems of thick-client applications. For example, they often provide complex and difficult-to-learn interfaces; they have to be installed and periodically updated on all host computers, 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.

With the rise of the web and scripting languages, web-based user interfaces have become more pervasive in recent years. Many applications that traditionally used thick-client interfaces have often developed thin-client versions. Examples of these applications 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. We decided to experiment with thin-client diagramming support for Pounamu-based diagramming applications to try and leverage these advantages in our tools. Our aim was to provide an optional extension to Pounamu that allows users to choose to access generated diagramming tools via a web-based, thin-client infrastructure instead of its conventional thick-client approach. Before detailing our approach to achieve this aim, it is necessary to investigate past research in developing web-based software engineering tools.

2.2.2 Research Examples

Since the web's appearance in the early '90s, various web-based software engineering tools have been developed to exploit the advantages of web-based information delivery. Some examples include MILOS [15,16], a web-based process management tool, BSCW [1], a shared workspace system, Rosetta [7], which supports web-based collaboration on software design, Web-CASRE [13], which provides software reliability management support, Ukase [20], a web-based use case management tool, CHIME [3], which provides a hypermedia environment for software engineering, and WikiWikiWeb [20], an editable hypertext system. In order to assist in understanding how these web-based tools are designed, we present below a brief description of some example applications.

MILOS

MILOS is a web-based process management tool. By integrating workflow technologies and project management over the Internet, it provides dynamic collaboration of software development team members at different locations. MILOS's architecture takes a three-tier approach, and the graphical user interfaces on the client tier is used for planning and enactment. When the system is running, this graphical user interfaces needs to be dynamically loaded via a web server and executed as applets within a web browser or as standalone applications.

BSCW

BSCW is a shared workspace system. It is meant to provide a means of supporting collaborative work for widely dispersed work groups, especially those involved in large research and development projects. The system consists of a server which maintains a number of workspaces. Each workspace is accessible via standard web browsers. Currently, the shared objects contained in a workspace can only be documents, links (to normal web pages and other workspaces), folders, groups and members.

Rosetta

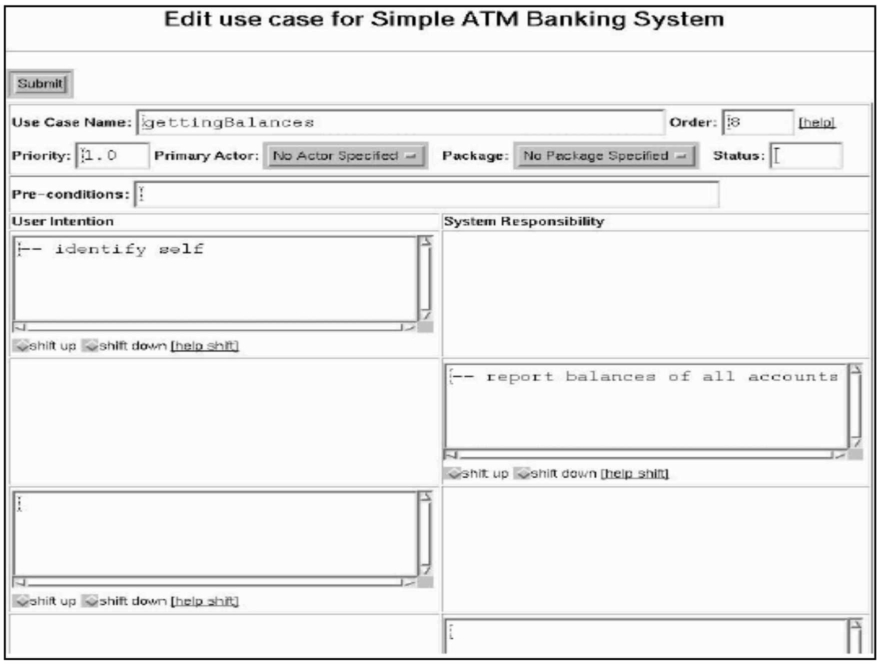
Rosetta supports collaborative software design by using a novel WWW architecture. In order to provide the easy creation, editing and viewing of design documents using web browsers, Rosetta design documents are written in HTML with design diagrams embedded through a JavaScript tag. While HTML design documents can be constructed using any HTML editor, the embedded UML diagrams have to be created with a special tool called Rosetta Editor. Since this editor is implemented as a Java Applet, it can only be invoked from web browsers supporting the Java Plug-in.

Web-CASRE

Web-CASRE [13] is a tool that provides software reliability management support, and it was extended from the original standalone CASRE tool. To make the original tool available in the web, its graphical user interface and problems solver that mingled together has to be separated, i.e., the GUI runs on the client side, while the computationally intensive solver remains on the server. In Web-CASRE, Tcl/Tk code used to implement the GUI has been embedded into a web page via an HTML tag. When this HTML page is visited the web browser will automatically load and install the Tcl/Tl plug-in required to run the client side user interface.

Ukase

Ukase is a light-weight web-based application built to manage essential use cases [21]. Originally designed to make the recording of use case models easier, Ukase provides mechanisms such as entering the details of essential use cases and displaying the results. Since its first appearance, Ukase has developed considerably. The latest version makes use of new web technologies such as JSP and JavaBeans etc., with the information stored in a MySQL database [28]. Figure 2-4 shows an input HTML form laid out the way that a Use Case would appear in the report. Developers will typically use a different view. Once the development is finished, the resulting use case model is available as a report.


Edit use case for Simple ATM Banking System

Use Case Name: gettingBalances
Order: 8
Priority: 1.0
Primary Actor: No Actor Specified
Package: No Package Specified
Status:
Pre-conditions:

User Intention	System Responsibility
-- identify self	
	-- report balances of all accounts

Figure 2-4 HTML form for entering the details of an Essential Use Case in Ukase (from [21])

CHIME

CHIME was built upon the efforts from both the User Interface and Hypermedia research communities to create a hypermedia like software development environment for managing information from all phases of the software lifecycle [3]. In the CHIME model, all sorts of project artifacts, including source code, design documentation, and email archives, still reside in their original locations, while CHIME provides data organization and hypertext linking capabilities for maintaining metadata such as the location and access information for the data. In doing so, CHIME generates a multi-user style virtual world in which users are allowed to freely gather information or communicate with others. In general, CHIME has a great potential in building applications for geographically and temporally dispersed users to perform their collaboration more efficiently.

After investigating the above web-based software engineering tools, we found most of them lack provision for any kind of graphical diagram construction and editing. Often their web-based interfaces are used for visualizing diagrams, such as UML diagrams derived from formal specifications as in TCOZ [20]. Those tools that do support a web-based diagramming facility always use Java Applets or similar thick-client browser plug-ins (e.g. Rosetta). This makes us assume that developing thin-client diagramming tools might be a challenging and difficult task. This assumption is reinforced by another discovery, that is, despite the fact that most CASE and Meta-CASE tools (e.g. Rational Rose and Meta-EDIT etc.) include a specification of the intended diagramming tool that might be interpreted to provide a thin-client version of the tool, none do so.

So what makes the development of thin-client diagramming tools a difficult task? Some possible problems are: the limited support provided by the web for interaction with diagrams; very rudimentary user interface toolkit (e.g. HTML form fields) available for use, and the “ pull-only” data transfer methodology used in HTTP/HTML. Many attempts have been made to address these restrictions, including sending entire applications over HTTP (e.g. Java Applets), and designing browser “plug-ins” that interpret users’ interaction according to its own language semantics (e.g. Flash, VRML and Shockwave). However, all these attempts to overcome the limitations give rise to some new problems. For example, with Java Applets, there are issues such as download speed and web browser’s compatibility with Java binary. Java Applets also raise security concerns because executable code has to be transported to the client via HTTP. Most importantly, the use of Java applets is often cited as against the thin-client principle of running most application logic on the server. Another problem that browser “plug-ins” suffer is that they are not widely available or supported by some old browsers.

Fortunately, some recent experiments prove that Java Applets and Browser “plug-ins” are not necessary to build thin-client diagramming tools. Further evaluation results show that thin-client diagramming tools created without using these augmentation technologies are reasonably easy to use although they offer less interactivity. The next section is focused on the discussion of these recent efforts.

2.3 Progress on Web-based Diagramming Tools

Some recent research work on building web-based diagramming tools have been attempting to avoid using Java Applets and any browser "plug-in", so that tools can be used on any computer as long as there is a standard web browser running. Examples of the newly developed thin-client diagramming tools following the above design principle include Seek [12], a UML sequence diagramming tool, NutCASE [14], a UML class diagramming tool, and Cliki [5], a thin-client meta-diagramming framework. In order to illustrate the basic ideas on how these tools get their work done, we prepare a short discussion for each of them below.

Seek

Seek is a sequence diagramming tool developed to maximize the accessibility and use by team members during the early part of the design process [12]. Seek takes a web-based but applet-free approach so that diagrams can be created, edited, or deleted via an ordinary browser. Figure 2-5 shows Seek's main user interface consisting of a sequence diagram being edited, tool facilities for editing and manipulation of the diagram, and image tiles that form the diagram. In Seek, image tilling is an important concept since it supports a simple, but effective means of diagram manipulation. As shown in Figure 2-5, a sequence diagram is represented as a two-dimensional grid of image tiles. Each tile is filled with one of a set of images that match particular UML states or actions. This set of images is displayed as palette on the left of the tool interface. The sequence diagram can be constructed by repeatedly clicking the appropriate palette image, and then clicking in the grid where that image should appear. If laid out in the correct configuration, this collection of small images can show a nice UML sequence diagram.

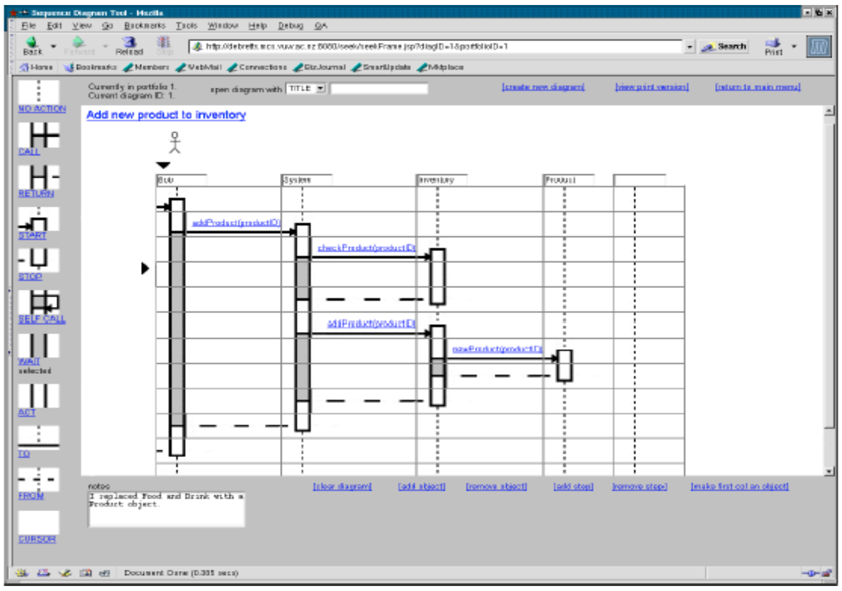

Sequence Diagram Tool - Mozilla
File Edit View Go Bookmarks Tools Window Help Debug QA
Back Forward Reload Stop
Search Print
Home Bookmarks Members WebMail Connections BizJournal SmartUpdate Mktplace
Currently in portfolio 1.
Current diagram ID: 1.
open diagram with TITLE
[create new diagram] [view print version] [return to main menu]
NO ACTION
CALL
RETURN
START
STOP
SELF CALL
WAIT
selected
ACT
TO
FROM
CURSOR
Add new product to inventory
Bob
System
Inventory
Product
addProduct(productID)
checkProduct(productID)
addProduct(productID)
newProduct(productID)
notes
I replaced Food and Drink with a Product object.
[clear diagram] [add object] [remove object] [add step] [remove step] [make first col an object]
Document Done (0.335 secs)

Figure 2-5 The main screen of Seek, showing a sequence diagram facilities for editing and manipulation of the diagram (adapted from [12])

NutCASE

NutCASE offers an environment in which UML class diagrams can be manipulated via the web, and types of manipulation provided include creation, deletion, editing and searching of classes [14]. As with the Seek tool, NutCASE uses some server-side technologies such as JSP and JavaBeans to accomplish most computations, and web browsers need only to display HTML pages with the dynamically generated GIFs embedded. An unusual aspect of NutCASE implementation is that all class diagram interactions use client-side image maps. The GIF image generation and image map are thus the keys for NutCASE to support web-based diagramming. Figure 2-6 shows an example of NutCASE in use. It can be seen that NutCASE provides most of core functionalities required to edit a class diagram, including adding a class, editing properties of a class, deleting a class and finding a class. However, no moving or resizing a class has been provided. So, in NutCASE, the layout of a diagram cannot be adjusted as users desire.

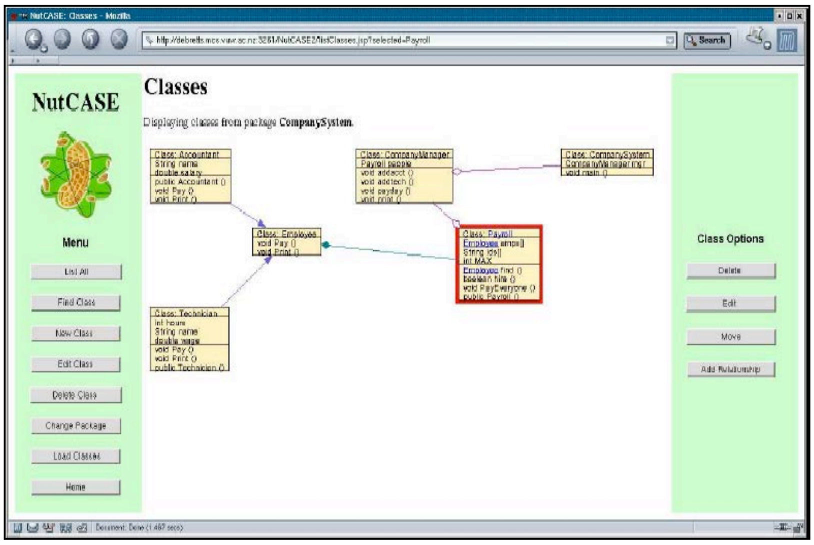


Figure 2-6 NutCASE: the main user interface for displaying a class diagram (Adapted from [14])

Cliki

Cliki is a framework used to design web-based visual applications [5]. It uses HTML image form fields to display images dynamically generated by the server. Since an image field is an input type for HTML forms, it can capture the coordinates of a mouse click and send this information to the server. The server responds according to the application logic, including highlighting the clicked icon or moving the clicked icon to a new position. Although the idea adopted in Cliki is very simple, its way of implementing thin-client diagramming tools has proved successful. For example, Cliki developers have created several experimental applications with this framework, including a diagram editor (Figure 2-7(a)), Sokoban game (b), and UML class diagram editor (c).

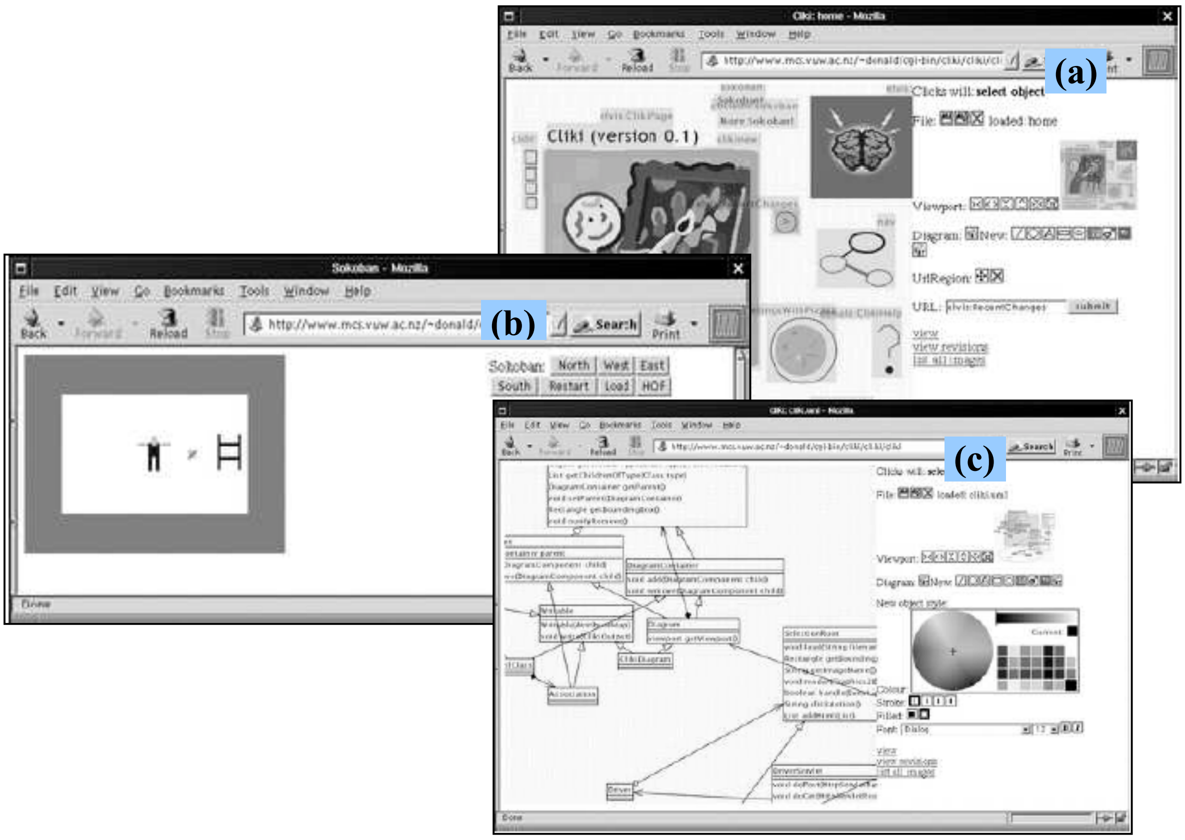


Figure 2-7 Example applications built with Cliki framework (adapted from [5])
(a) A diagram Editor Application (b) Sokoban game (c) UML class diagram Editor

From the above discussions, we can see that all three thin-client diagramming tools support reasonable visual interaction without resorting to the assistance of Java Applets and browser plug-ins. This in turn allows them to work on any modern web browser. Compared with the previously mentioned thick-client diagramming tools, these tools contain less complex functionality, thus making them easy-to-learn and use. Potential applications of these tools include teaching and learning software practices in a university.

However, despite the benefits, one disadvantage common to all these tools is that each uses a customized approach to realize the limited diagramming functionality. For example, with NutCASE, users are only allowed to draw very simple UML class diagrams because the server side application logic is programmed to do so; with Seek, users can only create and edit sequence diagrams expressed in UML notations because the left side image palette tool contains only visual representations of possible action states. Compared with NutCASE and Seek, Cliki, as a thin-client meta-diagramming tool, seems more advanced in diagrams that can be drawn and edited, and this has already been backed up by some practical examples (shown in Figure 2-7(a)-(c)) completed by the tool developers themselves. However, users who design visual applications with Cliki have to be very careful, as otherwise the resulting application would be too bothersome because large numbers of interactive clicks are required to perform any trivial operation. Additionally, Click has contained comprehensive functionalities and a complex user interface for novice users to pick up in a short time. In all, we feel the approaches taken to implement these thin-client diagramming tools are too customized, so that the resulting tools provide limited adaptability and integration support. From this point of view, we doubt the efforts and costs involved in developing these tools have really paid off.

2.4 Overview of Pounamu-based Thin-client Diagramming Tool

Our Pounamu meta-CASE tool was developed to provide a flexible, interactive diagramming tool specification. It was assumed that all generated diagramming tools would provide a thick-client user interface to tool users. In order to experiment with providing thin-client diagramming interfaces from these tool specifications, we proposed to develop an optional thin-client diagramming extension to Pounamu. The viability of this proposal is backed up by the fact that Pounamu is built using a component-based architecture [9], and software tools developed with this architecture are generally easier to enhance, extend and integrate with other tools. Figure 2-8 illustrates how this thin-client diagramming extension (we call Pounamu/thin) is fitted into Pounamu and used by software engineering practitioners who prefer the thin-client way of doing things.

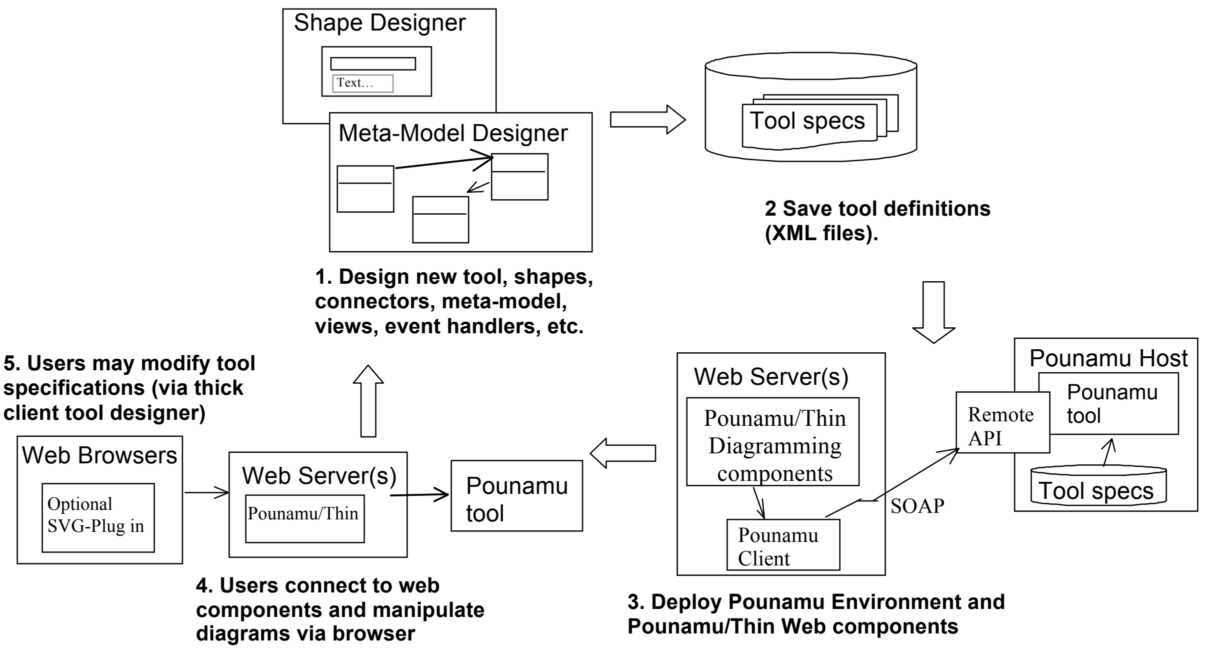

Shape Designer
Text...
Meta-Model Designer
Tool specs
2 Save tool definitions (XML files).
1. Design new tool, shapes, connectors, meta-model, views, event handlers, etc.
5. Users may modify tool specifications (via thick client tool designer)
Web Browsers
Optional SVG-Plug in
Web Server(s)
Pounamu/Thin
Pounamu tool
Web Server(s)
Pounamu/Thin Diagramming components
Pounamu Client
SOAP
Remote API
Pounamu Host
Pounamu tool
Tool specs
4. Users connect to web components and manipulate diagrams via browser
3. Deploy Pounamu Environment and Pounamu/Thin Web components

Figure 2-8 Using Pounamu Meta-CASE tool with thin-client diagramming

Initially a tool designer specifies a diagramming tool using the thick-client tools design facilities of our original Pounamu meta-CASE tool (1). This tool design process includes several steps. First, the designer has to define Meta-Model elements using the Meta-Model definer. Second, visual representation of these meta-model elements has to be created with shape and connector creators.

Then in the view-type definer, meta-model elements and their corresponding icons are matched against each other, and in the handler creator, event handlers are defined to perform certain tasks in response to some model and visual events. Finally, these tool specifications are saved to an XML-based tool repository (2) for use by the Pounamu tool interpreter (i.e., Pounamu modeller), which originally provided only thick-client diagramming facilities.

Upon a user's request to edit a diagram via the thin-client user interface, we deploy the Pounamu/Thin web component into the original Pounamu environment (3). In the meantime, tool specifications are loaded from XML-based storage and interpreted by Pounamu to provide the specified diagramming tool (3). Our Pounamu/Thin web components interact with Pounamu via a web services-based remote API.

Pounamu/Thin web components allow users to create, view and edit a graphical diagram in a similar way as has been done in original thick-client diagramming environment. There are at least two ways (GIF and SVG) that a Pounamu diagram can be rendered and displayed on a web browser. With a GIF-based rendering technique, our thin-client diagramming facility can be used via any conventional web browser as long as a web connection exists (4). But, if the alternative rendering technique is preferred, a SVG plug-in has to be installed on a host browser.

The last thing that we want to mention is the tool design facility of the Pounamu system. As Pounamu is a meta-CASE tool, the tool specifications such as diagram element appearance, meta-model entities and attributes, view definitions and event handlers can be changed while the tool is in use (5). Currently, this can only be done via the original thick-client tool designer.

2.5 Summary

Most traditional thick-client diagramming tools (e.g. Rational Rose™, MetaEdit+) have common problems such as: complex user interface, installation and setup overhead, a heavy infrastructure required for group collaboration and a multitude of functionality of which some are rarely used. From this point of view, allowing some users to choose to access these diagramming tools via a web-based, thin-client infrastructure seems reasonable and meaningful. However, our investigation shows most of these thick-client diagramming tools haven't developed this capability. Those tools that do support thin-client diagramming facility typically resort to the augmentation with Java Applet and browser "plug-ins", and therefore restrict the use of tools only to web browsers supporting such technologies. Although some recent attempts to get the rid of such a restriction are successful, diagramming tools built in these attempts are generally perceived too specific to provide appropriate tailorability or integration support. In this thesis, we propose a thin-client diagramming extension to our original Pounamu MetaCASE tool. Our aim is to allow users of Pounamu an alternative way of performing modeling task via web browsers, so that Pounamu can leverage the main advantages provided by web-based applications.

Chapter 3: System Requirements

This chapter describes the requirements for our Pounamu/Thin system. First, a discussion of key requirements of this thin-client diagramming tool is presented. We then focus on capturing system functional requirements and specifications. Next functional constraints captured previously are documented and described in Use Cases. Following that we present the basic set of objects the system needs at a conceptual level. Finally, system non-functional constraints are discussed.

3.1 Key Requirements

The overall task of this thesis project is to develop a Pounamu-based thin-client diagramming system, so that all the essential functionality that the original Pounamu tool has can be accessed via a web-based thin-client user interface. Such an expansion would bring the following advantages to our existing Pounamu tool:

Zero client administration cost: With web-based thin-client user interface, Pounamu can run on any computer as long as it has a web browser installed. This means any installation and maintenance cost of special software on client machines is avoided, and we will have a platform-independent and ease-of-maintenance software tool.

Group collaboration: Since our tool will use the web as a communication medium, it will gain all potential benefits that the web has brought. These include widely supporting group collaboration and allowing portable access from a variety of platforms, etc. With group collaboration any team member is able to contribute to the software project modeling process. With portability the Pounamu tool might not only be accessed via a web browser on a PC but also via cellular phones or personal digital assistants [14].

Before analyzing the detailed requirements that the proposed thin-client diagramming system should meet, the following main features have been captured from the system level:

Plug-in componentware: This thin-client user interface should be added as a plug-in software component to Pounamu. Such component-based solutions would offer great potential for reusing components that support tool integration, collaborative work, end-user interaction and configuration of applications [9].

Send HTTP get and Post commands to browse and edit a Pounamu diagram: By sending HTTP GET commands users can ask for a particular PounamuView diagram of a model project to be displayed on a web browser. Once users enter the editing state by clicking a button, they are able to edit this diagram by interacting with it via a web browser. The user's editing events will be sent as HTTP POST commands, and actual update will be done in the Pounamu server.

Use RMI, CORBA or SOAP as the communication protocol: RMI and CORBA are two commonly used protocols for distributed applications [29]. With them, programs in different hosts can communicate with each other. Like CORBA, SOAP also allows programs in different languages and on disparate hosts to communicate with each other.

Choose XML as the format of communicate messages: For most distributed applications, XML (eXtensible Markup Language) is the ideal transport mechanism since data or commands in this format can be shared between the disparate server and client. In our case, the web server knows nothing about the command objects types that can be executed in Pounamu server; however, it can always compose user requests into XML messages and send them across network. Subsequently, Pounamu, as an XML-based tool, can parse this received XML into its recognizable command object. Besides the above benefits, sending message in XML format also means transmitting Java Strings instead of objects through the connection, which is much faster and easier. Furthermore, the use of XML makes a step on the way towards using SOAP (Simple Object Access Protocol [30]) for remote procedure calls. Since SOAP is a standard message protocol to communicate with web services, making our system SOAP compatible means that we can easily expand it to make use of web services.

3.2 Functional requirements Specification

Functional requirements describe what the system should do, i.e. the services and operations provided for the users and for other systems.

By consulting developers of the existing Pounamu tool, we came up with a list of essential functionality that should be available via the thin-client user interface:

1. ***The Pounamu tool can be accessed via the WWW.***

Currently, the Pounamu tool is a Java-Swing based thick-client Meta-CASE tool, and users have to install a complete version of the tool on their own computer for modeling work. By making this tool assessable via the WWW, the only requirement for a user's machine is that there is a web browser running.

2. ***The system should allow authorized users to browse model projects stored in the file system of the Pounamu server.***

Project modeling is a process of representing concrete or "real-world" problems in a graphical form by using appropriate modeling languages. Results of the modeling process are model projects. In the thin-client multi-user environment, model projects are stored in XML files of the central server, and privileged users are allowed to view these files via web browser.

3. ***The system should allow users to select a Pounamu view diagram in a model project.***

Most times a model project for a real world problem contains more than one Pounamu view diagram (e.g., modeling work on a Video Store system may have class diagrams, sequence diagrams, collaboration diagrams etc.), and users should be able to select their interested Pounamu view diagrams.

4. The requested Pounamu view diagram should be mirrored and displayed on a web browser.

Upon a user's request, the graphical representation of the specified Pounamu view should appear on the user's web browser to be scrutinized.

5. The selected view diagram is editable via a web browser.

Users should be able to edit the Pounamu view diagram by interacting with its mirrored copy on the web browser, i.e., point and click on the displayed image.

6. The system should permit users to add/remove an entity or association type object.

Entity and association are the two fundamental building blocks of a Pounamu view diagram. To adequately edit a view diagram, the first thing is to make sure that we can add/remove these building blocks freely.

7. The system should permit users to move/resize an entity or association object contained in a Pounamu view diagram.

Besides add/remove operation, we also want users to be able to move/resize building elements of a Pounamu view diagram. Thus interaction procedures have to be defined for moving/resizing functionalities.

8. The system should permit users to update properties of an existing entity and association type object.

Property setting should be included to make editing actions complete.

9. The system should provide users with rich interactions so that all above editing actions can be done gracefully.

Although it is possible to fulfill the above editing tasks by submitting HTML form parameters, we would rather users be able to interact with the mirrored Pounamu view diagram. However, due to HTML's limitation in graphical interaction, special design should be taken so that users' experience with our system can be as pleasurable as possible.

10. The system should provide a way to handle the above editing actions and respond to them by displaying an updated Pouanmu view diagram.

3.3 Use Cases

Use cases describe the main interactions of actors with a system. Actors represent the roles that can be played by users of the system. The interaction between actors and the system include input from actors and the system's behavior that convert input to output. Use case diagrams leave details of use cases to use case descriptions that typically include the name of use cases, actors and a sequence of steps that describe event flows of the use case. In this section, all the system functional behaviors captured in the previous section are documented in the use case diagram (shown below), and following it are descriptions of each use case scenario.

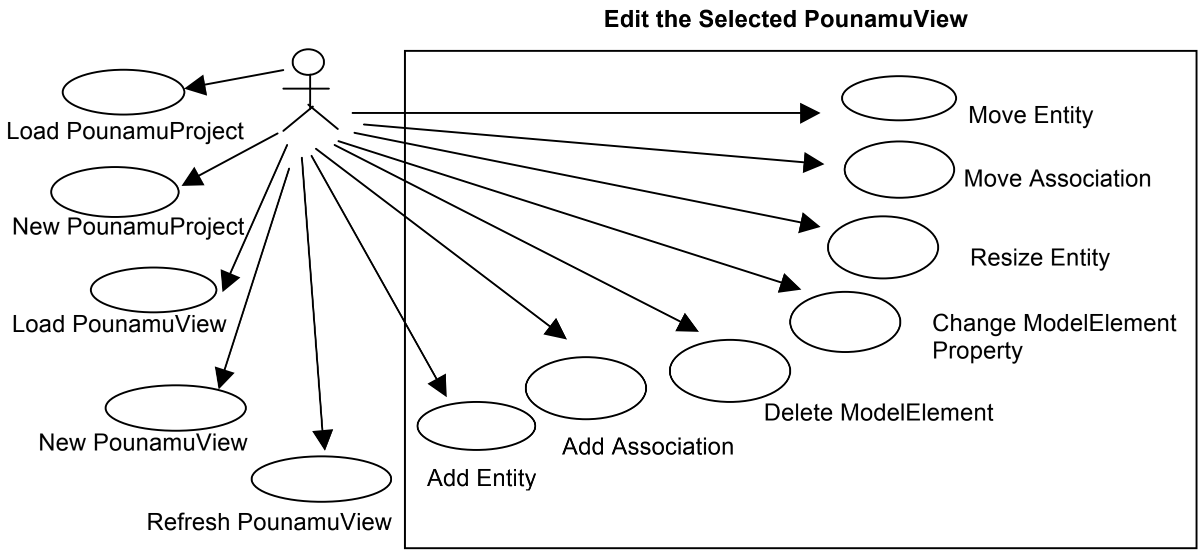


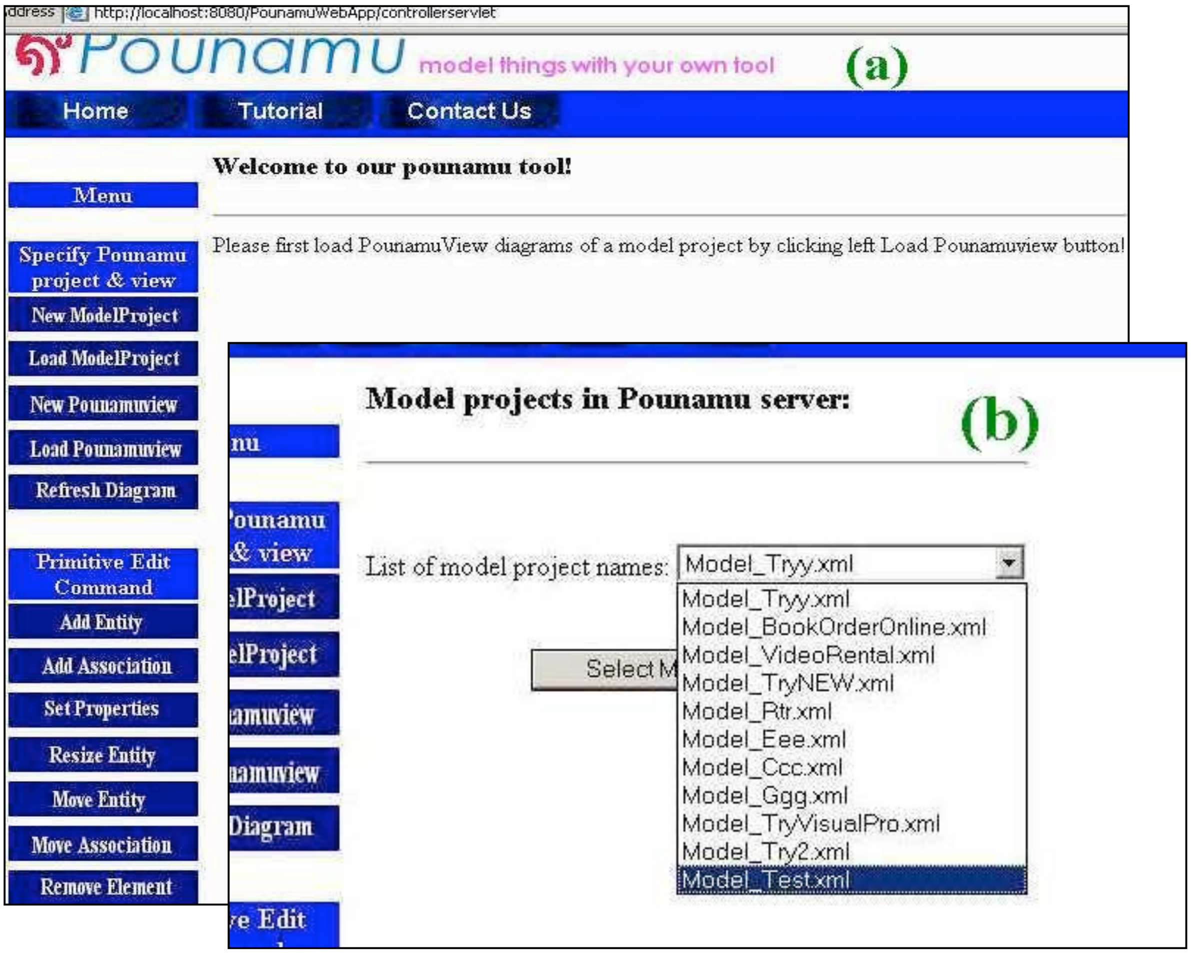
Figure 3-1 System Use Cases Diagram

Use Case "Load PounamuProject"

As mentioned in the previous section, a Pounamu view diagram always belongs to a certain model project. So before viewing and modifying a diagram, we must specify/load the corresponding project. Figure 3-2 shows examples of loading a Pounamu model project. First, the user clicks the "load ModelProject" button on the menu bar (a). Then the user specifies which project is loaded in (b). Last, the system indicates whether the specified project is loaded successfully. Table 3-1 provides a use case description.

Table 3-1 "Load PounamuProject" Use Case Description

Use Case Name:	*Load PounamuProject*
Description:	Used by software tool users to load a working Pounamu model project via a web browser.
Event Flows:	1. Repeat whenever users click the "Load ModelProject" button on the left menu bar. 2. The system provides names of all the model projects stored in the Pounamu server. 3. The user can choose any Pounamu model project as his working project (here working means a user can continue to load PounamuView diagrams contained in this project). 4. The system displays a message showing whether this project loading is successful or not. Upon successful loading, the user's selection is recorded into his session object.



Welcome to our pounamu tool!

Please first load PounamuView diagrams of a model project by clicking left Load Pounamuview button!

Model projects in Pounamu server:

List of model project names: Model_Tryy.xml

Figure 3-2 Examples of Loading a Model Project

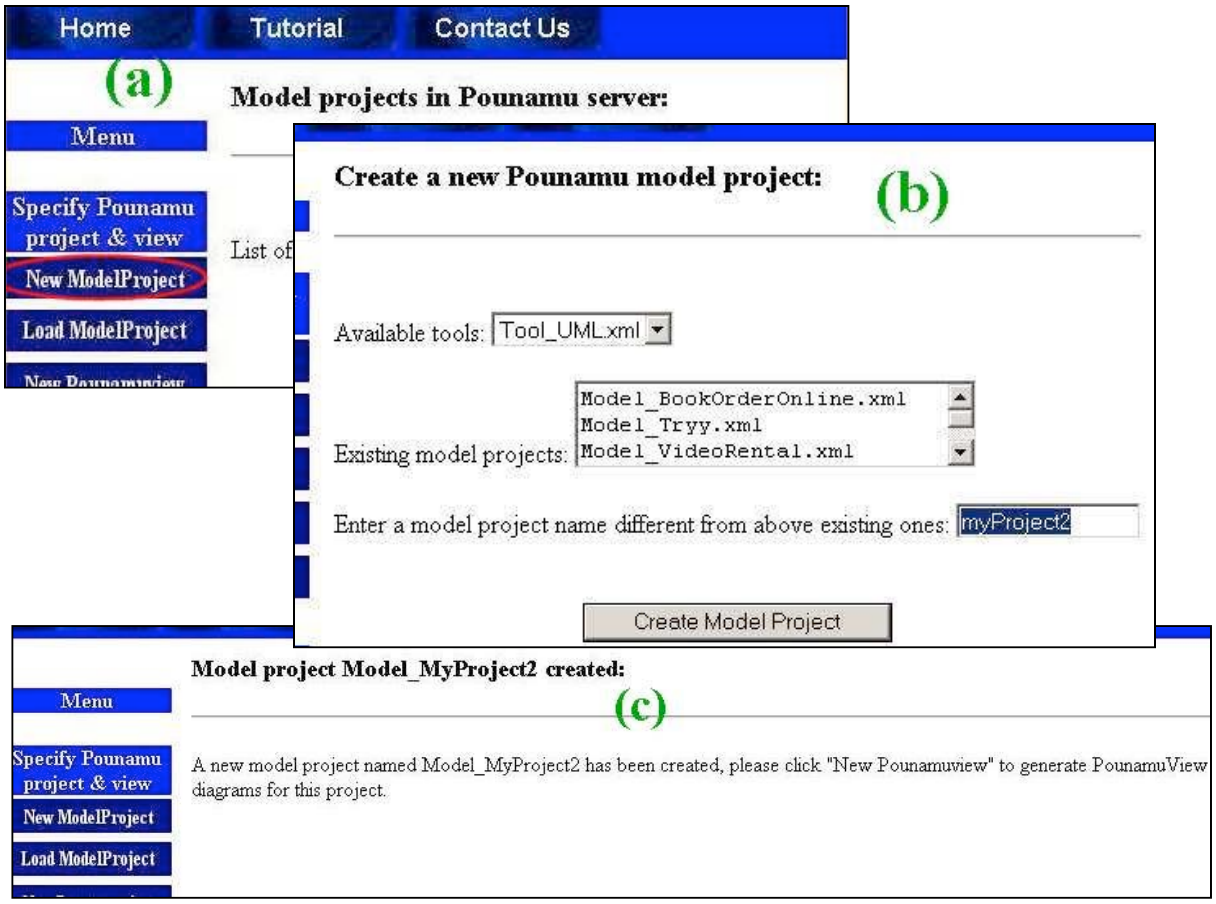
Use Case "New PounamuProject"

In addition to allowing users to load an existing Pounamu model project, the system should also support functionality for creating a new project. An example showing how to perform this creation task is illustrated in Figure 3-3. First, a user needs to click the "New PounamuProject" button displayed on the menu bar (a). Then he has to name this to-be-created model project in (b). Finally, the system gives messages about the result of this new operation. A more detailed use case description can be found in Table 3-2.

Table 3-2 "New PounamuProject" Use Case Description

Use Case Name:	*New PounamuProject*
Description:	Used by tool users to create a new Pounamu model project via a web browser.
Event Flows:	1. Repeat whenever a user clicks the "New ModelProject" button on the left menu bar. 2. The system allows the user to select the underlying tool and enter a name for this new project. 3. If the project with the given name has not already existed, the system will create a new project entry in the Pounamu server. Otherwise, the system would give messages "project creation failed, please give a new name".

	4. The system set this newly created project as the user's working project, this means the user can go on to create a Pounamu view diagram for it.


Home
Tutorial
Contact Us
(a)
Model projects in Pounamu server:
Menu
Specify Pounamu project & view
New ModelProject
Load ModelProject
List of
Create a new Pounamu model project:
(b)
Available tools: Tool_UML.xml
Existing model projects:
Model_BookOrderOnline.xml
Model_Tryy.xml
Model_VideoRental.xml
Enter a model project name different from above existing ones: myProject2
Create Model Project
Model project Model_MyProject2 created:
(c)

A new model project named Model_MyProject2 has been created, please click "New Pounamuview" to generate PounamuView diagrams for this project.

Figure 3-3 Examples of Creating a New Model Project

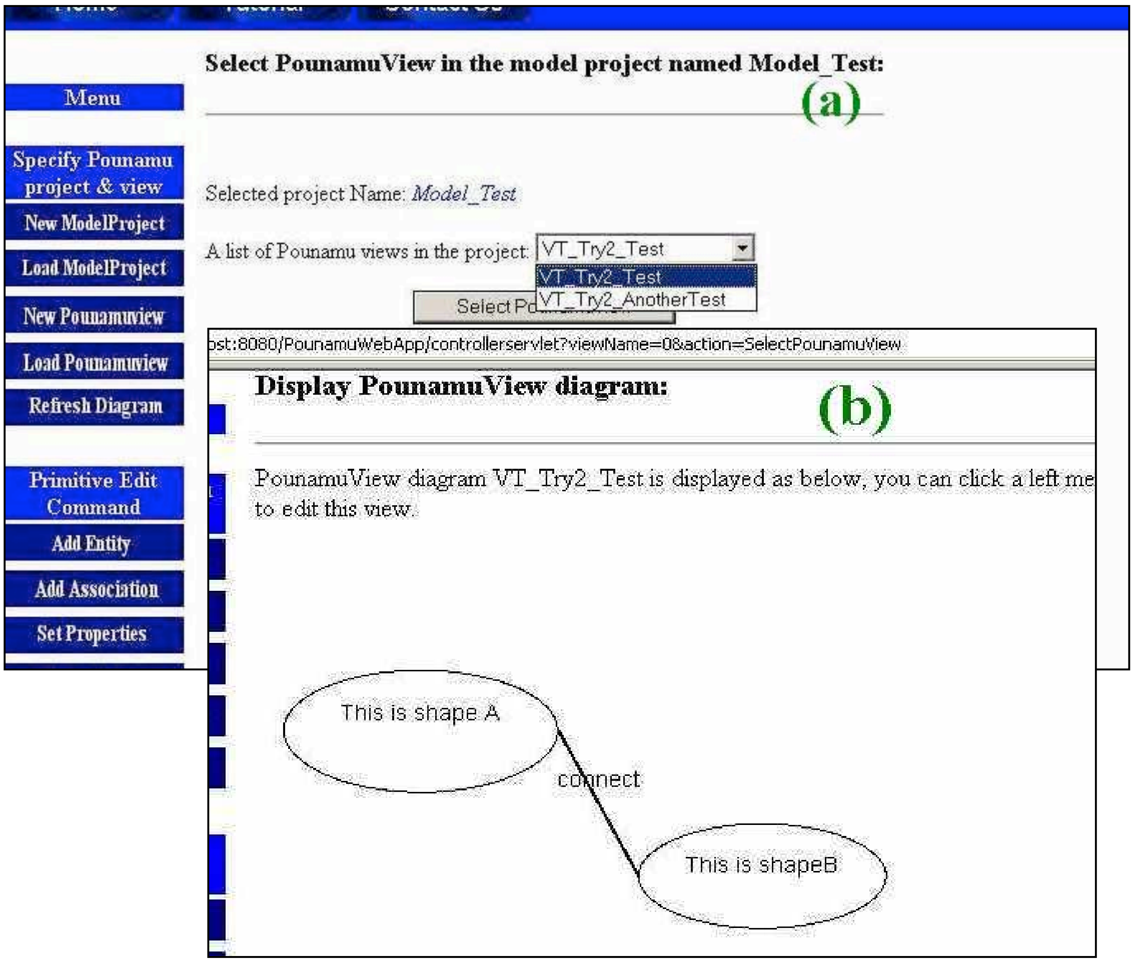
Use Case "Load PounamuView"

After a model project is specified and before an editing action can be initiated, the user must specify a Pounamu view diagram contained in the project. As an example, Figure 3-4(a) illustrates a user deciding to load a test diagram VT_Try2_Test, and the result of the user's action is shown in (b). A more detailed use case description can be found in Table 3-3.

Table 3-3 "Load PounamuView" Use Case Description

Use Case Name	*Load PounamuView*
Description:	Used by tool users to load a working PounamuView via a web browser.
Event Flows:	1. Repeat whenever users click the "Load PounamuView" button on the left menu bar. 2. The system provides names of all the PounamuViews contained in the current working model project.

	3. A user can select a PounamuView as his working view (here working means the user can browse or edit this view). 4. The system records the user's selection into session state. A mirrored image of the specified PounamuView diagram is displayed in the web browser.


Select PounamuView in the model project named Model_Test:

Selected project Name: *Model_Test*

A list of Pounamu views in the project: VT_Try2_Test

st:8080/PounamuWebApp/controllerservlet?viewName=0&action=SelectPounamuView

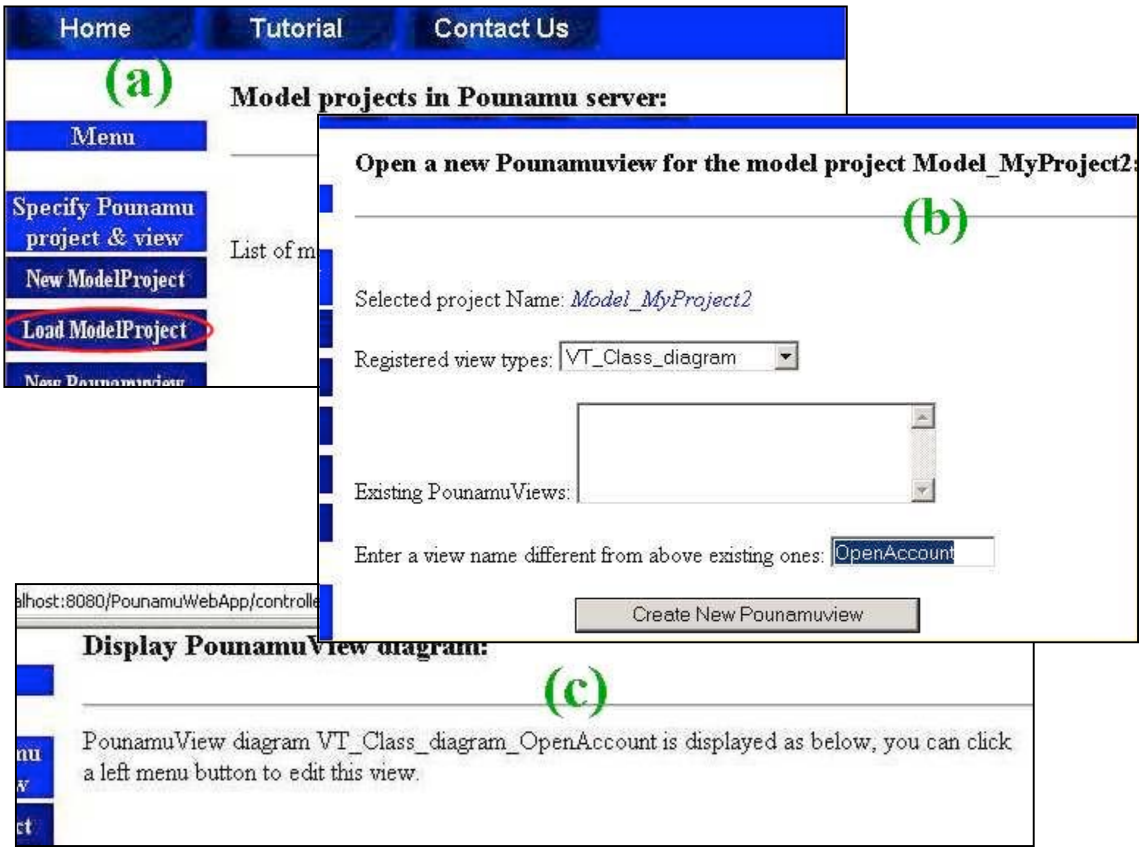
Display PounamuView diagram:

PounamuView diagram VT_Try2_Test is displayed as below, you can click a left me
to edit this view.

Figure 3-4 Examples of Loading a PounamuView Diagram

Use Case "New PounamuView"

For the same reason that users should be allowed to create a new project, our Pounamu/Thin needs to provide functionally for creating a new view diagram for some project, too. Figure 3-5 shows an example of creating a PounamuView diagram. First, the user has to choose a project to which the new view belongs by clicking the "Load ModelProject" button (a). Then the target view relevant information (e.g. the name and view type) needs to be entered in (b). Last, the system will tell the user whether the target has been successfully created or not (c). Table 3-4 lists a more detailed use case description.



Model projects in Pounamu server:

List of m

Open a new Pounamuview for the model project Model_MyProject2:

Selected project Name: *Model_MyProject2*

Registered view types: VT_Class_diagram

Existing PounamuViews:

Enter a view name different from above existing ones: OpenAccount

lhost:8080/PounamuWebApp/controlle

Display PounamuView diagram:

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

Figure 3-5 Examples of Creating a New PounamuView Diagram

Table 3-4 "New PounamuView" Use Case Description

Use Case Name	*Create a PounamuView Diagram*
Description:	Used by tool users to create a new PounamuView diagram via a web browser.
Event Flows:	1. Repeat whenever users click the "New PounamuView" button on the left menu bar. 2. The system allows the user to enter a name and view type for this target view object. 3. The system validates whether the view with given information has already existed in the current working project. If not, a new entry will be set up in the Pounamu server. Otherwise, the system will give messages "the view creation failed because the corresponding item has already existed". 4. The system set this newly created view (still blank) as the user's working view. This means the user can go on to add components to it.

Use Case "Refresh PounamuView Diagram"

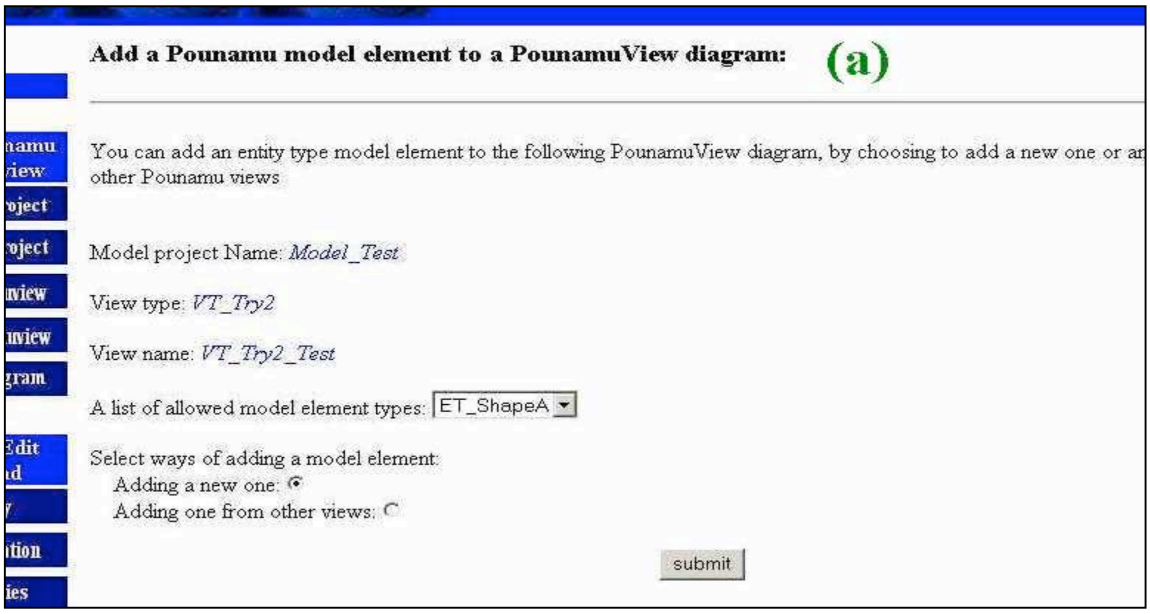
One advantage of the thin-client based diagramming tool is to allow multiple users to collaboratively work on one diagram simultaneously, and refresh functionality is quite useful in such collaborative editing scenarios. For example, Michael has loaded a PounamuView diagram into his browser and is doing some checking. At the same time, user Jessica is editing the same diagram via her browser at a different location. Michael may or may not have recognized that Jessica is working concurrently on the same view, but he can always ask for the latest version of the diagram to be displayed by clicking the "Refresh PounamuView" button. You can refer to Table 3-5 for the detailed use case description.

Table 3-5 "Refresh PounamuView Diagram" Use Case Description

Use Case Name:	*Refresh PounamuView Diagram*
Description:	Used by tool users to refresh an updated PounamuView diagram via a web browser.
Event Flows:	1. Repeat whenever the user click "Refresh PounamuView" button on the main menu bar 1.1 The system inquiry whether the session state information such as the current working PounamuProject and PounamuView is null. If yes, go to 2. 1.2 Upon non-null information, the mirror image of the working PounamuView diagram is regenerated and displayed in the web browser. 2. No PounamuView loaded error, go to "Load PounamuView" use case.

Use Case "Add Entity Element to PounamuView"

Since a Pounamu view diagram is composed of two fundamental building elements (entity shape and association connector), being able to add these visual elements via a web browser is the most basic editing operation. Figure 3-6 shows steps involved to add an entity element to a diagram. In (a), the user enters the type and name for this new element, in (b) the user needs to click a canvas position to place this new entity shape. The result of the addition action is shown in (c). Table 3-6 gives the use case description in detail.



Add a Pounamu model element to a PounamuView diagram:

You can add an entity type model element to the following PounamuView diagram, by choosing to add a new one or an other Pounamu views

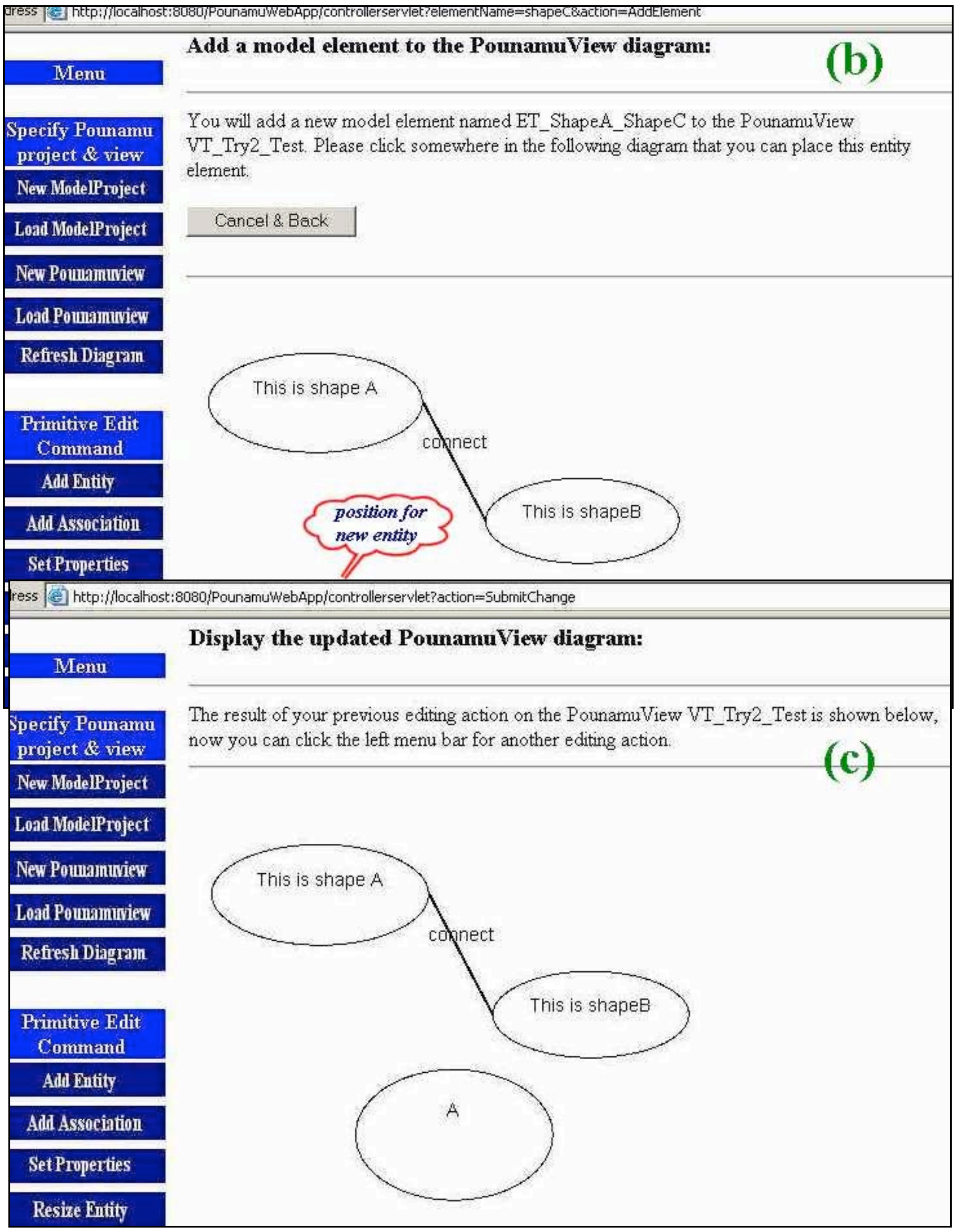
Model project Name: *Model_Test*

View type: *VT_Try2*

View name: *VT_Try2_Test*

A list of allowed model element types: ET_ShapeA

Select ways of adding a model element:
Adding a new one: ●
Adding one from other views: ○



http://localhost:8080/PounamuWebApp/controllerservlet?elementName=shapeC&action=AddElement

Add a model element to the PounamuView diagram:

You will add a new model element named ET_ShapeA_ShapeC to the PounamuView VT_Try2_Test. Please click somewhere in the following diagram that you can place this entity element.

http://localhost:8080/PounamuWebApp/controllerservlet?action=SubmitChange

Display the updated PounamuView diagram:

The result of your previous editing action on the PounamuView VT_Try2_Test is shown below, now you can click the left menu bar for another editing action.

Figure 3-6 Examples of Adding Entity Shape to a PounamuView Diagram

Table 3-6 "Add Entity Shape" Use Case Description

Use Case Name:	*Add Entity Element*
Description:	Used by tool users to add an entity type Pounamu model element to the working PounamuView via a web browser.
Event Flows:	1. Repeat whenever user clicks the "Add Entity" button on the main menu bar. 1.1 The system inquires whether the session state information such as the working project and PounamuView is null. If yes, go to 2.

	1.2 A page with information such as the name of the working view, available entity types in this view and selectable adding methods is displayed. The user chooses a certain entity type and preferable adding method, and clicks the "Add" button. 1.3 If the user has chosen "adding a new one" method, go to 1.5, else if the user has selected "adding one from other views", go to 1.4. 1.4 All the existing model elements with the chosen entity type are listed. The user selects what he wants, and then submits the selection. 1.5 The working PounamuView diagram is displayed. The user clicks somewhere in this diagram to place the new element. 1.6 A page for confirming this "Add Entity" editing action is displayed. The successful execution of "Add Entity" results in the updated PounamuView diagram being redisplayed. Otherwise, go to 3. 2. "No PounamuView loaded" error, please go to "Load PounamuView" use case. 3. Report message "Execution of Add Entity action failed".

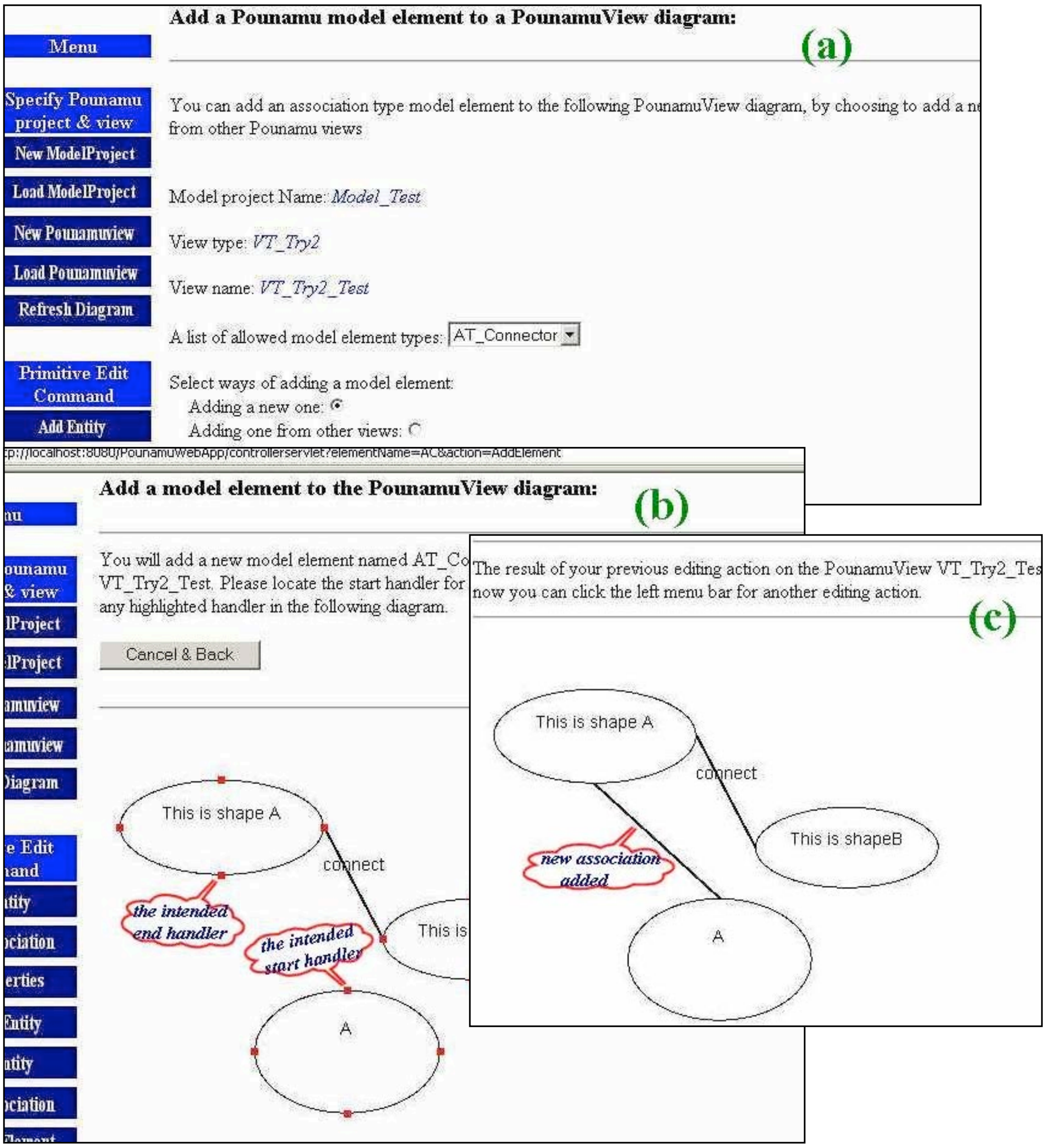
Use Case "Add Association Element to PounamuView"

Similar to the above "Add Entity" Use Case, there are several steps involved in adding association connector to a view diagram. These were demonstrated in Figure 3-7. In (a), some important parameters such as element name and type need be entered. In (b), the user has to specify which two shape handles (highlighted as red small rectangles around the shapes) are to be associated with. The result is shown in Figure(c). Additionally, the use case description can be seen in Table 3-7.

Table 3-7 "Add Association Connector" Use Case Description

Use Case Name:	*Add Association Element*
Description:	Used by tool users to add an association type Pounamu model element to the working PounamuView via a web browser.
Event Flows:	1. Repeat whenever user clicks the "Add Association" button on the main menu bar. 1.1 The system decides whether the session state information such as current working PounamuProject and PounamuView is null. If yes, go to 2. 1.2 A page containing information such as the name of the working view, available association types in this view and adding methods is displayed. The user chooses a certain association type and his preferable adding method, and clicks "Add". 1.3 If the user has selected "adding a new one", go to 1.5, else if the user has selected "adding one from other views", go to 1.4. 1.4 All the existing model elements with the chosen association type are listed. The user selects what he wants, and then submits it. 1.5 The working PounamuView diagram is displayed, in which the handles of all the shapes are highlighted by small red rectangles. 1.6 The user tells the system which two shapes need to be connected with by picking a handle from them. Two handles will be arranged as the first or the second handle of the new association element, respectively. If the user miss-clicked a shape handle, go to 3. Else if two handles are from one shape go to 4. 1.7 A page for confirming this "Add Association" action displays. The successful execution of "Add Association" results in the updated PounamuView diagram being redisplayed. Otherwise, go to 5. Upon

	choosing "Cancel Add", the system reports "previous Add Association action not committed". 2. "No PounamuView loaded" error, please go to "Load PounamuView" use case. 3. Error message "You didn't click a valid shape handle", go to 1.6. 4. Error message "You need to click two handles from different shapes", go to 1.6 5. Report message "Execution of Add Association action failed".



Add a Pounamu model element to a PounamuView diagram:

You can add an association type model element to the following PounamuView diagram, by choosing to add a n
from other Pounamu views

Model project Name: *Model_Test*

View type: *VT_Try2*

View name: *VT_Try2_Test*

A list of allowed model element types: AT_Connector

Select ways of adding a model element:
Adding a new one: ◉
Adding one from other views: ○

tp://localhost:8080/PounamuWebApp/controllerservlet?elementName=AC&action=AddElement

Add a model element to the PounamuView diagram:

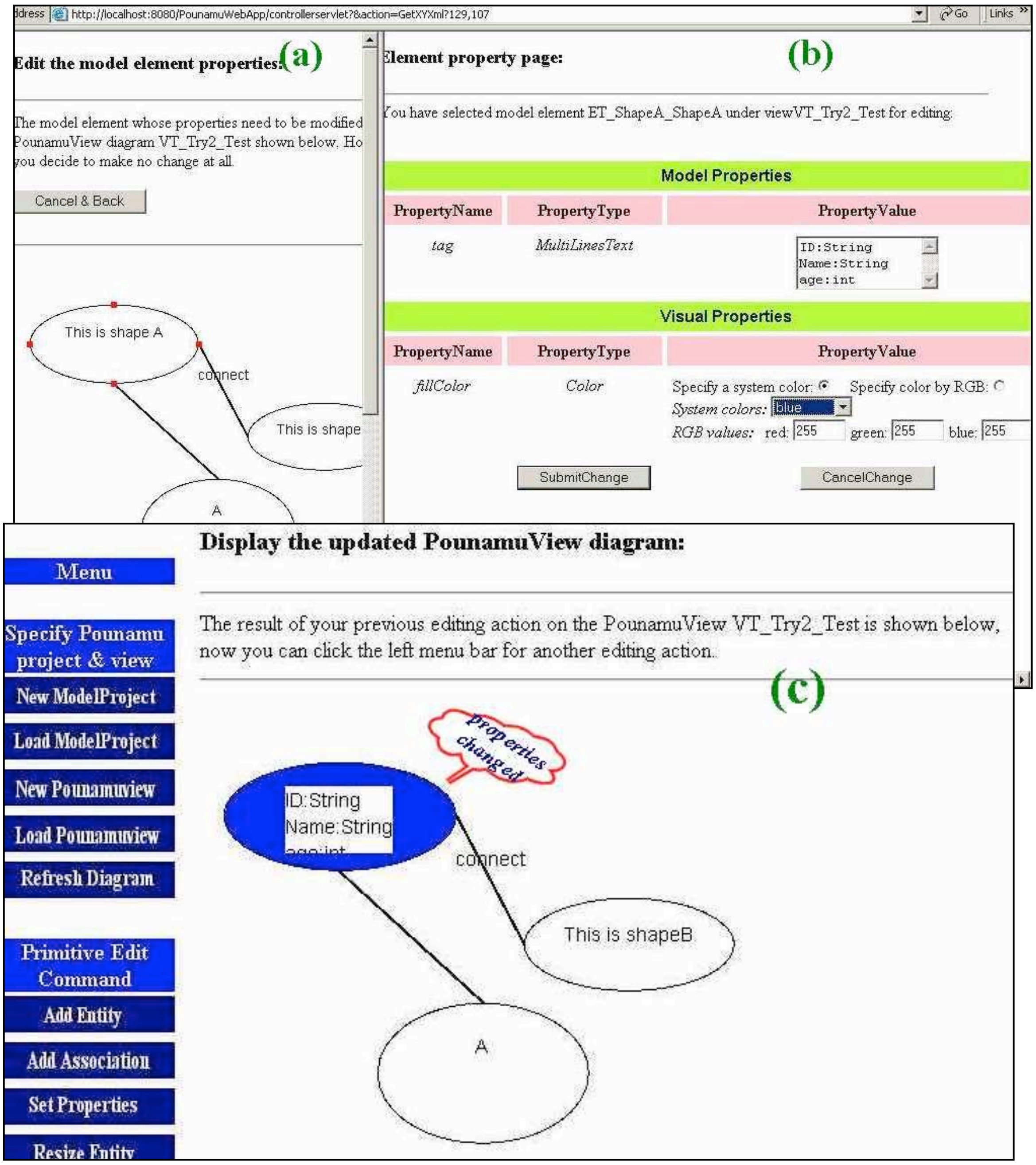
You will add a new model element named AT_Co
VT_Try2_Test. Please locate the start handler for
any highlighted handler in the following diagram.

The result of your previous editing action on the PounamuView VT_Try2_Tes
now you can click the left menu bar for another editing action.

Figure 3-7 Examples of Add an Association Connector to a PounamuView Diagram

Use Case "Set Model Element Properties"

Project modeling is a process in which new instances of model elements are created and added. Most times these instances are differentiated from each other by having disparate property values. From this point of view, being able to change a model element's property is another crucial editing action that needs to be supported. Figure 3-8 illustrates property setting examples. Comparing the original diagram (a) with its updated counterpart (c), you can see the highlighted icon in two diagrams has different model data (text strings) and background color. The use case description can be found in Table 3-8.



ddress http://localhost:8080/PounamuWebApp/controllerservlet?&action=GetXYXml?129,107

Edit the model element properties:

The model element whose properties need to be modified PounamuView diagram VT_Try2_Test shown below. Ho you decide to make no change at all.

Element property page:

You have selected model element ET_ShapeA_ShapeA under viewVT_Try2_Test for editing.

Model Properties

PropertyName	PropertyType	PropertyValue
tag	MultiLinesText	ID:String Name:String age:int

Visual Properties

PropertyName	PropertyType	PropertyValue
fillColor	Color	Specify a system color: ● Specify color by RGB: ○ System colors: blue RGB values: red: 255 green: 255 blue: 255

Display the updated PounamuView diagram:

The result of your previous editing action on the PounamuView VT_Try2_Test is shown below, now you can click the left menu bar for another editing action.

ID:String
Name:String
age:int

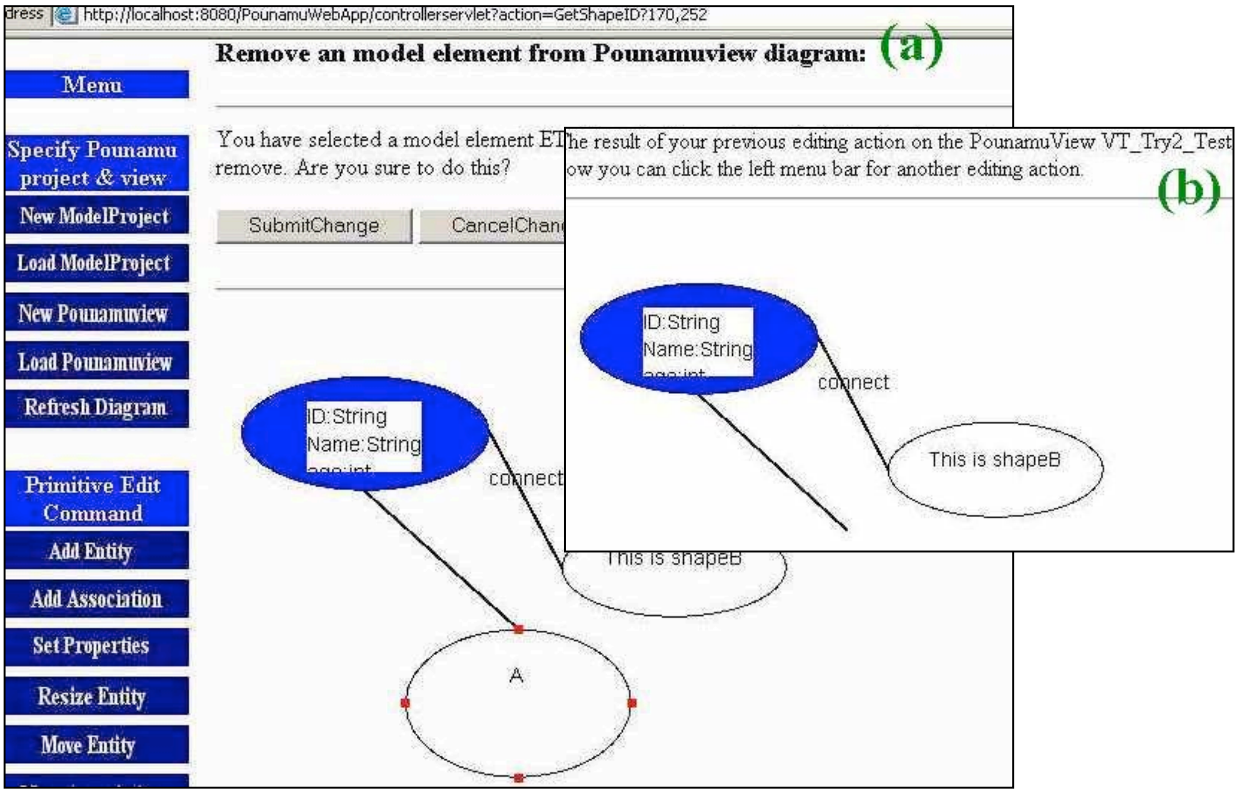
Figure 3-8 Examples of Model Element Property Setting

Table 3-8 "Set Model Element Properties" Use Case Description

Use Case Name:	*Change Model Element Properties*
Description:	Used by tool users to change the properties of an existing model element via a web browser.
Event Flows:	1. Repeat whenever user clicks the "Set Property" button on the main menu bar. 1.1 The system inquires whether the session state information such as the current working project and working view is null. If yes, go to 2. 1.2 The working PounamuView diagram is displayed on the web browser. The user clicks a model element whose properties need to be changed. If the user miss-clicked a model element, go to 3. 1.3 The selected element is highlighted in the displayed diagram. Information such as element name, type, names and values of element properties etc. are also displayed. 1.4 The user types in new values for all the properties he/she wants to change, and then clicks "submit" button. 1.5 A page for confirming this "Change Property" action is displayed. By selecting "Submit Change" button, property setting is executed. By selecting "Cancel Change" button, property setting action is dropped. 1.6 Upon choosing "Submit Change", the successful execution of "Change Property" results in the updated PounamuView diagram being redisplayed. Otherwise, go to 4. Upon choosing "Cancel Change", go to 5. 2. No PounamuView loaded error, please go to "Load PounamuView" use case. 3. Error message "You didn't click a model element", go to 1.2. 4. Report message "Execution of Change Property action failed". 5. Report message "Previous property setting action is not committed".

Use Case "Remove Model Element"

An example of removing a Pounamu model element is illustrated in Figure 3-9. First, the user clicks the target that needs to be removed from the diagram, and the clicked shape/connector is highlighted with surrounding red handles (a). After confirmation of this removal action, the result diagram looks like (b). Table 3-9 shows a detailed description for this use case.


http://localhost:8080/PounamuWebApp/controllerservlet?action=GetShapeID?170,252

Remove an model element from Pounamuview diagram:

You have selected a model element ET... remove. Are you sure to do this?

...he result of your previous editing action on the PounamuView VT_Try2_Test ...ow you can click the left menu bar for another editing action.

Figure 3-9 Examples of Removing Element from the PounamuView Diagram

Table 3-9 "Remove Model Element" Use Case Description

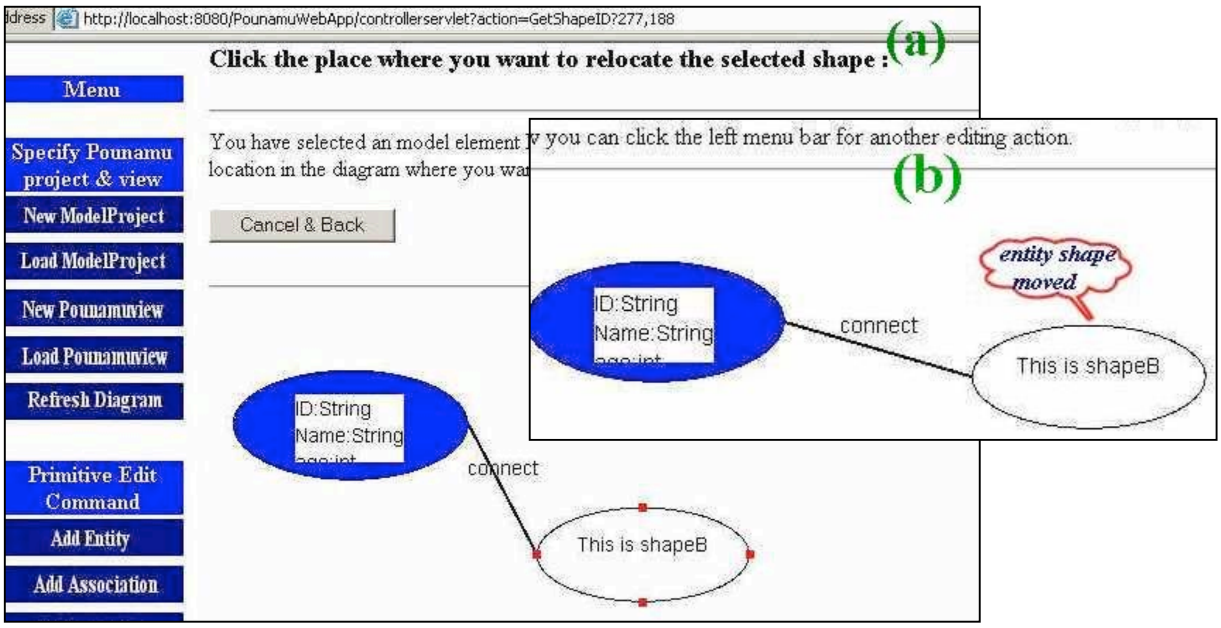
Use Case Name:	*Remove Model Element*
Description:	Used by software tool users via WWW browser to remove an existing model element from the working PounamuView.
Event Flows:	2. Repeat whenever user click "Remove Element" button on the main menu bar. 1.1 The system inquires whether the current session state information such as the working PounamuProject and PounamuView is null. If yes, go to 2. 1.2 The mirrored image of the working PounamuView diagram is displayed. The user can click a model element which needs to be deleted from the diagram. If the user miss-clicked a model element, go to 3. 1.3 A page for confirming this "Remove Element" action displays. By selecting "Submit Change" button, "Remove Element" is executed. By selecting "Cancel Change" button, this editing action is dropped. 1.4 Upon choosing "Submit Change", successful execution of "Remove Element" results in the updated PounamuView diagram being redisplayed. Otherwise, go to 4. Upon choosing "Cancel Change", go to 5. 3. "No PounamuView loaded" error, please go to "Load PounamuView" use case. 4. Error message "You didn't click a model element", go to 1.2. 5. Report message "Execution of Remove Element action failed". 6. Report message "Previous Remove Element action is not committed".

Use Case "Move Entity Shape"

In order to have an optimized layout, positions of modeling elements (entity shape and association connector) in a PounamuView diagram may need to be adjusted. Figure 3-10 illustrates an example of moving an entity shape. In (a), you can see the target has been highlighted with surrounding red handles. Messages shown at the top of the page remind the user to click an ideal position for this target. The result of this action looks like (b). Table 3-10 gives a detailed description for this use case.

Table 3-10 "Move Entity Shape" Use Case Description

Use Case Name:	*Move Entity Shape*
Description:	Used by tool users to move an existing model element in the working PounamuView diagram to a new position via a web browser.
Event Flows:	1. Repeat whenever user clicks the "Move Entity" button on the main menu bar. 1.1. The system decides whether the current session state information such as working PounamuProject and PounamuView is null. If yes, go to 2. 1.2. The working PounamuView diagram is displayed. The user can click a model element that needs to be moved in the diagram. If the user miss-clicked a model element, go to 3. 1.3. The user specifies a new position for the model element by clicking somewhere in the PounamuView diagram. 1.4. A page for confirming this "Move Entity" action is displayed. The successful execution of "Move Entity" results in the updated PounamuView diagram being redisplayed. Otherwise, go to 4. Upon choosing "Cancel Change", the system reports "previous move entity action is not committed". 2. No PounamuView loaded error, please go to "Load PounamuView" use case. 3. Report error message "You didn't click a model element", go to 1.2. 4. Report message "Execution of Move Entity action failed".



Click the place where you want to relocate the selected shape :

You have selected an model element ... location in the diagram where you wa...

... you can click the left menu bar for another editing action.

Figure 3-10 Examples of Moving an Entity Shape in a PounamuView Diagram

Use Case "Move Association Connector"

For the same reasons for moving an entity shape, positions of association connectors also need to be adjusted to make the whole diagram look better. Generally speaking, an association connector is moved by relocating its start and end shape handles. Figure 3-11 illustrates an example of how to do this. A detailed use case description is shown in Table 3-11.

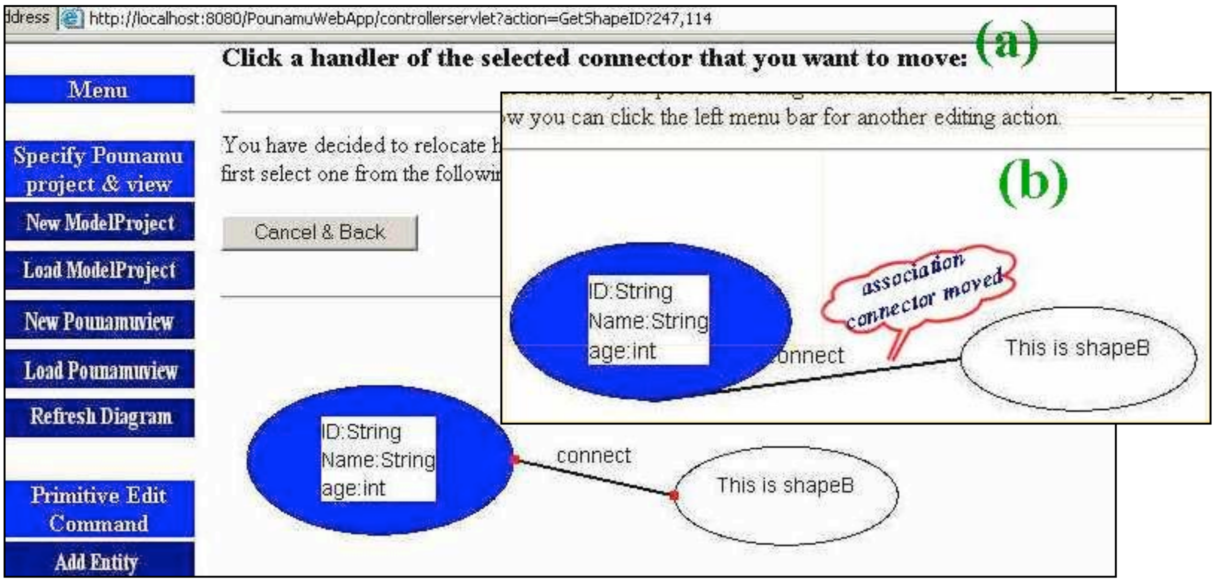

ddress http://localhost:8080/PounamuWebApp/controllerservlet?action=GetShapeID?247,114
(a)
Click a handler of the selected connector that you want to move:
Menu
Specify Pounamu project & view
New ModelProject
Load ModelProject
New Pounamuview
Load Pounamuview
Refresh Diagram
Primitive Edit Command
Add Entity
You have decided to relocate h
first select one from the followi
Cancel & Back
ID:String
Name:String
age:int
connect
This is shapeB
w you can click the left menu bar for another editing action.
(b)
association connector moved
onnect

Figure 3-11 Examples of Moving an Association Connector in a PounamuView Diagram

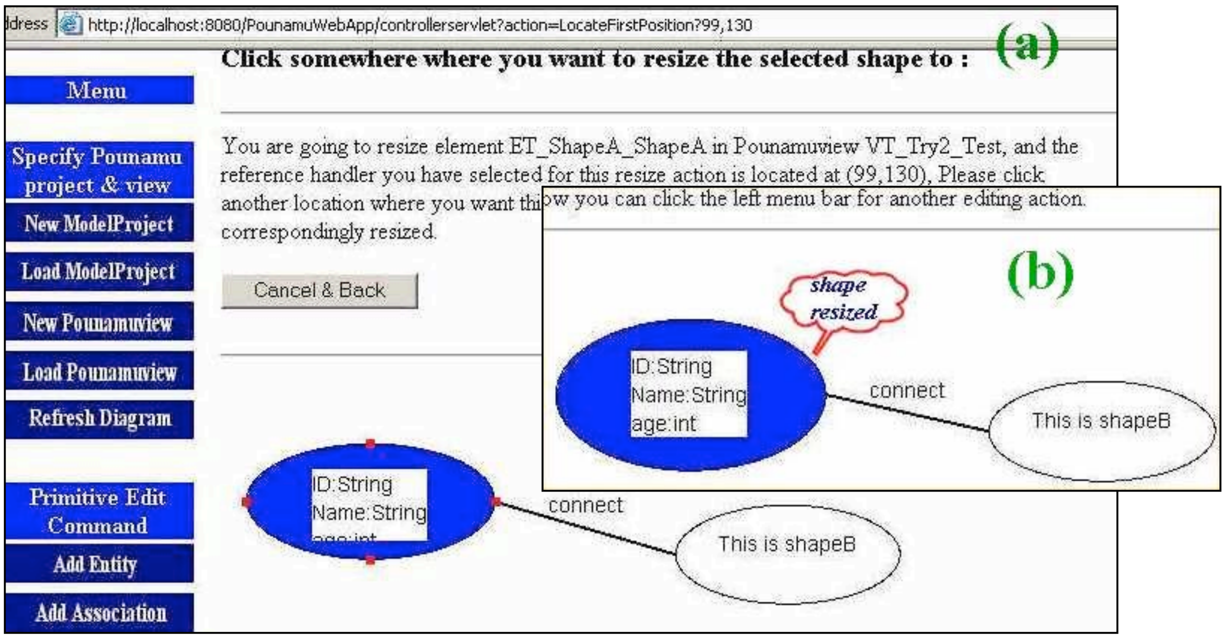
Table 3-11 "Move Association Connector" Use Case Description

Use Case Name:	*Move Association Connector*
Description:	Used by tool users to move an association object in the working diagram via a web browser.
Event Flows:	1. Repeat whenever the user clicks the "Move Association" button on the main menu bar. 1.1 The system inquiries whether the current session state information such as working PounamuProject and PounamuView is null. If yes, go to 2. 1.2 The working PounamuView diagram is displayed. The user can click any association element which needs to be moved in the diagram. If the user miss-clicked an association element, go to 3. 1.3 The selected association is highlighted. The user then needs to choose which one of two handles will be moved. If no handle has been selected, go to 4. 1.4 Upon successful selection of a handle, one of two entity shapes that are connected by this association element is highlighted. The one got highlighted is the parent of the chosen handle. 1.5 The user clicks another handle of the highlighted shape so that the association element is relocated there. If no handle has selected, go to 5. 1.6 A page for confirming this "Move an association element" action is displayed. By selecting "Submit Change" button, system relocates the association to a different shape handle. By selecting "Cancel Change" button, this editing action is dropped and system goes to 6. 1.7 The successful execution of "Move an association element" results in the updated PounamuView diagram being redisplayed. Otherwise, go to 7.

	2. "No PounamuView loaded" error, please go to "Load PounamuView" use case. 3. Error message: "You didn't click a model element", go to 1.2. 4. Error message: "You didn't select a valid association handle to move", go to 1.3. 5. Error message: " A valid shape handle is not selected for relocating the association", go to 1.5. 6. Report message "Previous "Move Association" action is not committed". 7. Report message "Execution of Move Association action failed".

Use Case "Resize Entity Shape"

Similar to the previous moving shape action, shape resizing also aims to make a diagram look better. Figure 3-12 describes examples of resizing an entity shape. The selected shape icon has been highlighted in (a). Messages shown at the tope of the page tell the user to click a reference point for the resizing. After that, another click in a diagram offers the destination for the previous reference point. Finally the system proportionally resizes the whole entity shape based on information provided and displays the resultant diagram (b). A more detailed description for this "Resize Entity" use case is listed in table 3-12.



ddress http://localhost:8080/PounamuWebApp/controllerservlet?action=LocateFirstPosition?99,130

Click somewhere where you want to resize the selected shape to :

You are going to resize element ET_ShapeA_ShapeA in Pounamuview VT_Try2_Test, and the reference handler you have selected for this resize action is located at (99,130), Please click another location where you want thi

correspondingly resized.

w you can click the left menu bar for another editing action.

Figure 3-12 Examples of Resizing an Entity Shape in a PounamuView Diagram

Table 3-12 "Resizing Entity Shape" Use Case Description

Use Case Name:	Resize Entity Shape
Description:	Used by tool users to resize an existing entity shape in the working PounamuView diagram via a web browser.
Event Flows:	1. Repeat whenever user clicks the "Resize Entity Shape" button on the main menu bar. 1.1 The system decides whether the current session state information

	such as the working PounamuProject and PounamuView is null. If yes, go to 2. 1.2 The working PounamuView diagram is displayed. The user can click an entity shape which he wants to resize. If the user miss-clicked a shape element, go to 3. 1.3 The selected entity shape is highlighted. The user picks a handle from this shape as a reference point for the resizing action. If no handle has been selected, go to 4. 1.4 The user specifies a final position for the previous designated reference point by clicking somewhere in PounamuView diagram. Both of the initial and final positions of a reference handle are recorded. 1.5 A page for confirming this " Resizing Entity Shape" action is displayed. By selecting "Submit Change" button, system executes resizing action based on the relative distance between two recorded positions. By selecting "Cancel Change" button, this editing action is dropped and system goes to 5. 1.6 Successful execution of "Resize Entity Shape" results in the updated PounamuView diagram being redisplayed. Otherwise, go to 6. 2. "No PounamuView loaded" error, please go to "Load PounamuView" use case. 3. Error message "You didn't click a model element", go to 1.2. 4. Error message " You didn't select a valid handle as resizing reference point", go to 1.3. 5. Report message "Previous "Resize Entity" action is not committed". 6. Report message "Execution of Resizing Entity action failed".

3.4 Conceptual Objects Modeling in UML Class Diagrams

In this section, we will use UML class diagrams to develop functional requirements specified in the above use cases. The class diagrams describe the types of the objects and the static relationships that exist among them [31], and they will represent the concepts of the system without concerning any other details such as user interface, middleware, and data manager. To develop these diagrams, first we need to identify the basic sets of objects which encapsulate the main data and functions of the system. Second, we have to determine the relationships among these objects. In summary, according to Bruegge [33], the transforming activities from system use cases to OOA class diagrams are described as the following:

- Identifying entity objects
- Identifying boundary objects
- Identifying control objects
- Mapping use cases to objects
- Identifying associations among objects
- Identifying object attributes

Considering the overall objective of this project is to design the thin-client user interface for the exiting Pounamu tool, we think it would be much easier and clearer to describe the task of developing OOA class diagrams with two sub-parts: a remote interface plug-in class diagram and a web front-end class diagram, so here this explanation approach is pursued.

3.4.1 Remote Interface Plug-in Class Diagram

Before discussing our remote interface plug-in class diagram, it would be helpful to have a brief introduction to some essential OOA objects already in the Pounamu Meta-CASE tool. These objects are generalized below:

- ***Pounamu:*** *Pounamu tool working environment*

This object represents the Pounamu visual programming tool itself. It's an entry point for the Pounamu application.

- ***PounamuProject:*** *Representation of Pounamu working instance, it can be a defined tool or modeling project*

This object holds information such as the project name, type, as well as a list of PounamuViews that the project contains. The main methods it includes are Add/Remove projects from Pounamu, as well as find/restore projects etc. It also has set/get attributes (project name, type etc.) functions.

- ***PounamuModelElement:*** *Data repository of modeling*

This object holds information such as the model element name, the name of the view which this element is in, the matching icon, and a list of model properties. In addition to property setting/getting functions, this object also contains methods such as Set/Get the related visual icon, Add/Remove itself from a PounamuView diagram, and Move/Resize its underlying icon etc.

- ***PounamuView:*** *The graphical representation of modeling work*

A PounamuView object holds information such as view name, view type, and a list of PounamuModelElements that make up this diagram. The key functions include New/Remove/Sore/Load this view diagram, as well as set/get view's properties (name, type etc.).

- ***PounamuIcon:*** *The visual representation of PounamuModelElement*

This object holds information such as the icon name, the icon type, the related model element name and a list of visual properties (including shape type, background color, line type, font etc.). The key methods include set/get the related visual properties, as well as save/register the icon etc.

- ***PounamuCommand:*** *Being used to "run" updates on Pounamu model/view state*

This object holds information such as the name and type, and it also contains an instance variable of the PounamuView type. A key method this object has is to execute updates on the Pounamu model/view state. Additionally, it includes functions used to undo and redo updates that have already been made.

To better understand the functional role of these classes, a UML tool specified with the Pounamu Meta-CASE tool has been used as an example in our explanation below. Actually this customized UML tool is a tool-type PounamuProject. When the user defines UML class diagrams or sequence diagrams with this tool, he has created a model-type PounamuProject. These class diagrams or sequence diagrams are so-called PounamuViews. Shapes or connectors which are basic components of these diagrams are PounamuModelElements.

The OOA class diagram depicting inter-relationships among classes explained above is illustrated in Figure 3-13.

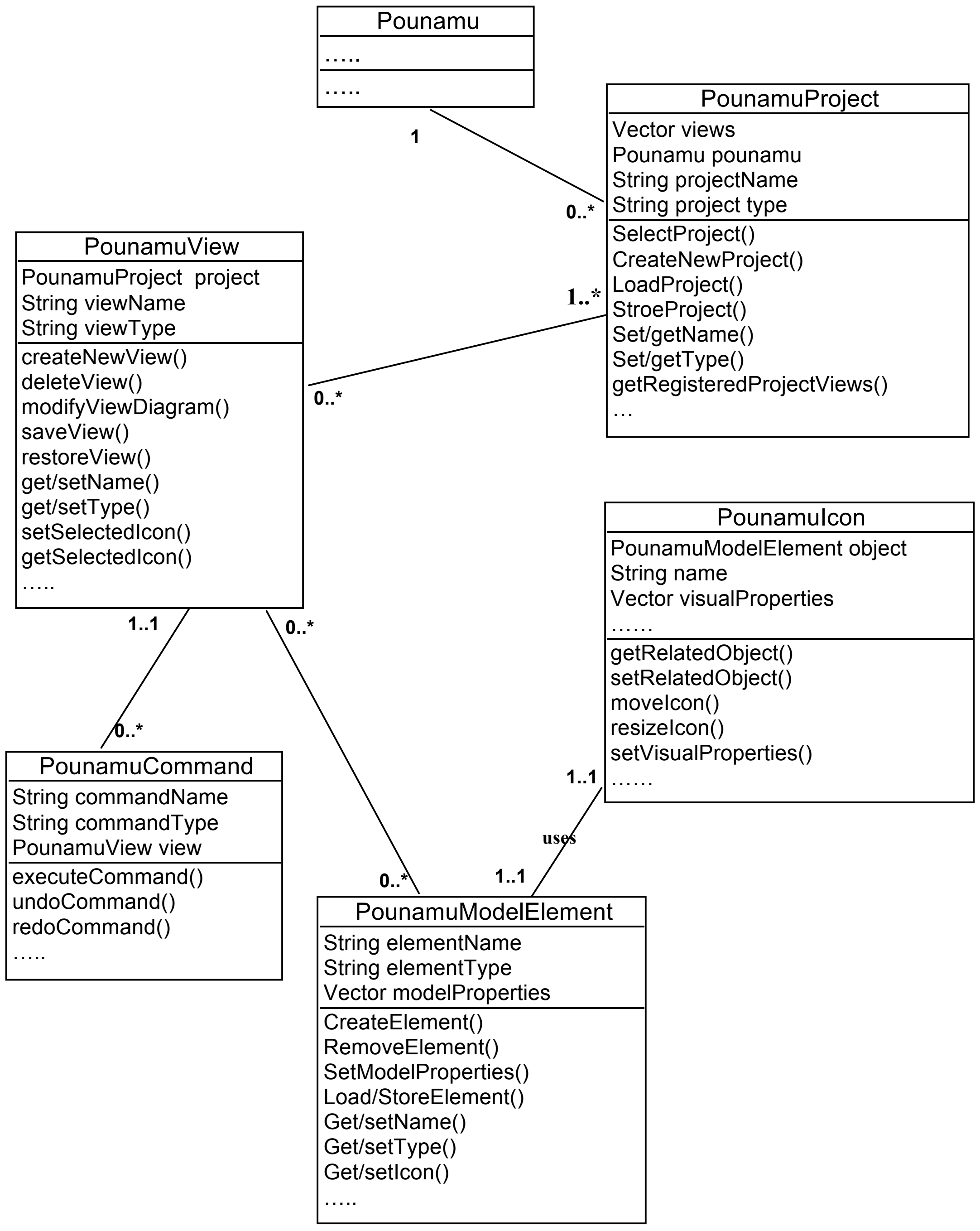


Figure 3-13 OOA objects in the original Pounamu Meta-CASE tool

Having understood basic knowledge of some essential OOA classes in the existing Pounamu, we can embark on the journey of developing OOA objects for a remote editing plug-in component. Since the main task of this component is to receive remote editing messages from a web front-end and then respond to them by executing these editing commands, undoubtedly classes representing remote command objects are needed. Additionally, the role of receiving and analyzing the remote editing messages is taken by another important conceptual model object, *RemotePounamuEditor*. A more detailed description of these objects is given below, and a UML class diagram showing their inter-relationship is illustrated in Figure 3-14.

RemotePounamuEditor:

This object represents the pluggable remote editing environment in which PounamuView diagrams can be manipulated remotely via a web browser. It holds a reference to the original Pounamu object. A list of functions it provides are: opening a model project, getting information of all PouanmuViews in a model project, generating a mirrored image of a PounamuView diagram (in a format that is displayable on a web browser), receiving and analysing remote editing command messages etc.

RemoteCommand:

This object contains all attributes for a remote editing command, including the command name, the command type, the view name (the working PounamuView which this command is actioned on) etc. It provides functions for executing the command on the corresponding PounamuView diagram.

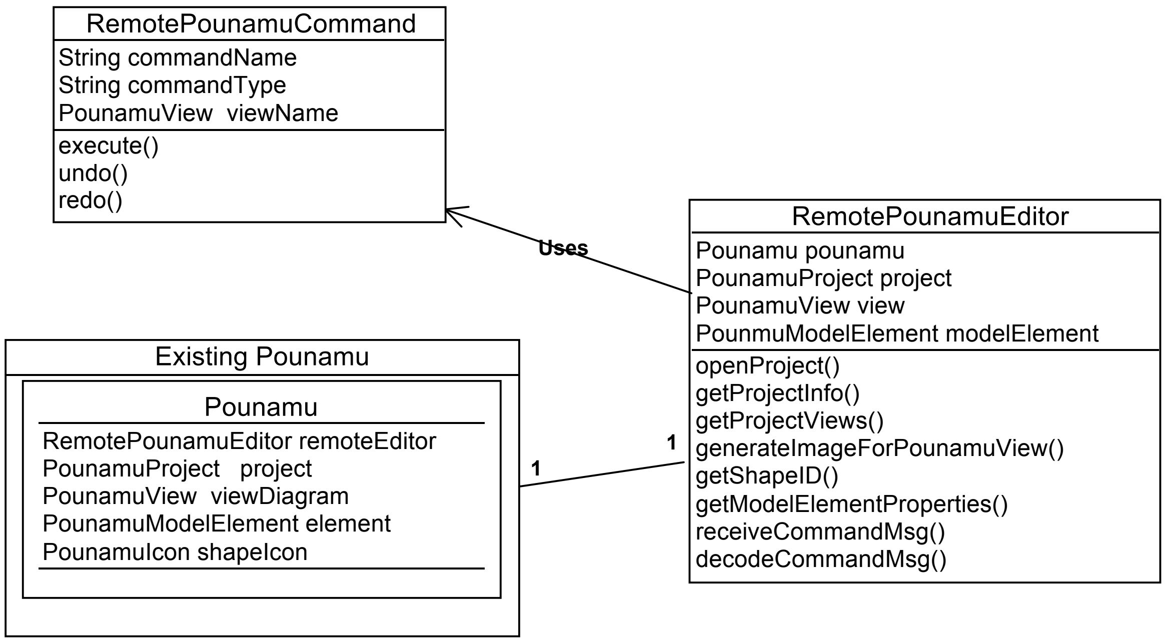


Figure 3-14 Conceptual Model Objects & Inter-relationships for a Remote Interface Plug-in

3.4.2 Web Front-end Class Diagram

In this section, we have identified three conceptual model objects. A detailed description about these classes is presented below, and it will be followed by a UML class diagram (Figure 3-15) depicting inter-relationships among them.

EditingActionData: *An object representing an editing action on the mirrored image of the working PounamuView diagram.*
The attributes this object contains are the action name, the name and type of the target entity/association (PounamuModelElement), property names and values of the target object. The main function it provides is to construct a remote editing command object which can be transmitted across the network. Additionally, it contains functions to lay/remove edit sketches to/from the original diagram.

ApplicationSession: *An object used to maintain the application session state.*
This object holds information such as the name of the working *PounamuView* diagram, the name of the working *PounamuModelProject* and the current *EditngActionData*. Functions it provides include loading the working project and view, as well as setting the user-specific data properties.

PounamuImage:
This object holds information about a *PounamuView* diagram. These include the name of *PounamuView*, the name of *PounamuModelProject* to which this view belongs, and the image data (including content, size, type). It mainly provides the following functions: displaying the working *PounamuView* diagram on a web browser, locating a mouse click on the image, obtaining ID and properties of a target shape/connector under the mouse. Other useful functions include: adding/removing a model element to/from the diagram, moving or resizing an existing model element.

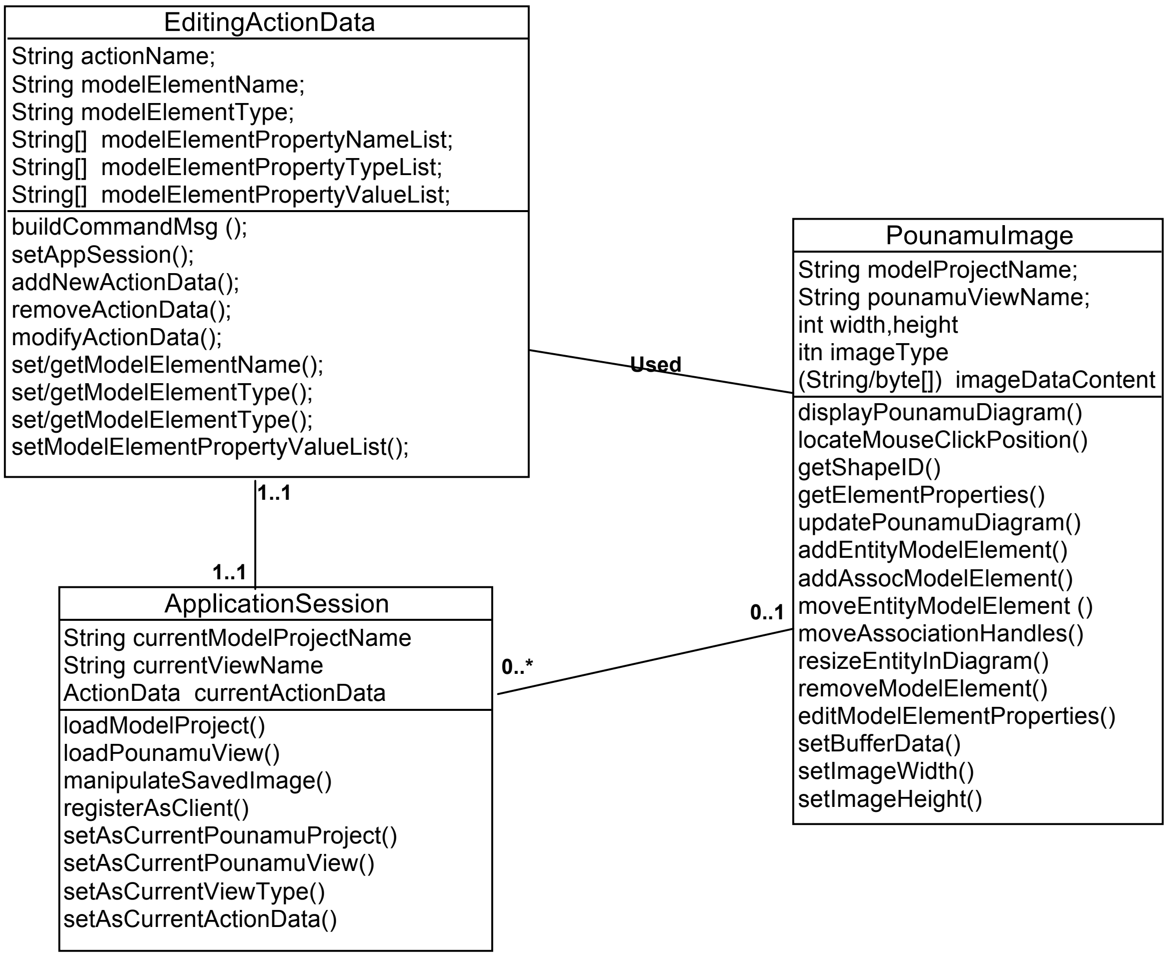


Figure 3-15 Conceptual Model Objects & Inter-relationships for a web front-end

3.5 Non-functional requirement specifications

Non-functional requirements refer to those requirements that are not directly concerned with specific functions performed by the system [32]. They define constraints on the system as a whole rather than individual functional requirements, and examples of these constraints include system reliability, response time, data representations used in the system interface etc. In our system, non-functional requirements are summarized as the following:

♦ **User Interface and Human Factors:** The Pounamu/Thin system has a web-based UI, and this allows multiple users to access the Pounamu tool via a web browser without the need to download or install any special software. Simply speaking, this means a user should be able to create, edit and delete *PounamuView* diagrams displayed in a web browser. In addition to supporting all the core functionalities enabled in the original tool, an extra requirement for this web-based UI is that it should provide as conventional approach to design of web based UIs as possible, i.e., using the POST/GET page display metaphor. Moreover, this interface should be user-friendly and satisfy all functional requirements.

- **Performance & System Response:** Our system should allow dozens of people to collaboratively edit a *PounamuView* diagram simultaneously (this feature is important as the initial stage of software development process often involves group collaboration to come up with a satisfied system design.). The system should have a good response to the user's editing commands, i.e., the updated *PounamuView* diagram need be redisplayed at typical LAN connection speed.
- **Hardware considerations:** The Web server and the Pounamu server host machines should be high-end machine with large memory, fast CPU, large & fast hard disk etc. These server machines are networked by a high-speed LAN. Ordinary users can have a moderate-sized desktop PC, and they can access the system via a wide or local area network.
- **Documentation:** Whenever needed, users can always refer to the detailed on-line help and tutorial about how to use the system. This would be especially useful for novices.
- **Error handling and Extreme conditions:** Any incorrect use of the system is reported with user-friendly error messages. In our system, most of the functionalities have been designed to have a series of associated interactions with the system. A mistake in any single interaction should be handled gracefully, otherwise the whole series of interactions would be wasted. For any invalid interaction, meaningful error messages are recommended.
- **System Integration:** The thin-client user interface should be integrated into the existing Pounamu using a remote object API and component technology. This arrangement ensures our remote user interface can be easily plugged in/out without influencing other system functionality. More importantly, this component-based integration makes system enhancement and extension easier tasks.

3.6 Summary

System requirements provide the descriptions of this system's functional behaviors and non-functional constraints. A high degree of accuracy and completeness in the requirement analysis is essential for any successful system. In this chapter, 12 use cases have been generalized to describe the main interactions of actors with this thin-client Pounamu tool. A set of high-level conceptual model objects is identified, and static relationships among these objects are illustrated in UML OOA class diagrams. In summary, this chapter prepares us for the next system design task.

Chapter 4: System Design

Based on the system requirements and object-oriented analysis developed in the previous chapter, we can move to the system design stage of software life cycle. The tasks involved in this stage can be described as follows:

- First, according to non-functional specification and functional specification of the system, we need to build a suitable software architecture model for the system.
- Secondly, the logical objects and their processes retrieved at the OOA stage will be further split into objects based on software components, which come closer to the system implementation phase in the software development.
- Thirdly, the functionality and dynamic interaction between system components also needs further expansion.

Before stepping into any of the above tasks, we will give an overall picture of the system, its main parts and how subparts can fit together to solve the problem. This is the first attempt we make to put the whole system into perspective.

4.1 The Overall System Diagram

Figure 4-1 shows an overall system diagram that roughly describes how to integrate a thin-client user interface component (Pounamu/Thin) with the existing Pounamu Meta-CASE tool.

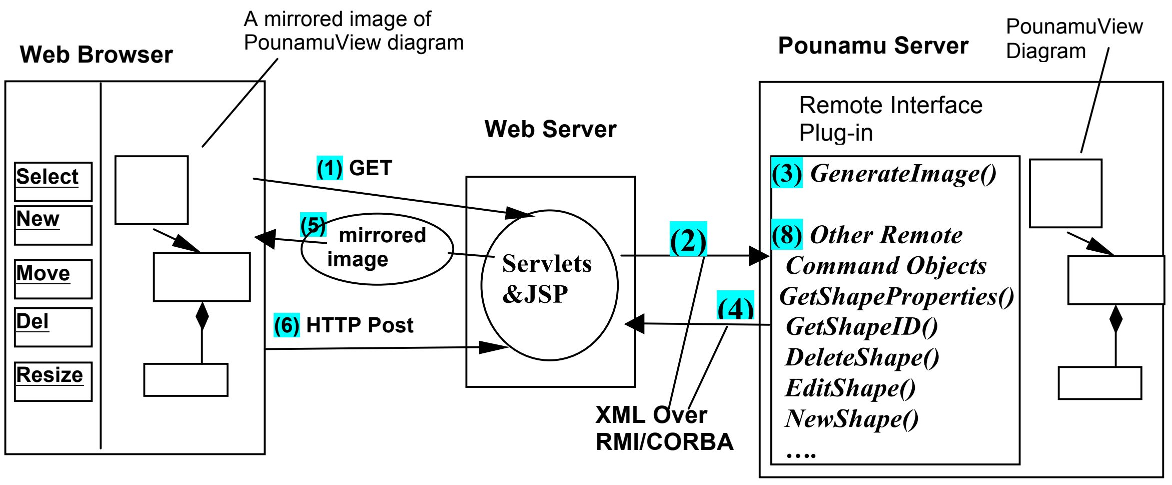


Figure 4-1 System Diagram

The above diagram tells that users can ask for a modeling artifact (stored in the form of a PounamuView diagram) to be displayed in their web browser in a similar manner to request a web page (1).

Servlets/JSP in charge of this HTTP Get command would communicate with the Remote Pounamu Server through the RMI/CORBA protocol (2). Then the mirrored image of the requested PounamuView diagram is dynamically generated by the Pounamu server (3) and sent back to the web browser for display (4)(5). Users can also edit the mirrored image by entering an *editable* state, and their editing events will be sent to the Web Server as HTTP POST commands (6). The Servlet/JSP recomposes this POST command into a remote command object and sends it back to the Pounamu Server (7). Finally, the remote command object is executed in the Pounamu Server and the view diagram stored there is updated (8). It's worth noting that we don't prescribe the format of a mirrored PounamuView image in this diagram. It can be any image format that is displayable in a web browser, including GIF, PNG, JPEG, and SVG. In this project, we are especially interested in two of them, GIF and SVG, and the reasons can be generalized as follows:

a) Compared with other image formats, GIF is the most popular and versatile format for distributing color images on the web. Moreover, it is the only graphic file format that is universally supported by all graphical browsers, regardless of version.
b) SVG, as a newly emerged web image format, uses an XML based specification to define web images [34]. This means it can offer all the advantages of XML, including interoperability, easy manipulation etc. Most importantly, SVG supports dynamic and interactive graphics far more sophisticated than bitmapped images. One downside is that SVG requires a plug-in to be downloaded and installed for the client PC web browsers. However, many newer browsers have this installed by default.

4.2 System Architecture

Software architecture forms the backbone for building successful software systems. It establishes communication among the system's stakeholders. In addition to conform to our OOA specification and non-functional constraints, designing a suitable "software architecture" also should take into account the following factors [35]:

- ♦ Allocation of processes to machines and programs
- ♦ Communication between processes or programs
- ♦ Synchronization of processing and data update
- ♦ Data storage & retrieval
- ♦ Processing performance

When designing a software architecture for our system, we mainly focus on the following considerations:

- ♦ Client/Server or non Client/Server architecture: if Client/Server, then 2-tier, 3-tier or n-tier
- ♦ Network situation between machines: LAN (Local Area Network) or WAN (Wide Area Network)
- ♦ Multi–user access: thread-safe protection
- ♦ System performance

- ♦ Ease of implementation, maintenance, and use
- ♦ High fault tolerance

4.2.1 System Deployment Diagram

The UML deployment diagram (shown in Figure 4-2) illustrates the system architecture based on a web-based N-tier architecture model. The first tier is a web browser, which acts as thin-client user interface. The second tier (i.e. web server tier) runs Servlets/JSPs to handle requests from users and provides the presentation logic. Since this tier is the most important part of our system, it is necessary to describe its interior architecture specifically, and this is done in the succeeding section. The third tier — Pounamu tool server — provides all the business functions and logic for this system. RMI/CORBA is middleware that connects the web tier and Pounamu tool server. The fourth tier is the file system that provides back-end persistent storage.

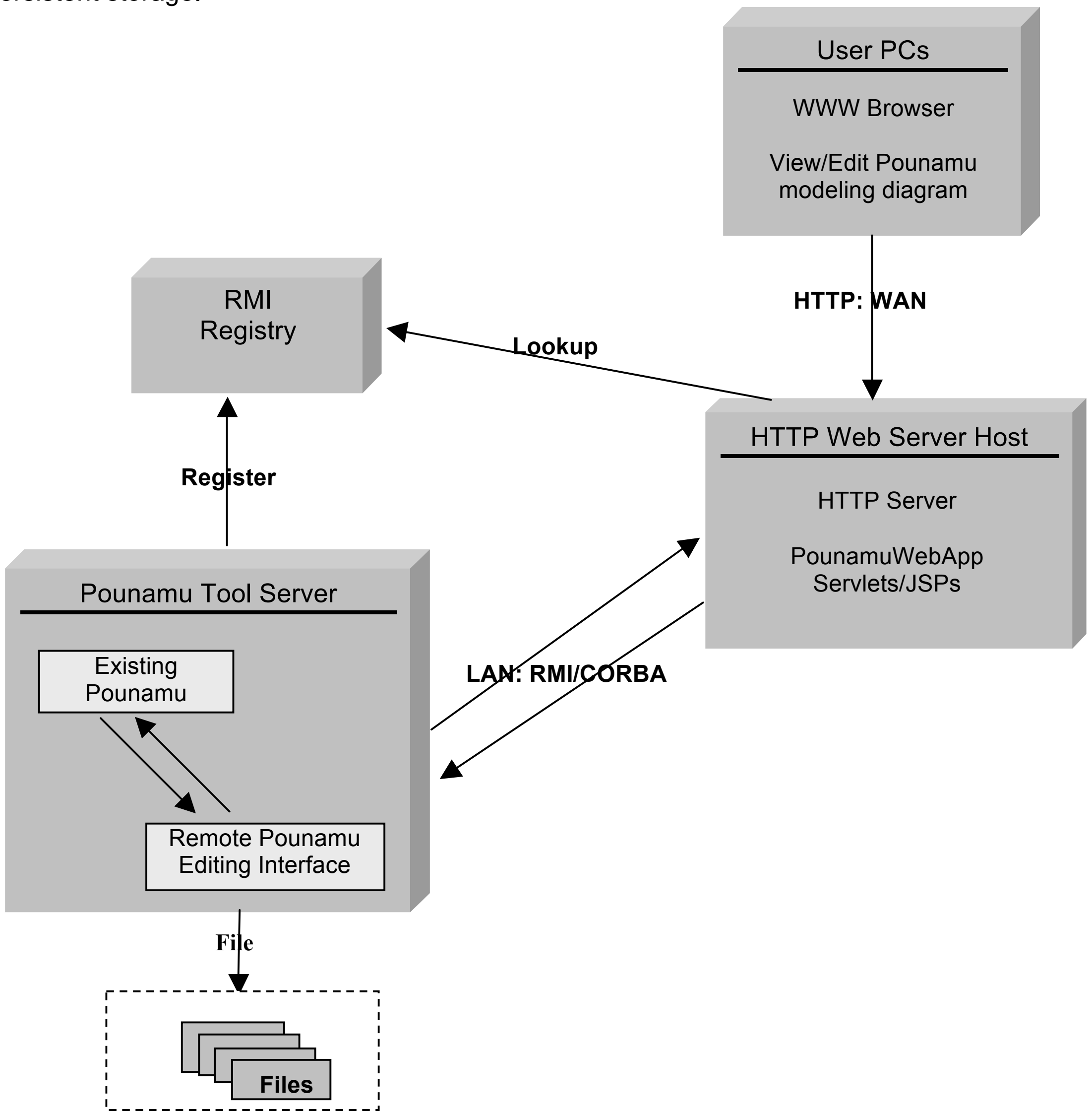


Figure 4-2 The System Architecture in a UML Deployment Diagram

4.2.2 Architecture Design of the Web Tier

Design of JSPs contained in the Web Server is often complicated by the need for JSPs to both perform requests handling, and present responses. These tasks are quite distinct for many pages, and request processing may be complicated. Generally, there are several ways (shown in Figure 4-3) to design JSP-based web applications [36]. The most basic is where we put all code: presentation mark-up, processing logic and data management, into the JSP script itself. This is a bad way to do things for many reasons, especially as it becomes unwieldy for all but the most trivial examples [36]. The second approach has HTML and Java processing /data management code in the JSP, with JavaBeans to hold data (b). This is slightly better, although putting processing logic and data management code into the JSP itself is a bad practice. It makes the JSP and associated code hard to reuse and ties the presentation and processing logic to a particular data management strategy, which may change during development [36]. A generally accepted approach, also the best one, is to have the JSP solely responsible for HTML mark-up and Java code to access bean content, with a Bean (or Servlet) responsible for handling screen processing logic, typically making use of other classes(c) [36]. However, even this best approach is not suitable for our purpose because our system has the following features:

(1) A central entry point is needed to manage application-wide resources (e.g. a reference to the remote Pounamu server object) or session state before responses can be generated;
(2) The fact that most requests come from a user's interaction with a mirrored image rather than an HTML form makes mapping request properties onto a bean difficult;
(3) The system is to a large extend process-intensive rather than presentation-intensive;

Encountering the above features is a clear indication that our web-tier needs a different architecture design.

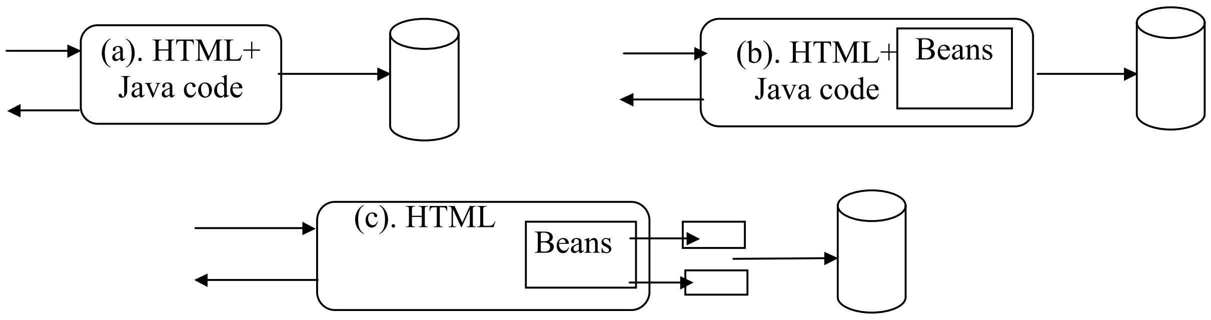


Figure 4-3 Three Ways to Design JSP-based WebApp [36]

Fortunately, there is an elegant solution, often called the "JSP Model 2 Architecture" or "Request Controller Architecture" [37,38]. This is a variant of MVC for web applications, and a sketch of this architecture is shown in Figure 4-4. From it, we can see this so-called "Request Controller Architecture" is a hybrid approach for serving dynamic content, since it combines the use of both Java Servlets and JSP. It takes advantages of the predominant strengths of both technologies, using JSP to generate the presentation layer and Servlets to perform process-intensive tasks. Here Servlets act as the controller and are in charge of the request processing and the creation and populating of worker beans used by JSPs, as well as deciding which JSP page to forward the request to. JSP is just responsible for retrieving information from worker beans that may have been previously populated by the Servlet and simply presenting it. Using this approach typically results in the cleanest separation of presentation from the business logic.

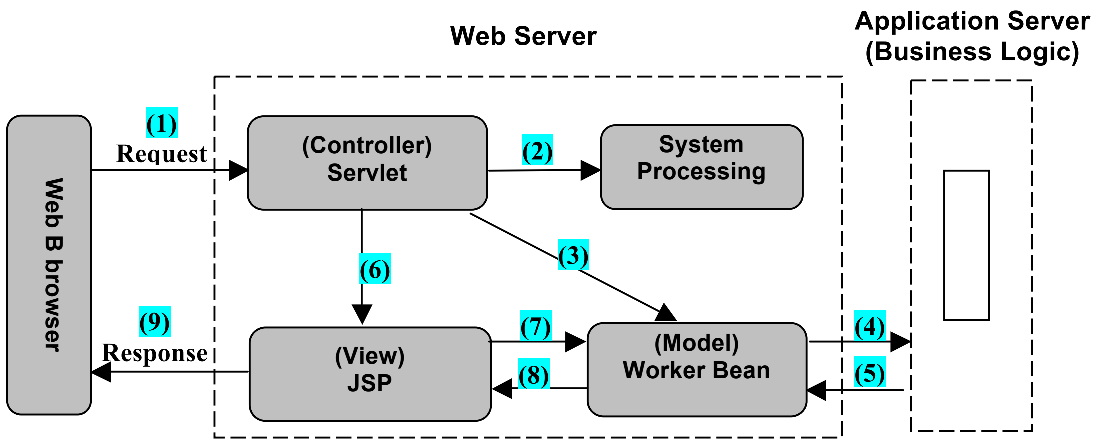


Figure 4-4 Request Controller Architecture (JSP Model 2 Architecture [38])

4.3 Object-Oriented Design

Having OOA objects and system architecture, we now move onto a more detailed OO design. This normally involves several steps:

- According to the architecture, we refine OOA classes into OOD objects by splitting up OOA objects into appropriate OOD classes;
- Introduce service classes, which are interfaces to the operating system including user interface libraries, middleware and data communication, data management (files, database, cache).
- Implement the management of inter-class data relationships represented in the OOA.

In the following sections, we will follow these steps to develop an OOD for our system.

4.3.1 Services Objects

Service objects often refer to specific APIs, which are used to access operating system support features such as user interface frameworks, communications and data management [36]. As our system is built on the top of an N-tier web-based architecture, the service classes determined in these tiers are illustrated in Figure 4-5, which is followed by a detailed description of these service objects in Table 4-1.

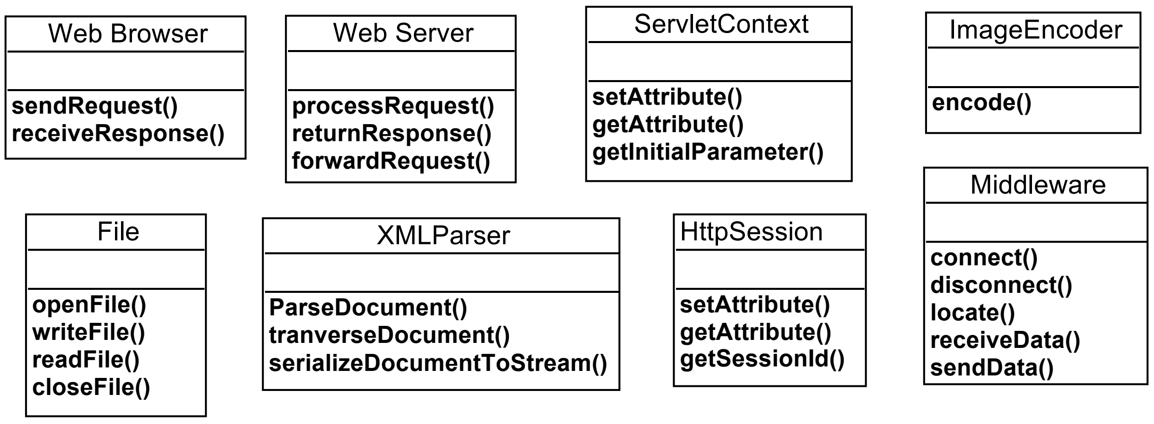


Figure 4-5 System Service Objects

Table 4-1 System Service Objects Description

- HTML browser service object: In the first tier of an N-Tier system, a web browser sends requests to a web server, obtains response from the web server and provides a user interface where users can interact with the system via HTML forms, text and images.
- Web Server service object: In the second tier of an N-Tier system, Servlets/JSPs receive requests from users, process these requests and prepare responses for display on the user's web browser.
- ServletContext service object: This object allows Servlets to store and retrieve the global resources sited on the Servlet engine of web server. For example, a reference to the remote server (stored on the Servlet engine environment during the initialization stage of a web application) can always be accessed through the ServletContext for a later remote procedure call.
- HTTPSession service object: This object can be used to transform the stateless HTTP protocol into an integrated seamless thread of activity. Actually, the HTTPSession object has been given the capability of associating data that is related to each user's session to it. The data can be profile information, user preferences, the current selections, etc.
- Middleware service object: Middleware provides a communication channel between the second tier and the third tier. The methods contained in this service object allow a client to locate and connect the remote server object, and transfer data to and from the server.
- XMLParser service object: In the system, information transferred between the second tier and third tier is in the form of an XML String. So, an XMLParser object is needed to parse the XML string back to the predefined java objects.
- ImageEncoder service object: This object helps to encode the generated image into the requested image format.
- Back-end storage service object: This object provides a dedicated service such as processing file I/O requests, management of file consistency and security, etc.

4.3.2 OOD of the Remote Interface Plug-in Component

Figure 4-6 illustrates a UML OOD diagram of the remote editing interface plug-in. Compared with the OOA class diagram (shown in Figure 3-14), we can see several main changes:

- An interface *RMIServerInterface* and class *RMIServer* which implements this interface have been added as a wrapper of the original class *RemotePounamuEditor*. This addition is based on our decision to use RMI as the communication protocol between the web server and the remote *Pounamu* server. The arrangement of having an *RMIServer* as a wrapper rather than putting all the remote function definitions into it directly means our code can be easily adapted to another communication protocol (e.g. CORBA) in the future.
- Two classes named *PounamuGIFGenerator* and *PounamuSVGGenerator* are also added. As their names suggest, *PounamuGIFGenerator* implements the functionality of generating a GIF version of mirrored image for a given PounamuView diagram, while *PounamuSVGGenerator* is used to obtain the SVG image document of a given view diagram. Both of them use other helper classes to achieve their goals.
- Owing to the decision to encode all the remote editing commands in XML and transfer this XML message over RMI, an OOD object *RemoteCommandDeserializer* is added to decode the XML message back into a *RemotePounamuCommand* object. Furthermore, *RemotePounamuCommand* and *RemoteCommandDeserializer* are extended to have an interface each. The reason behind this extension is that more than one type of remote command object exists in this system. By letting them have a common interface, we are actually writing more reusable and maintainable code. For example, if a new command object has to be added to the system at a later time, this can be done more gracefully without modifying the existing code.
- Based on eight basic editing commands that can be executed on a *PounamuView* diagram, we have designed their remote counterparts. Likewise, there should be eight specific classes implementing *RemoteCommandDeserializer*, with one for each possible remote command in the Pounamu tool. Names of remote Pounamu commands and their corresponding deserializer classes are below:

  RemoteAddAssociation — *RAADeserializer*

  RemoteAddEntity — *RAEDeserializer*

  RemoteChangeProperty — *RCPDeserializer*

  RemoteMoveConnector — *RMCDeserializer*

  RemoteMoveShape — *RMSDeserializer*

  RemoteRemoveEntity —*RREDeserializer*

  RemoteRemoveAssociation —*RRADeserializer*

  RemoteResizeShape— *RresSDeserializer*

Note: due to the limitation of page size, only one of the 8 remote command classes, as well as the corresponding deserializer class, is shown in Figure 4-6.

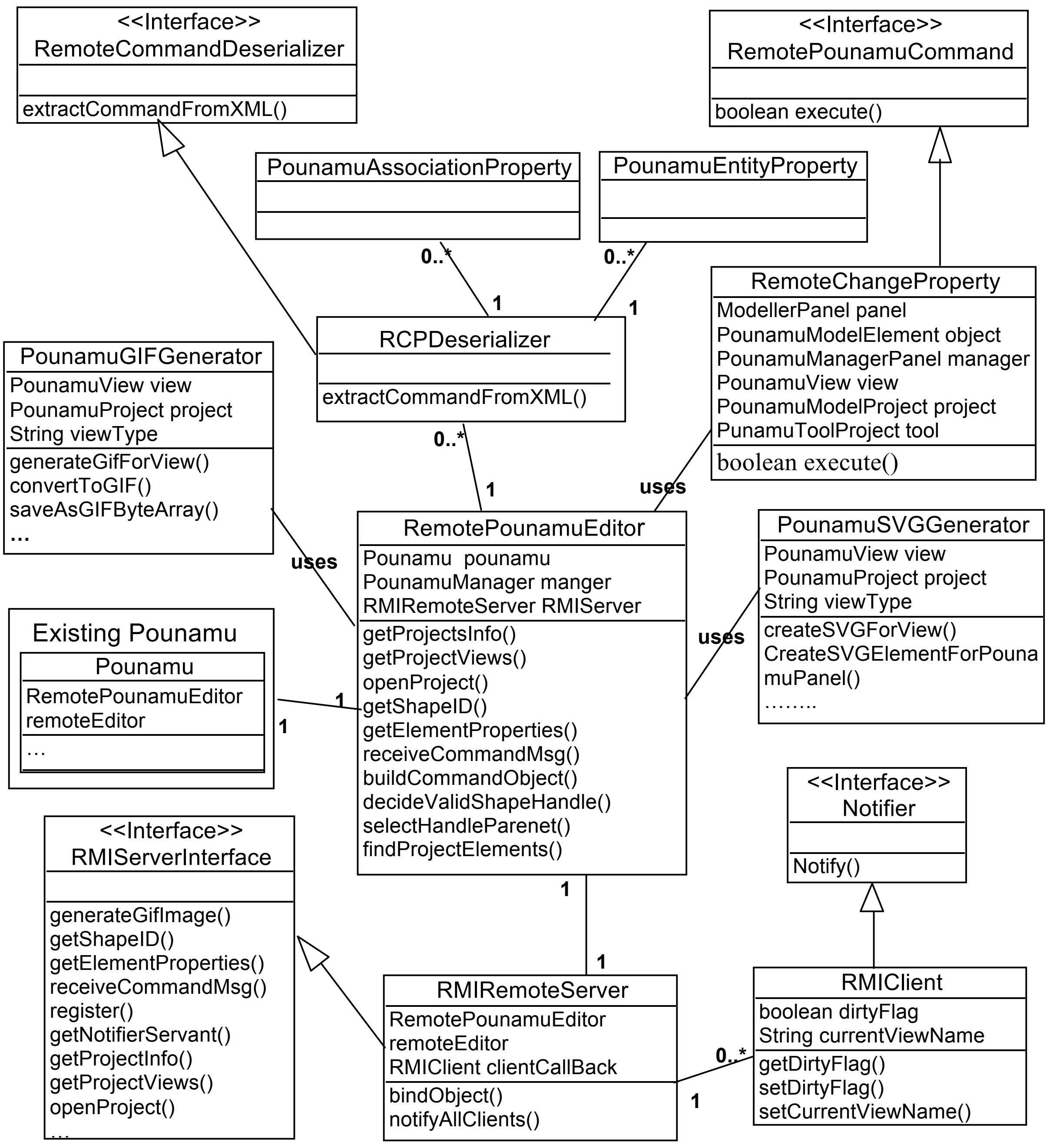


Figure 4-6 UML OOD Diagram of the Remote Interface Plug-in Component

4.3.3 OOD of the Web Tier

Until now, most design decisions that we have made are universal to both GIF- and SVG-based thin-client Pounamu user interfaces. However, considering that inherent image data representations and mouse clicks capturing/handling are so distinct between GIF and SVG, design decisions need to be made for the web tiers of the two versions, respectively. As a result, an individual subsection is specially written for each of these versions.

4.3.3.1 OOD for the web tier of the GIF-based thin-client user interface

Here three objects (*ActionData*, *ApplicationSession* and *PounamuImage*) identified at the OOA stage need be further split according their respective roles in the Request Controller Architecture. In addition, a *ControllerServlet* class is added to manage the application-scope resource such as a reference to the remote Pounamu *RMIServer*, as well as handle all incoming requests.

Controller Servlet

Design of the controller Servlet is a challenging task. As we already know, the instances of incoming requests from the system actor (here users) include: loading the working PounamuView, displaying the working PounamuView diagram, locating XY value of a user’s click on the image, as well as various editing requests on the working PounamuView etc. Handling all these in the controller Servlet would end up with a procedural class, and, more likely un-maintainable chains of if/else statements. A better approach would be to design a set of request handler classes to handle each user request, and the controller’s main responsibility is just to decide which request handler class is needed to handle a certain request and then delegate request processing to it. In this way, future enhancements of our system become very easy since new requests can be handled by adding the corresponding request handlers. Additionally, such design also ensures easier maintenance and code reuse. In our system, request handler dispatching is completed with the cooperation of *ControllerServlet* and *RequestController* classes. While *ControllerServlet* is an entry point for all user requests, *RequestController* uses Java reflection to instantiate request handlers as required. Moreover, for system efficiency, already instantiated request handlers are kept in a hash data structure so that they can be used to handle future requests of the same type. This arrangement ensures that use of reflection does not degrade the system’s performance significantly.

Request Handlers

Unlike the Controller Servlet, implementation of the RequestHandler should be application specific. Normally, a typical request handler needs to carry out the following tasks:

(1) Analyze the incoming request parameters
(2) Update the application status out of necessity
(3) Collect necessary data and make it available to JSP views for display

(4) Select a JSP view to which the controller will dispatch its response

In our implementation, to avoid being limited to handling requests currently envisaged, we have designed a *RequestHandler* interface. Thus, whenever needed, additional subclass can always be added without significant effect. Currently, there are 17 specific handler subclasses (shown in Table 4-2) to deal with all possible user requests:

Table 4-2 Available Handler Classes & Invoking Requests

Request handler Class Name	**User's Request**	**Response**
DisplayProjectsInfomation	Show the names of all the available model project	Query and return all the available project names
LoadModelProject	Load a working Pounamu Model Project	Set the selected model project as a working one
SelectPounamuView	Load a working PounamuView from the current working model project	Names of PounamuView and PounamuProject selected by user are stored into the Session object
GenerateGifImage	Display the selected Pounamu view diagram in web browser	Gif image of the matching PounamuView is generated and displayed
GetShapeID	Click a shape in the Gif image and ask for its ID	ShapeID at clicking position is determined and returned
GetElementXML	Ask for the detailed information about a shape	Shape's XML document is returned and parsed
AddAssociation	Initiate the editing action "Add an association type model element to the working view "	System enter the corresponding editing status by initializing and setting up the matching action data in Session object
AddEntity	Initiate the editing action "Add an entity type model element to the working view"	Similar to above
EditProperty	Initiate the editing action "Edit property values of the selected model element"	Similar to above
MoveEntity	Initiate the editing action "Move an entity type element to a new location"	Similar to above
MoveAssociation	Initiate the editing action "Move an associate type element"	Similar to above
RemoveModelElement	Initiate the editing action "Delete a model element from the view"	Similar to above
ResizeEntity	Initiate the editing action "Resize entity element"	Similar to above
LocateFirstMouseClickLocation	Locate a reference position for ResizeEntity editing action	The entity's resizing reference position is returned
	Locate the current position for a Will-be-moved shape.	The current position of a Will-be-moved shape is returned
	Locate one of the association' handles for MoveAssociation editing action	One handle of an association element has been selected for moving.

	Select an location where to put a will-be-added entity	The location for a will-be-added entity is determined.
	Locate the first shape handle for a will-be-added association	The first handle of a will-be-added association is located
LocateSecondMouseClickLocation	Locate a destination for a ResizeEntity editing action	The final location for the previously selected reference point is decided.
	Locate a new position for the will-be-moved shape.	A new location for the selected shape is returned.
	Locate an idea place for the handle of will-be-moved association	A new position for the previously located association handle is decided.
	Locate the second shape handle for a will-be-added association	The second handle of a will-be-added association is located
SubmiteChange	Submit any editing action made on the working PounamuView diagram.	An editing action is committed.
CanelChange	Cancel any editing action made on the working PounamuView diagram	An editing action is canceled.

Data Models

Data model objects provide a way of communicating between *RequestHandlers* and JSP views. With regard to our system, the *ApplicationSession* object, which holds information about the user's session (names of the working project and *PounamuView*, the current *EditingActionData* etc.), is an example of a data model object. Actually, *ApplicationSession* is a session bean. This bean will be declared in every JSP view since it exposes model data used by JSP views. However, in order to maintain session state related information, this bean will be instantiated in only one place — the *RequestController* class.

Another data model object to which we need to pay attention is the OOA class *EditngActionData*. As mentioned earlier, this object is mainly responsible for building the user's editing actions into XML-messages of remote Pounamu commands. Since there is more than one type of editing action in our system, in the OOD phase, an interface *ActionData* and 8 specific subclasses implementing this interface replace the previous OOA class. These subclasses are listed as the following:

AddAssociationActionData — Representing the "Add association element to the working view" editing action

AddEntityActionData — Representing the "Add an entity element to the working view" editing action

ChangePropertyActionData — Representing the "Change property values of the existing model element" editing action

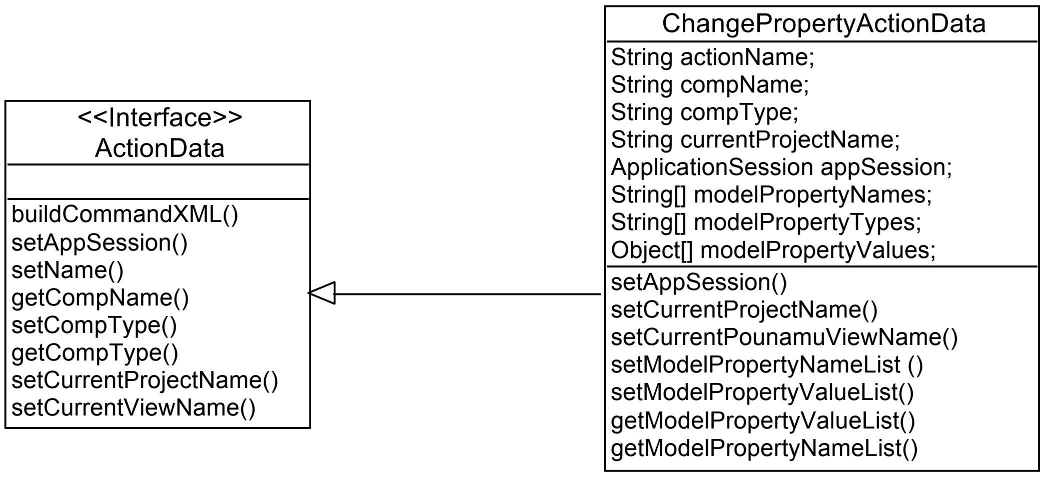
MoveShapeActionData — Representing the "Move an existing shape to a new location" editing action

MoveAssociationActionData — Representing the "Move an association element" editing action

RemoveActionData — Representing the "Delete an existing model element" editing action

ResizeShapeActionData — Representing the "Resize an existing shape" editing action

To show what an *ActionData* class looks like, *ChangePropertyActionData* is given below as an example:


<<Interface>>
ActionData
buildCommandXML()
setAppSession()
setName()
getCompName()
setCompType()
getCompType()
setCurrentProjectName()
setCurrentViewName()
ChangePropertyActionData
String actionName;
String compName;
String compType;
String currentProjectName;
ApplicationSession appSession;
String[] modelPropertyNames;
String[] modelPropertyTypes;
Object[] modelPropertyValues;
setAppSession()
setCurrentProjectName()
setCurrentPounamuViewName()
setModelPropertyNameList ()
setModelPropertyValueList()
getModelPropertyValueList()
getModelPropertyNameList()

Finally, a UML OOD class diagram of the web-tier is presented in Figure 4-7. Due to the limitation of page size, only 2 of the 18 *RequestHandler* classes and 2 of the 7 *ActionData* classes are shown in this diagram.

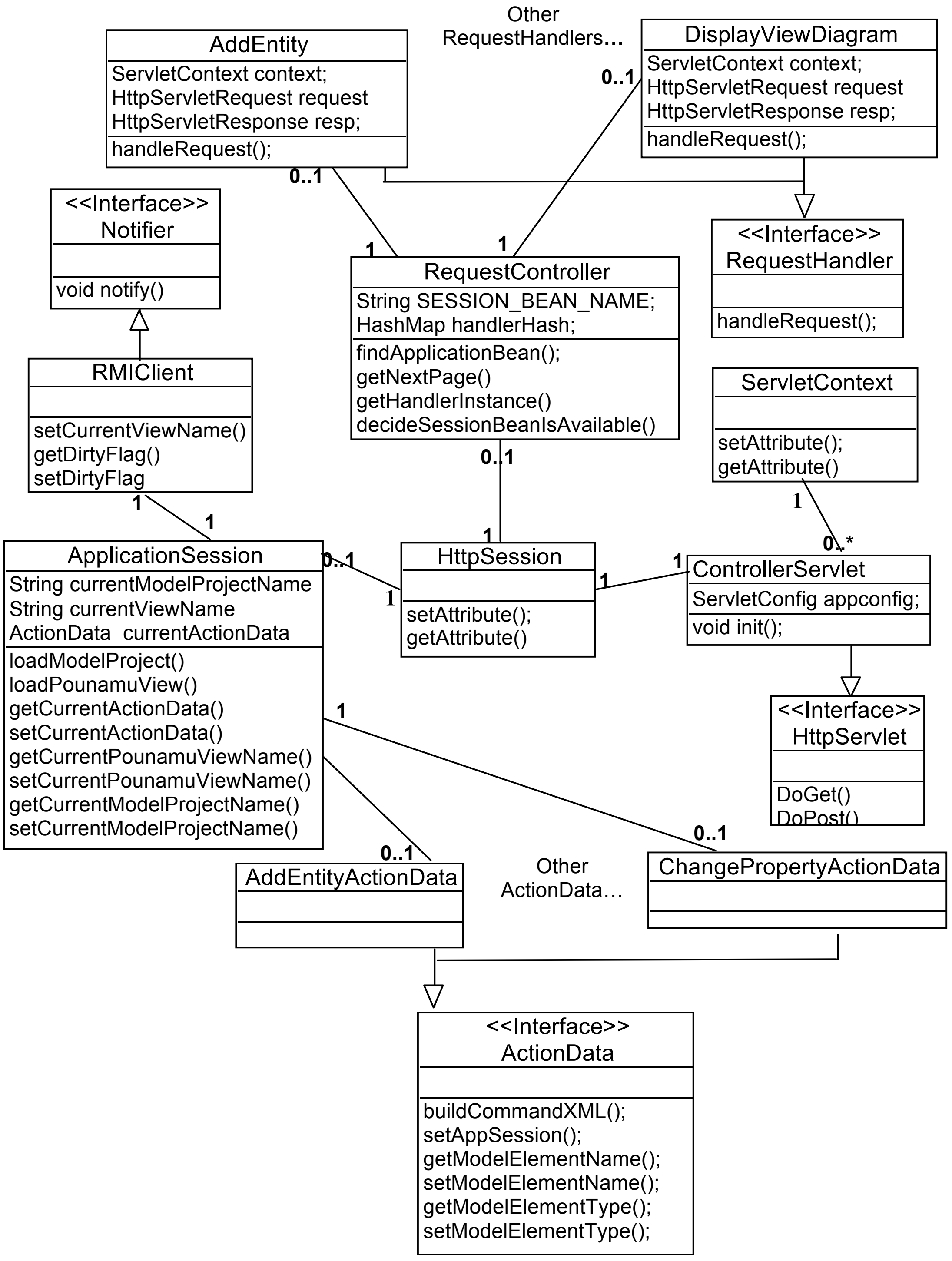


Figure 4-7 UML OOD Class Diagram of the Web tier

4.3.3.2 OOD for the web tier of the SVG-based thin-client user interface

After careful design consideration, in this subsection, we arrive at similar OOD objects as those obtained in the previous GIF version thin-client Pounamu tool. However, following further investigation, significant changes have been made on some methods contained in this SVG version of the OOD classes. Since most changes are made to enable multiple editing on a PounamuView diagram via a web browser, a brief introduction to what multiple editing is and how SVG helps to make multiple editing possible is presented before the main task of design changes discussion.

Multiple Editing Concept

Figure 4-8 shows scenarios of multiple editing (a) and single editing (b). In multiple editing, a series of editing actions merely involves the web browser and the web server, and editing actions buffered at the web server are executed on the Pounamu server in batch form, while for single editing, every editing action involves all three sides (the web browser, the web server and the Pounamu server) and each editing action is executed immediately on the Pounamu server instead of buffering. The aim is for multiple editing to significantly improve system performance by reducing communication latency. Also, in multiple editing, a set of edits by each user can be done without knowledge of another set of user-buffered edits. “Transactions” of edits per user is achievable in this editing mode.

Although multiple editing is has preference over the other, two prerequisites have to be met to make multiple editing on a PounamuView diagram possible. First, the web server must contain an information repository so that editing action related data (such as ID, location of shapes or connector etc.) is retrievable, otherwise, the remote Pounamu server has to be inquired for this information. Second, sketches representing each cached editing have to be overlaid on the original diagram so that users clearly know what editings have been made to the diagram. In order to draw these sketches, the original diagram should allow a certain degree of manipulation on the web server. Simply because of this second requirement (just imagining how difficult it would be to manipulate a GIF image’s binary data), multiple editing functionally has been ruled out of the GIF-version of the thin-client Pounamu user interface. In contrast, SVG’s adoption of plain text description instead of binary data allows arbitrary manipulation of image data, and therefore multiple editing is implementable. A detailed explanation on why and how to enable multiple editing in the SVG version thin-client tool ensues in a later paragraph.

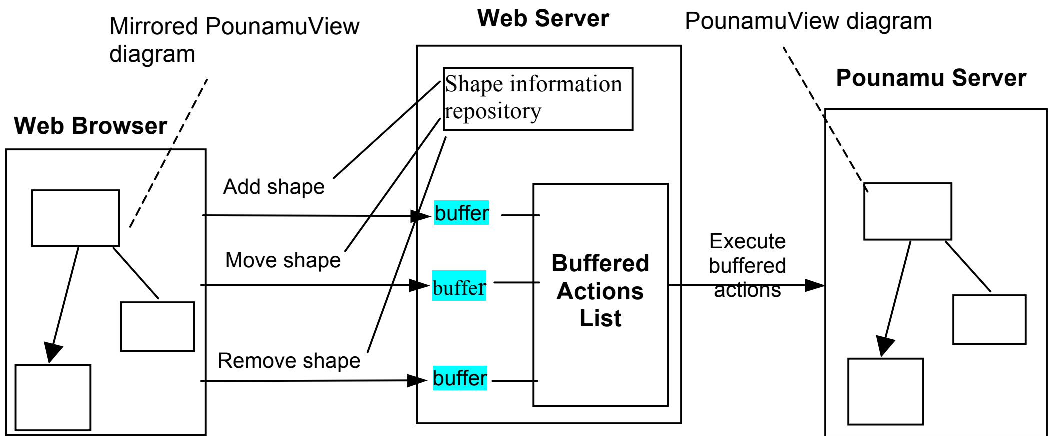


Figure 4-8(a) Scenario of Multiple Editing on a PounamuView Diagram

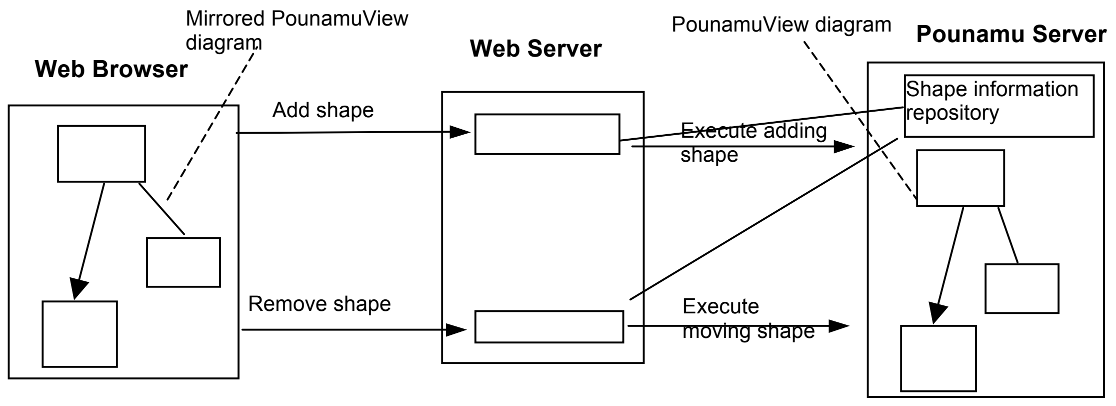


Figure 4-8(b) Scenario of Single Editing on a PounamuView Diagram

SVG-based Multiple Editing

Scalable Vector Graphics (SVG) [39] is a newly emerged web based graphics. It uses the XML grammar to describe two-dimensional graphic objects such as lines, texts, and polygons, so each individual graphics object can be manipulated in a similar way that XMI elements, attributes and properties are manipulated through the DOM API and XMLParser. Generally speaking, there are two main reasons why it is possible to enable multiple editing in this SVG-based thin-client diagramming tool:

(1) SVG images generated for PounamuView diagrams can package and deliver all the information (including ID, location, and properties of shapes and connectors) needed for multiple editing in our familiar XML format. By comparison, the remote Pounamu server needs to be referred to repeatedly for such information in our GIF version thin-client tool.

(2) Since we can manipulate an SVG image as we do a normal XML document, sketches of any editing action such as moving an entity or adding an entity can be easily added and removed from the original image. By comparison, drawing additional editing sketches on a GIF binary content may be possible, but extremely difficult.

Another important feature that makes people interested in using SVG web graphics is its interactivity. This interactivity is supported by assigning a rich set of event handlers to individual graphics objects in an SVG document through the DOM (Document Object Model) interface. In our system, *locateShapeHandle LocateCanvasLocation*, and *getShapeID* are examples of special purpose event handlers, and a description of these handlers is given below:

GetShapeID: Identify the shape or connector that was clicked;

LocateCanvasLocation: Obtain X,Y values of the canvas point that was clicked;

locateShapeHandle: Locate and return a unique shape handle ID (Note: Shape handles are special features that Pounamu diagrams have, which are useful for adding connectors and other editing actions)

Design Changes

As a result of splitting three OOA level objects (*ActionData*, *ApplicationSession* and *PounamuImage*) according to their respective roles in the Request Controller Architecture, we come up with similar OOD objects to those obtained in the previous GIF version of a thin-client Pounamu tool. These can be generalized as follows:

- *ControllerServlet*: An OOD object used to manage application-scope resources, as well as handle all incoming requests
- *RequestHandler* interface and its application specific subclasses: OOD objects used to analyze request parameters, update application status equally, and select proper JSP views to forward response.
- Data model classes (*ApplicationSession*, *ActionData* and its 8 specific subclasses): the *ApplicationSession* object assists the communication between *RequestHandlers* and JSP views (i.e., *ApplicationSession* will be populated by *RequestHandlers* and used by JSP views). Each of the 8 *ActionData* subclasses represents one type of editing action.

In the following paragraphs, more detailed descriptions of these objects are provided focusing on the differences to their GIF version counterparts.

(1) ControllerServlet

This object is the same as its GIF version counterpart.

(2) Application Specific RequestHandlers

Compared with the previous GIF version thin-client tool, a different mechanism is used here to capture and handle mouse click events, so *RequestHandler* subclasses are required to change correspondingly. *RequestHandlers* needed for this SVG version tool are summarized in Table 4-3.

Table 4-3 Required RequestHandlers & Invoking Requests

Request handler Class Name	*User's Request*	*Response*
LoadModelProject	Load a working PounamuModelProject	Set the selected model project as a working one
SelectPounamuView	Choose and load a working PounamuView from the current working model project	Names of PounamuView and PounamuProject selected by user are stored into the Session object
GenerateSVGImage	Display the selected PounamuView diagram in SVG format on web browser	SVG image of the requested diagram is generated and displayed
AddAssociation	Initiate the editing action "Add an association type model element to the working view "	System enters the corresponding editing status by initializing and setting up the matching action data in Session object
AddEntity	Initiate the editing action "Add an entity type model element to the working view"	Similar to above

SetProperties	Initiate the editing action "Set property for the selected model element"	Similar to above
MoveEntity	Initiate the editing action "Move an entity type element to a new location"	Similar to above
MoveAssociation	Initiate the editing action "Move an associate type element"	Similar to above
RemoveModelElement	Initiate the editing action "Delete a model element from the view"	Similar to above
ResizeEntity	Initiate the editing action "Resize entity element"	Similar to above
GetShapeID	Click a shape in the SVG image to request its ID value (needed for almost all editing actions)	ID of the clicked shape is obtained. System responds differently to different editing actions.
locateShapeHandle	Click and locate a preferred shape handle (needed for add association, resize entity actions)	Shape handle ID is obtained. For each of possible editing actions matching responses are sent out accordingly.
locateMouseClickOnCanvas	Click and locate a preferred position on the displayed SVG image canvas (needed for add, move and resize entity actions)	Mouse click's coordinates XY are obtained. How these XY are stored will depend what type of editing action it is.
GetElementProperties	Request visual and model properties about a shape	Shape's XML document is returned and parsed
BufferChange	Buffer a just-completed editing action	Current editing action is added to a buffered action list.
CancelChange	Cancel a just-completed editing action	Current editing action is not buffered for later execution.
ModifyBufferedChanges	Review a buffered editing list and delete unwanted editings.	A buffered editing action list is updated.
SubmitBufferedChanges	Submit the buffered editing action list for execution.	Editing actions are committed on the remote server and the updated view diagram is displayed.

(3) Data Model Classes

Figure 4-9 shows the ApplicationSession object developed for the GIF-version (a) and the SVG-version (b) of thin-client Pounamu user interfaces. Differences are highlighted in bold font. For example, in (a), the cached GIF-format of a *PounamuView* diagram is stored as a byte array, in contrast, the cached SVG-format of a *PounamuView* diagram is stored as a XML String object. Another difference is that the *manipulateSVGImageDoc()* method has been added to the SVG-version *ApplicationSession object* to manipulate the cached SVG image through standard APIs (XMLParser and DOM interface). There is a working scenario to show why this manipulation functionality is necessary. Assuming the user has selected a shape to reset its properties, or simply to remove it. The displayed view diagram needs to be able to reflect the user's selection. If no SVG manipulation is allowed on the web server, the remote Pounamu server has to be revisited to generate a diagram with the selected shape highlighted.

Having explained how the two versions of ApplicationSession object are different, we now take a look at other pairs of objects (*ActionData* and its subclasses). Figure 4-10 shows the GIF-version (a) and the SVG-version (b) *ActionData* and one of its subclass. Comparing (a) with (b), you can notice two new methods (*drawSkectch()* and *removeSketch()*) highlighted in bold font have been added to the SVG-version *ActionData*. *DrawSketch()* is called to add extra sketches to the original diagram to visually represent a specific editing action, while removeSketch() is called to remove any editing-related sketches from the diagram if a buffered editing is canceled. As stated earlier, drawing edit sketches onto a view diagram is especially useful for multiple editing, so users can clearly see what editing actions have been made and buffered. Of course, any buffered editing actions can be reviewed and deleted if the user chooses to do so.

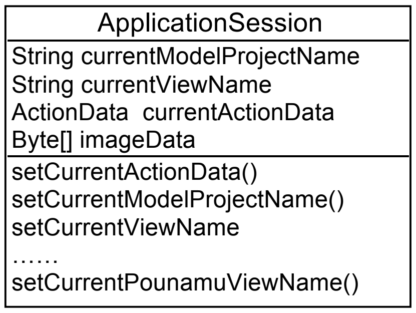

ApplicationSession
String currentModelProjectName
String currentViewName
ActionData currentActionData
Byte[] imageData
setCurrentActionData()
setCurrentModelProjectName()
setCurrentViewName
……
setCurrentPounamuViewName()

Figure 4-9 (a) the GIF version ApplicationSession OOD Object

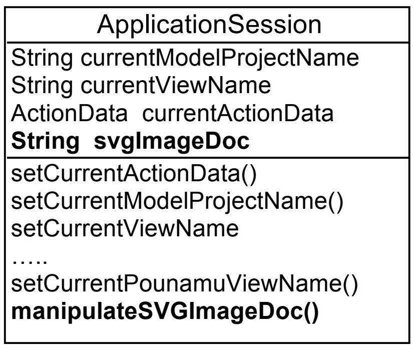


Figure 4-9 (b) the SVG version ApplicationSession OOD Object

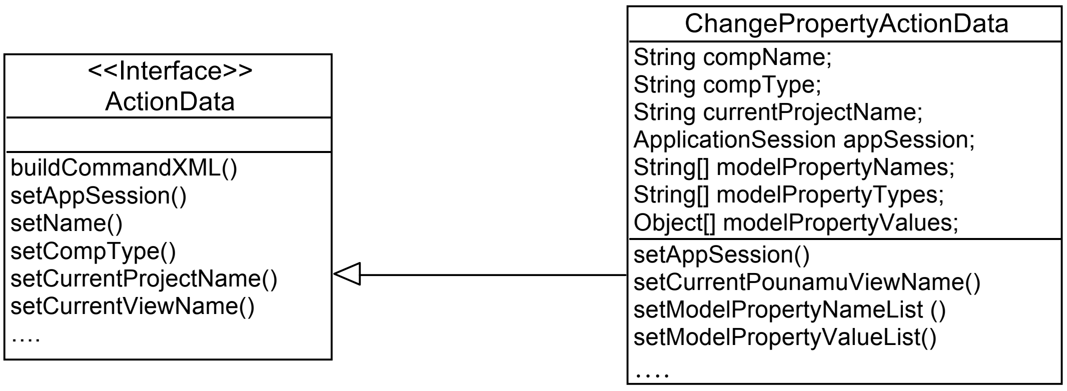

<<Interface>>
ActionData
buildCommandXML()
setAppSession()
setName()
setCompType()
setCurrentProjectName()
setCurrentViewName()
….
ChangePropertyActionData
String compName;
String compType;
String currentProjectName;
ApplicationSession appSession;
String[] modelPropertyNames;
String[] modelPropertyTypes;
Object[] modelPropertyValues;
setAppSession()
setCurrentPounamuViewName()
setModelPropertyNameList ()
setModelPropertyValueList()
….

Figure 4-10(a) the GIF version ActionData OOD Object and one Subclass Example

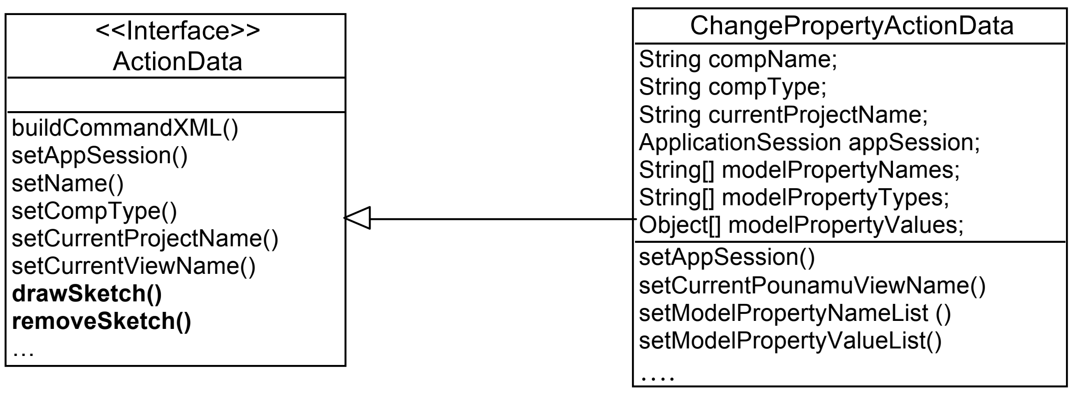

<<Interface>>
ActionData
buildCommandXML()
setAppSession()
setName()
setCompType()
setCurrentProjectName()
setCurrentViewName()
drawSketch()
removeSketch()
…
ChangePropertyActionData
String compName;
String compType;
String currentProjectName;
ApplicationSession appSession;
String[] modelPropertyNames;
String[] modelPropertyTypes;
Object[] modelPropertyValues;
setAppSession()
setCurrentPounamuViewName()
setModelPropertyNameList ()
setModelPropertyValueList()
….

Figure 4-10(b) the SVG version ActionData OOD Object and one Subclass Example

4.3.4 Data & Message passing among OOD objects

While the above sections describe static objects and relationships among these objects, this section will focus on documenting dynamic system behavior through some typical scenarios in our system, i.e.

- Adding a remote interface plug-in to the existing Pounamu tool;
- Registering a user;
- Loading a PounamuView diagram;
- Initiating an editing action;
- Confirming editing action on the working PounamuView diagram;
- Identifying a shape/connector that the current editing acts upon;
- Locating a specific shape handle;
- Setting properties of a model element;

The above scenarios will be described through collaboration diagrams or sequence diagrams illustrated below.

4.3.4.1 Sequence Diagram for "Adding a remote interface plug-in"

Shown below is a sequence diagram capturing the basic interactions involved when adding a remote editing interface plug-in to the Pounamu software tool.

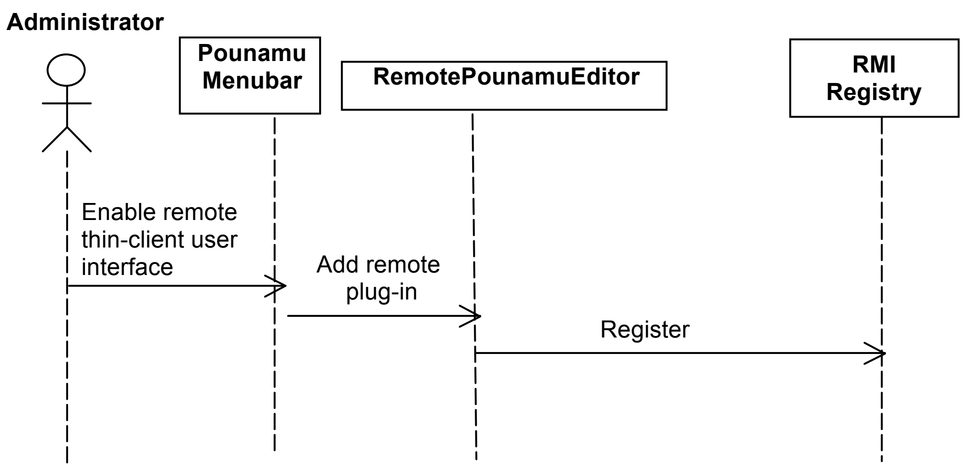


Figure 4-11 Sequence Diagram for Adding the Remote Editing Interface Plug-in

4.3.4.2 Event flows for "Registering a User"

Only the authorized users can access and edit PounamuView diagrams in a named model project, so any user has to login to the system first. The following collaboration diagram (Figure 4-12) depicts a typical user login process, and Table 4-4 gives a detailed description for each step annotated in the diagram.

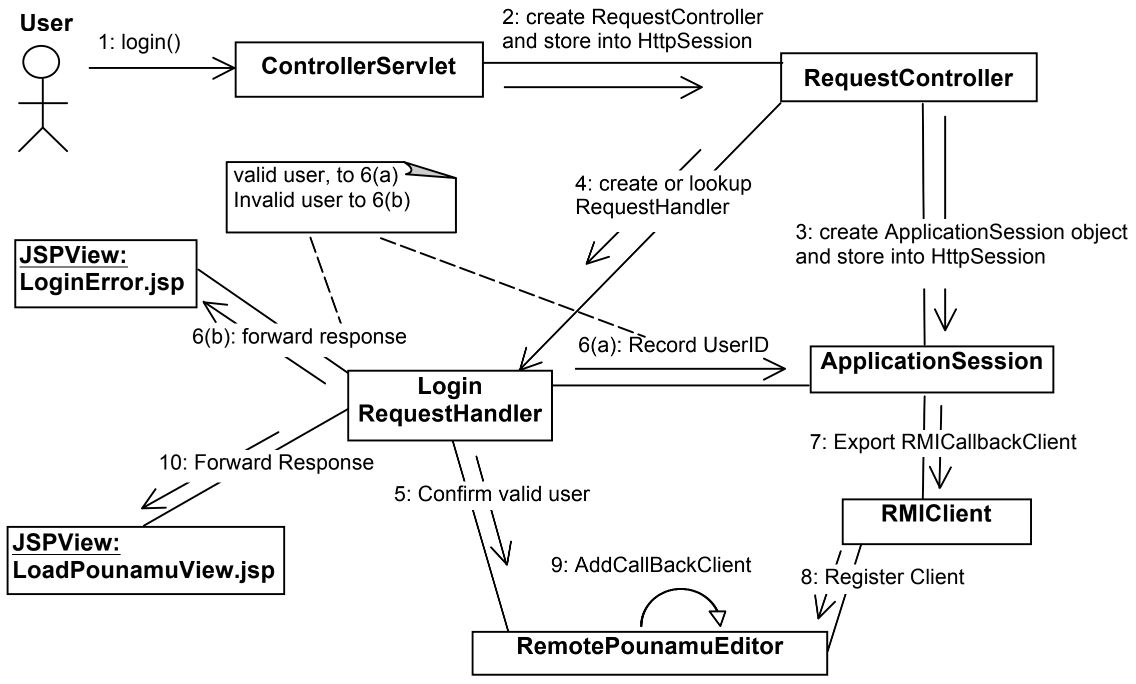


Figure 4-12 Collaboration Diagram for Registering a User

Table 4-4 Descriptions of events shown in Figure 4-12

Step	Description	
1	Invoked when a participant asks for accessing to Pounamu tool via web browser.	
2	Invoked by step 1: Retrieve the RequestController object from HTTPSession. This object is in charge of instantiating or looking up an appropriate handler object to deal with the user's login requests.	
3	Invoked by step 2: Create an application session object for this specific participant.	
4	Invoked by step 2: A special-purpose request handler object is created if not existed, or retrieved if already there.	
5	Invoked by step 4: The verifyUser() method of the Pounamu server is called so that the user's identification is checked against the user information file stored there.	
6	(a)	Invoked by step 4: Upon a valid user, his/her user ID is stored into the application session object.
	(b)	Otherwise, response is forwarded to the login error page.
7	Invoked by step 6(a): For a valid user, an RMICallBackClient object is instantiated and exported in the RMI registry.	
8	Invoked by step 7: This newly instantiated RMIClient also registers itself with the Pounamu server for the callback service when a certain Pounamuview diagram is updated.	
9	Invoked by step 8: The Pounamu server will append this RMIClient at the end of its callback list.	
10	Invoked by step 4: Forward the system response to the loadPounamuView.jsp page.	

4.3.4.3 Event flows for “Loading a Working PounamuView Diagram”

The fact that a PounamuView diagram belongs to a Pounamu model project means that to load a working PounamuView diagram, a model project has to be loaded first. So two collaboration diagrams are drawn here to describe the scenario of "Loading a Working PounamuView diagram". Figure 4-13 (a) describes event flows when selecting and loading a Pounamu model project, and Figure 4-13(b) describes event flows when setting and displaying a working PounamuView diagram. In addition, Table 4-5 and Table 4-6 explain each individual step of these two diagrams.

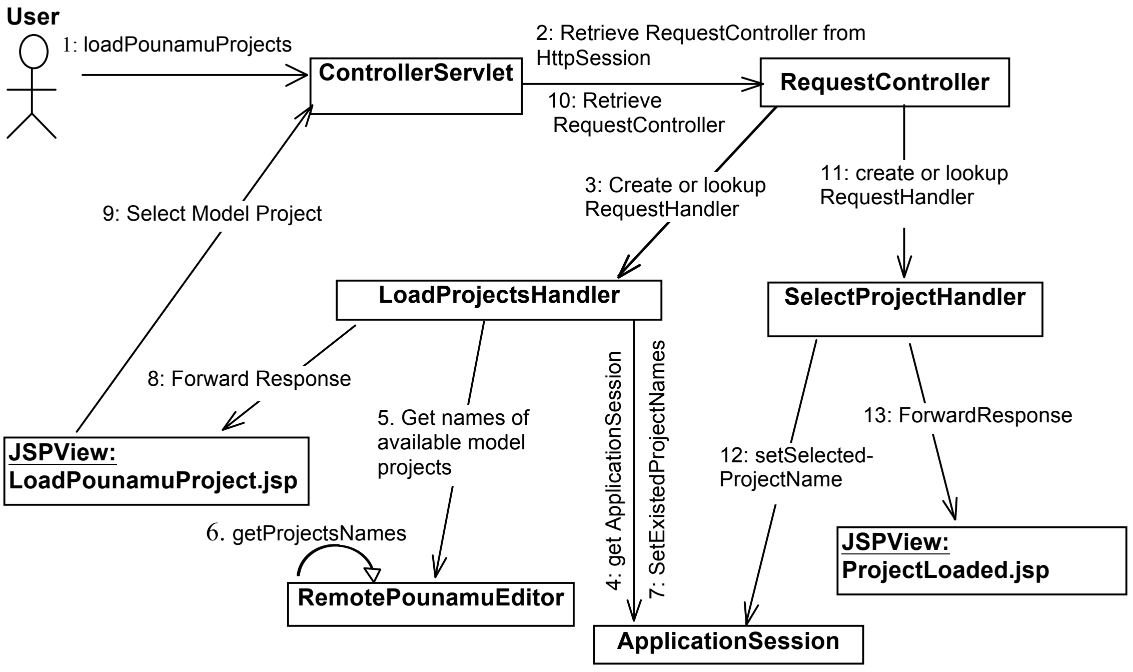

User
1: loadPounamuProjects
ControllerServlet
2: Retrieve RequestController from HttpSession
10: Retrieve RequestController
RequestController
3: Create or lookup RequestHandler
11: create or lookup RequestHandler
9: Select Model Project
LoadProjectsHandler
SelectProjectHandler
8: Forward Response
5. Get names of available model projects
4: get ApplicationSession
7: SetExistedProjectNames
12: setSelected-ProjectName
13: ForwardResponse
JSPView: LoadPounamuProject.jsp
JSPView: ProjectLoaded.jsp
6. getProjectsNames
RemotePounamuEditor
ApplicationSession

Figure 4-13(a) Collaboration Diagram for Loading a Working Model Project

Table 4-5 Descriptions of events shown in Figure 4-13(a)

Step	Description
1	Invoked when a user initiates loading a model project from Pounamu server
2	Invoked by step 1: Retrieve the RequestController object from HTTPSession. This object is in charge of instantiating or looking up an appropriate handler object to deal with the user's loading requests.
3	Invoked by step 2: A special-purpose request handler object is created if not existed, or retrieved if already there.
4	Invoked by step 3: Retrieve the user's application session object from HTTPSession.
5	Invoked by step 3: Obtain all available model projects stored in the Pounamu server by calling the getProjectNames() method.
6	Invoked by step 5: It's the getProjectsNames() method of RemotePounamuEditor that is actually executed to return names of all existing model projects.
7	Invoked by step 3: Store names of all the model projects existed on the Pounamu server.
8	Invoked by step 3: Forward the response to loadPounamuProject JSP page.
9	Invoked by selecting a specific model project.
10	Invoked by step9: Retrieve the RequestController object from HTTPSession. This object is in charge of instantiating or looking up an appropriate handler object to deal with the project selection request.
11	Invoked by step 10: A special-purpose request handler object is created if not existed, or

	retrieved if already there.
12	Invoked by step 11: Store the user's choice of model project into the application session object.
13	Invoked by step 12: Forward the system response to the projectLoaded.jsp page.

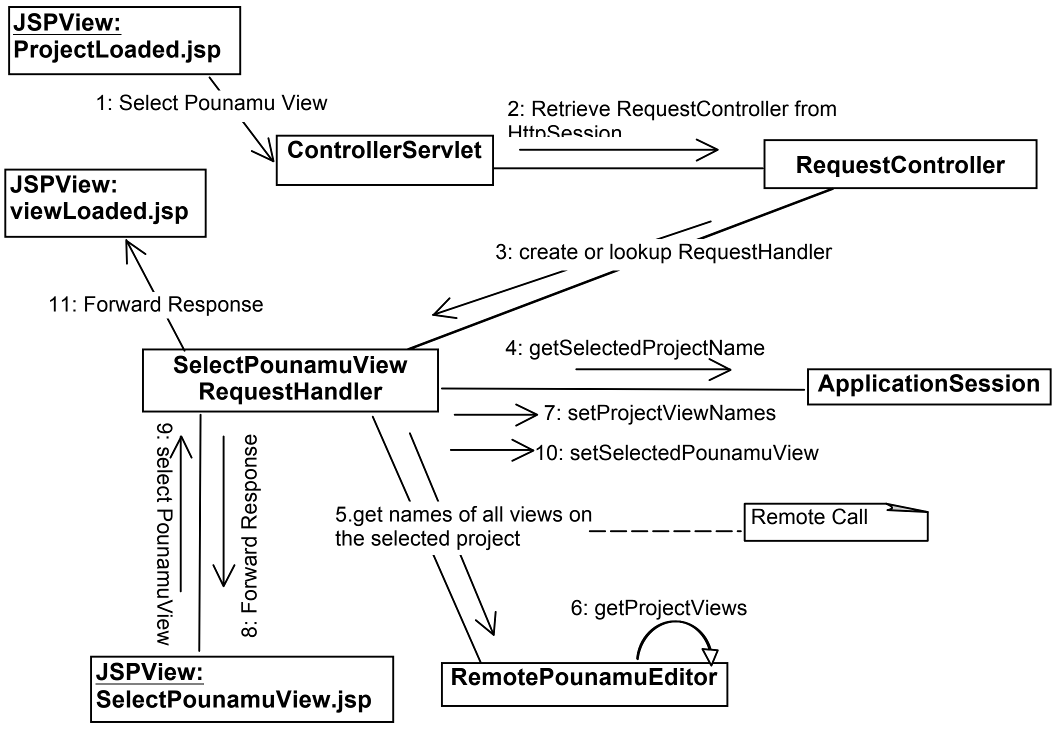


Figure 4-13(b) Collaboration Diagram for Loading and Displaying a Working PounamuView

Table 4-6 Descriptions of events shown in Figure 4-13 (b)

Step	Description
1	Invoked when a user initiates loading a PounamuView contained in the previously specified Pounamu model project
2	Invoked by step 1: Retrieve the RequestController object from HTTPSession. This object is in charge of instantiating or looking up an appropriate handler object to deal with the user's loading requests.
3	Invoked by step 2: A special-purpose request handler object is created if not existed, or retrieved if already there.
4	Invoked by step 3: Retrieve the selected model project from the user's application session
5	Invoked by step 3: Make a remote procedure call to obtain names of all the PounamuView diagrams contained in the selected project
6	Invoked by step 5: It is the getProjectViews() method of the RemotePounamuEditor object that is responsible for returning view names.
7	Invoked by step 3: Store the returned names into the application session for later presentation.
8	Invoked by step 3: Forward response to the SelectPounamuView.jsp page for exposing selectable Pounamu view diagrams
9	Invoked by step 3: Select a PounamuView diagram from a list of choices exposed via the SelectPounamuView.jsp view.
10	Invoked by step 9: Store the user's choice into the session state.
11	Invoked by step 9: Embed the selected PounamuView diagram in a JSP page for displaying

4.3.4.4 Event flows for "Initiating an Editing Action"

This is a common scenario for all the possible editing actions a user would take. The companying collaboration diagram (shown in Figure 4-14) describes events occurring whenever a user initiates an editing action by clicking the corresponding button on the main menu bar. A detailed description for each step involved in this diagram can be seen in Table 4-7.

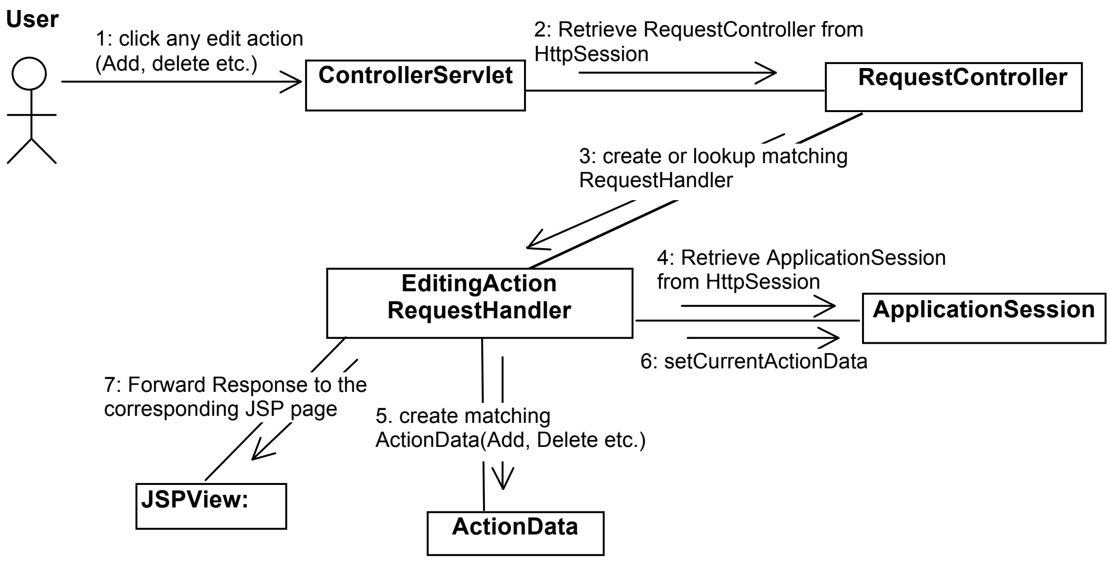

User
1: click any edit action (Add, delete etc.)
ControllerServlet
2: Retrieve RequestController from HttpSession
RequestController
3: create or lookup matching RequestHandler
EditingAction RequestHandler
4: Retrieve ApplicationSession from HttpSession
ApplicationSession
6: setCurrentActionData
7: Forward Response to the corresponding JSP page
5. create matching ActionData(Add, Delete etc.)
JSPView:
ActionData

Figure 4-14 Collaboration Diagram for Initiating an Editing Action

Table 4-7 Descriptions of events shown in Figure 4-14

Step	Description
1	Invoked when the user clicks an editing menu item (on the left menu bar) to start any possible editing action.
2	Invoked by step 1: Retrieve the RequestController object from the service object HTTPSession, and this object is in charge of instantiating or looking up an appropriate handler object to deal with the user's editing request made via a web browser.
3	Invoked by step 2: A special-purpose RequestHandler object is created if not existed, or retrieved if already there.
4	Invoked by step 3: Retrieve the user's application session object from HTTPSession, and this object is used to store editing related information, such as the command type, the name and properties of the target element etc.
5	Invoked by step 3: A new editing action data of a certain type is created.
6	Invoked by step 3: Set the newly created editing action data as the current action data and store it for later reference.
7	Invoked by step 3: Select a proper JSP page to forward the execution response.

4.3.4.5 Event flows for "Confirming an Editing Action on the Working PounamuView Diagram"

Similar to the above “Initiating an Editing Action”, this is also a common scenario relating to all editing actions. Figure 4-15 shows the collaboration diagram for such a scenario, illustrating events that take place when a user finally decides to submit the current editing action in his/her session. Table 4-8 provides a detailed description of every step depicted in this event flow diagram.

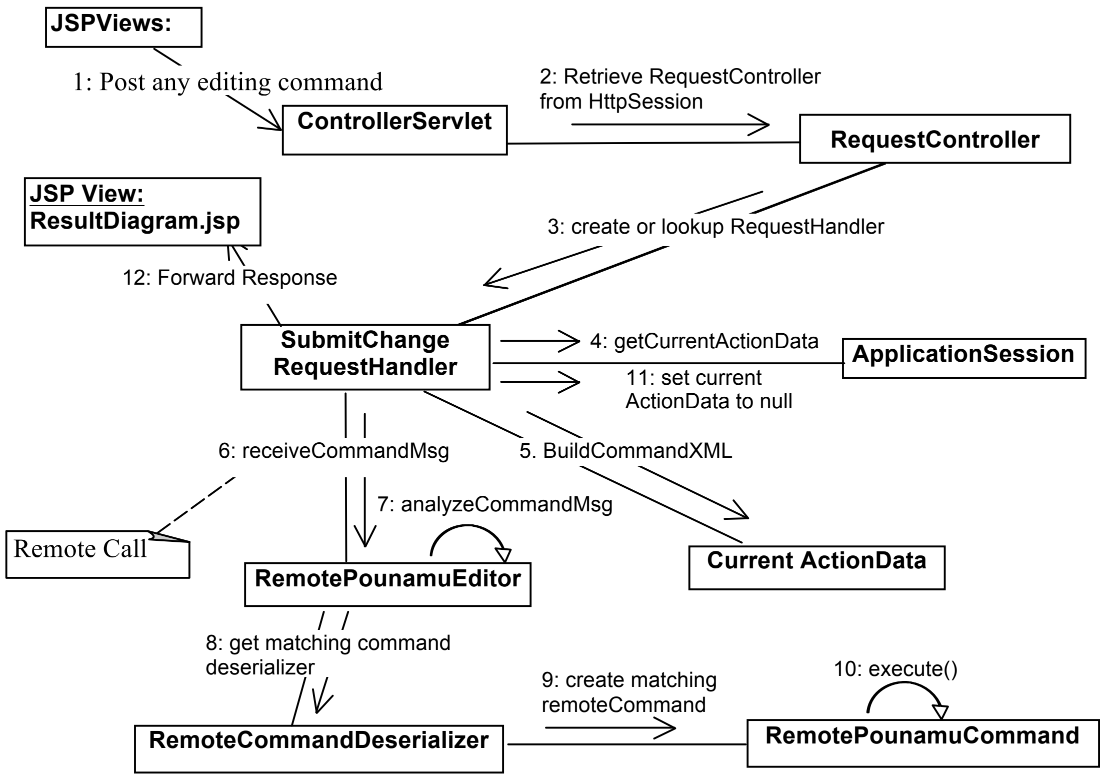

JSPViews:
1: Post any editing command
ControllerServlet
2: Retrieve RequestController from HttpSession
RequestController
JSP View: ResultDiagram.jsp
3: create or lookup RequestHandler
12: Forward Response
SubmitChange RequestHandler
4: getCurrentActionData
ApplicationSession
11: set current ActionData to null
6: receiveCommandMsg
5. BuildCommandXML
7: analyzeCommandMsg
Remote Call
RemotePounamuEditor
Current ActionData
8: get matching command deserializer
9: create matching remoteCommand
10: execute()
RemoteCommandDeserializer
RemotePounamuCommand

Figure 4-15 Collaboration Diagram for Confirming an Editing Action

Table 4-8 Descriptions of events shown in Figure 4-15

Step	Description
1	Invoked when the user decides to submit current editing action data stored in his/her application session object.
2	Invoked by step 1: Retrieve the RequestController object from the service object HTTPSession. This RequestController is in charge of instantiating or looking up appropriate handler object to deal with the user’s requests sent via web browser.
3	Invoked by step 2: A special-purpose request handler object is created if not existed, or retrieved if already there.
4	Invoked by step 3: Recover the current action data stored in the user’s application session
5	Invoked by step 3: The BuildCommadXML() method of the current editing action data is called to return XML string representation of an editing command.
6	Invoked by step 3: The ReceiveCommandMsg() method of the Pounamu server is called to receive XML format of an editing command from the thin-client user.
7	Invoked by step 3: XML representation of a remote editing command is analyzed by the Pounamu server
8	Invoked by step 7: XML representation of a remote editing command is interpreted and decoded by the RemoteCommandDeserializer object.
9	Invoked by step 8: The Pounamu-recognizable remote editing command object is created from the previous decoding process
10	Invoked by step 9: The user’s editing command finally gets executed.

4.3.4.6 Event Flows for "Identifying an Editing Target Shape/Connector"

Almost all editing actions are executed on a shape or connector, so after initiating a specific editing action, we have to decide which shape/connector is the working object. Figure 4-16 depicts event flows for the target identification process in a collaboration diagram, with a detailed description for each procedure within this process available in Table 4-9.

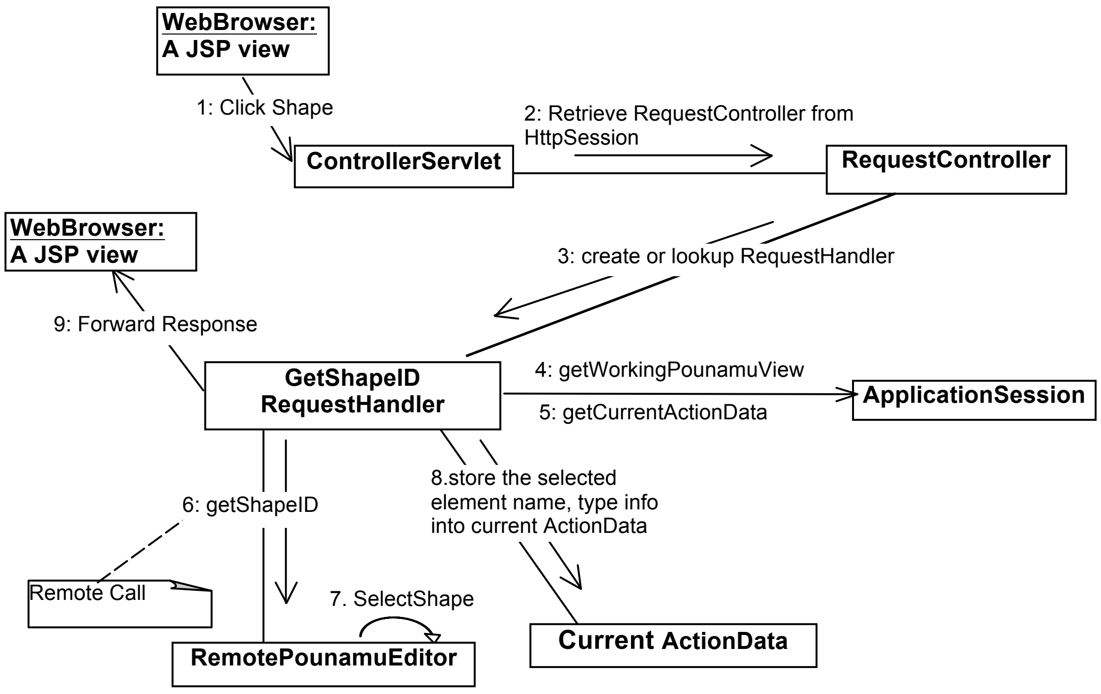


Figure 4-16 Collaboration Diagram for identifying a Target for an Editing Action

Table 4-9 Descriptions of events shown in Figure 4-16

Step	Description
1	Invoke by clicking an existing shape/connector in the pounamuview diagram displayed in a web browser.
2	Invoked by step 1: Retrieve the RequestController object from the service object HTTPSession. This RequestController is responsible for instantiating or looking up appropriate handler object to deal with the user's request sent via a web browser.
3	Invoked by step 2: A special-purpose request handler object is created if not existed, or retrieved if already there. Additionally, X,Y values are parsed from the user's clicking action.
4	Invoked by step 3: Recover the current Pounamuview name from the application session.
5	Invoked by step 3: Retrieve the action data currently active from the application session.
6	Invoked by step 3: The getShapeID() function of the Pounamu Server is called, and X,Y values saved previously are passed as parameters.
7	Invoked by step 3: A target shape or connector is highlighted with red rectangle handles.
8	Invoked by step 3: The name, type information of the identified shape or connector is stored into the current action data.
9	Invoked by step 3: A proper JSP view is selected to receive the system response to this shape identification request.

4.3.4.7 Event Flows for "Locating a Valid Shape Handle"

For some editing actions such as "Add Association" and "Resize Entity shape", a user should be able to locate a specific shape handle on the PounamuView diagram. To assist in understanding event flows in this locating operation, a collaboration diagram is shown in Figure 4-17. Additionally, Table 4-10 offers a description for each tangible step that has been marked in the figure.

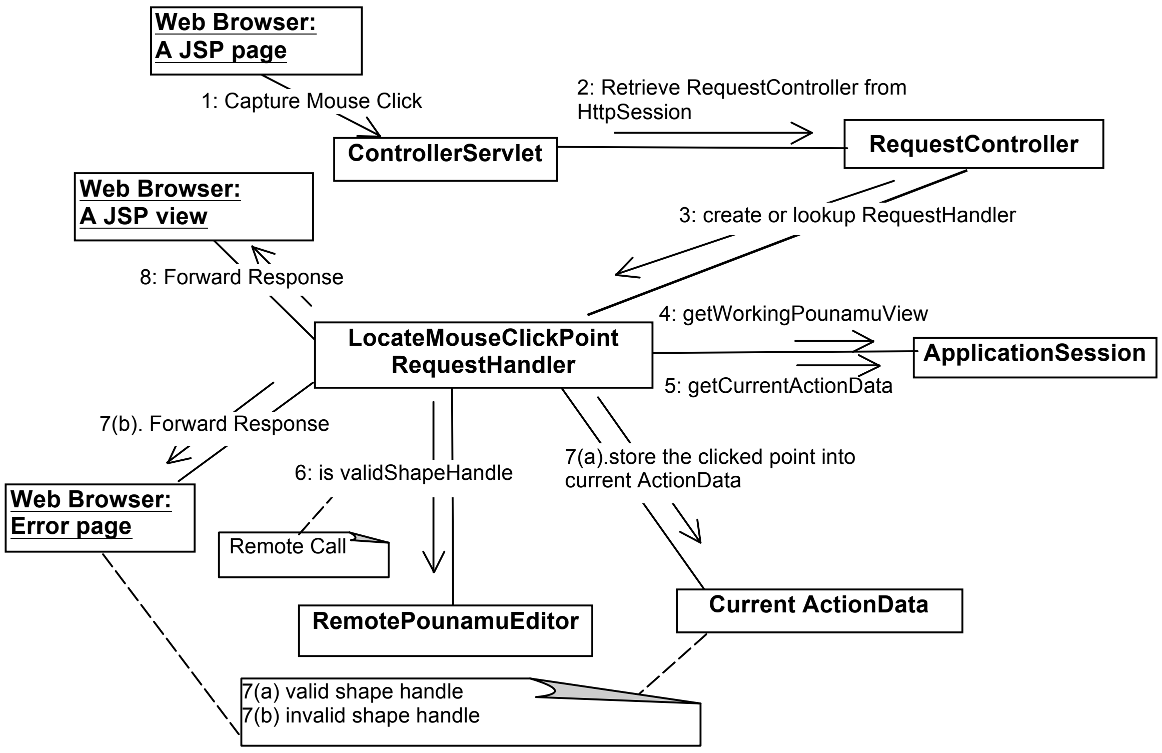


Figure 4-17 Collaboration Diagram for Specifying a Valid Shape Handle

Table 4-10 Descriptions of events shown in Figure 4-17

Step	Description	
1	Invoked when the user clicks the image-mapped PounamuView diagram embedded in a certain JSP page	
2	Invoked by step 1: Retrieve the RequestController object from HTTPSession. This object is in charge of instantiating or looking up appropriate handler object to handle the user's request sent via a web browser.	
3	Invoked by step 2: A special-purpose request handler object is created if not existed, or retrieved if already there. Additionally, X,Y values are parsed from the user's clicking action.	
4	Invoked by step 3: Recover the current Pounamuview name from the application session.	
5	Invoked by step 3: Retrieve the action data which is currently active from the application session.	
6	Invoked by step 3: The decideValidShapeHandle() method of the Pounamu server is called with X,Y values obtained previously passed as parameters.	
7	(a)	Invoked by step 3: Upon a valid shape handle, the clicking point is saved into the currently active editing data.
	(b)	Otherwise, forward the response to an error page with message "not a valid shape handle"
8	Invoked by step 3: A proper JSP view is selected to receive the system response to this locating shape handle request.	

4.3.4.8 Event flows for "Setting Element Properties"

Like all the other editing actions, several phases are involved in the property setting editing action:

a) Initiating a property changing action
b) Identifying a shape/connector for the action
c) Displaying old properties (including name, type, values) of the target object and entering new properties values
d) Confirming and submitting this action

To fully describe this setting properties action scenario, separate collaboration diagrams need to be drawn for each phase. However, as phase a), b) and d) are common scenarios to all the possible editing actions and have already been described, here only Phase c) is considered. The collaboration diagram depicting event flows for this phase is shown in Figure 4-18, with descriptions of these events provided in Table 4-11.

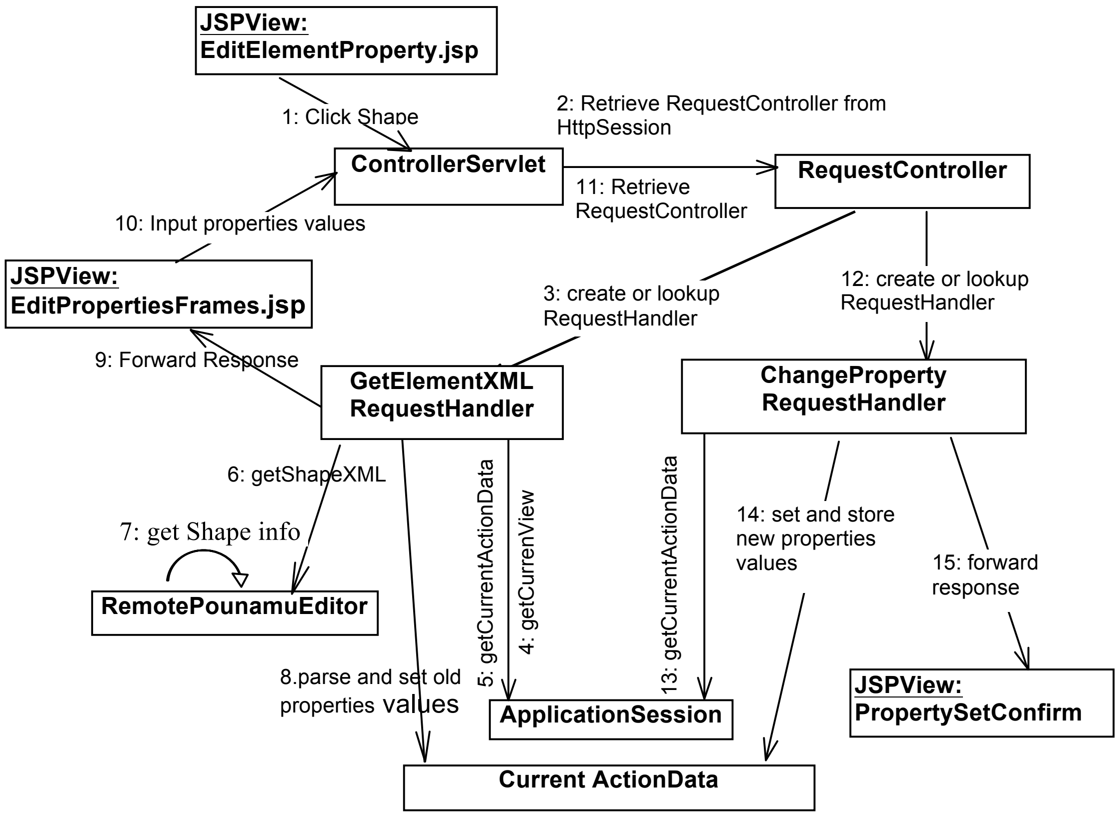

JSPView: EditElementProperty.jsp
1: Click Shape
2: Retrieve RequestController from HttpSession
ControllerServlet
RequestController
11: Retrieve RequestController
10: Input properties values
JSPView: EditPropertiesFrames.jsp
3: create or lookup RequestHandler
12: create or lookup RequestHandler
9: Forward Response
GetElementXML RequestHandler
ChangeProperty RequestHandler
6: getShapeXML
7: get Shape info
5: getCurrentActionData
4: getCurrenView
13: getCurrentActionData
14: set and store new properties values
15: forward response
RemotePounamuEditor
8.parse and set old properties values
ApplicationSession
JSPView: PropertySetConfirm
Current ActionData

Figure 4-18 Collaboration Diagram for Setting New Properties for the Target Element

Table 4-11 Descriptions of events shown in Figure 4-18

Step	Description
1	Invoked by clicking a shape within the image-mapped PounamuView diagram displayed on a web browser
2	Invoked by step 1: Retrieve the RequestController object from HTTPSession. This

	RequestController is responsible for instantiating or looking up an appropriate handler object to deal with the user's request sent via HTTP GET/POST.
3	Invoked by step 2: A special-purpose request handler object is created if not existed, or retrieved if already there. Additionally, X,Y values are parsed from the user's clicking action.
4	Invoked by step 3: Recover the current Pounamuview name from the application session.
5	Invoked by step 3: Retrieve the current editing action from the application session object.
6	Invoked by step 3: The getShapeXML() method of the Pounamu Server is called with X,Y valued obtained previously passed as parameters.
7	Invoked by step 3: Property values of the target element are packaged and sent back.
8	Invoked by step 3: Obtain the element name, type and properties values from the received XML string, and populate these values into the current editing action data.
9	Invoked by step 3: Select a proper JSP view to expose property values of the target element through the ApplicationSession bean.
10	Invoked by step 9: New property values are inputted via the JSP page displayed in step 9.
11	Invoked by step 10: Retrieve the RequestController object from HTTPSession. This RequestController is responsible for instantiating or looking up an appropriate handler object to handle the user's request sent via a web browser.
12	Invoked by step 11: A special-purpose request handler object is created if not existed, or retrieved if already there. Additionally, newly entered property values are parsed.
13	Invoked by step 12: Retrieve the current editing action from the application session object.
14	Invoked by step 12: Store the newly entered property values into the current editing data
15	Invoked by step 12: Forward response to the PropertySettingConfirmation.jsp page.

4.4 Summary

In this chapter, the design issues involved in the development of our Pounamu/Thin system have been discussed, and our discussion is mainly focused on three parts. First, we constructed and documented a web-based N-tier software architecture to form the backbone of the system. Then for each tier we proposed a set of OOD-level packages and classes to fulfill the required functionality. Lastly, we presented system dynamic behaviour via collaboration and sequence diagrams. In summary, the design presented in this chapter is a bridge between system analysis activities describing what a system should be and should do, and implementation activities that describe the language and platform-dependant manner in which the system is built.

Chapter 5: Implementation of the GIF-based Prototype

In this chapter, we describe implementation of the GIF-version prototype of the system based on the design described in the previous chapter. Included are explanations of how the different parts were implemented, code samples and screen dumps of the system running. Before discussing implementation details, computing technologies and tools relevant to the system implementation are summarized first. Then, the communication between the web tier and the remote Pounamu server, which is realized in RMI technology, is introduced. Following we describe the techniques used to dynamically display a PounamuView diagram on a web browser as a GIF image. In the section following that, we discuss our approach for interpreting and executing a user's editing action on the selected PounamuView. Finally, screen dumps and usage of this GIF version thin-client Pounamu tool are presented.

5.1 Aided Tools and Technologies

In this section, a brief introduction is given to tools and technologies involved in building the system, as well as what we do with them.

5.1.1 Tomcat Servlet/JSP Engine

Tomcat is a Servlet container used by a Web server to load and execute Java Servlets. It is the official reference implementation for the Java Servlet and JavaServer Pages (JSP) technologies [40]. Both Java Servlets and JSP pages are essential components of the Java 2 Platform, Enterprise Edition (J2EE).

Tomcat is an open source Servlet/JSP container that is part of Jakarta project [40]. It is free for most users. Although Tomcat can be used effectively with either Apache Web Server or a commercial web server in a production environment, in this project we use Tomcat's built-in web server for our development work. Compared with those commercial web servers such as Apache, Tomcat is less robust, but is sufficient for the purpose of prototyping our system. Additionally, Tomcat setup is comparatively easy and simple. The Tomcat version used here is Tomcat 4.1.24.

5.1.2 RMI and CORBA

RMI [42] and CORBA [43] are two of most commonly used distributed objects solutions, and are developed by SUN and OMG respectively. Both technologies have their strengths and weaknesses.

RMI allows Java developers to invoke object methods located on remote Java Virtual Machines. Unlike many remote procedure calls that require parameters to be primitive data types or structures composed of primitive types, in RMI, the entire java object can be passed as a parameter [44]. This exciting feature means that new code can be sent across a network and dynamically loaded by foreign virtual machines

[44]. As a result, developers have a greater freedom when designing distributed systems. Additionally, RMI, as a pure Java technology, can take full advantage of Java features such as object model, security, and garbage collections. In general, RMI has great potential to cover the range “from remote processing and load sharing of CPU’s to transport mechanisms for higher-level tasks”[44].

Although an advantage in some cases, working only with pure Java environments can also be a disadvantage for the RMI technology, because this signifies RMI’s lack of ability to deal with legacy systems written in C/C++, Fortran, Cobol, and other programming languages [44]. In contrast, CORBA, as a language- and platform-neutral technology, offers greater portability than RMI. Unlike RMI, CORBA is not tied to a particular language and can be integrated with legacy systems written in different languages as long as those languages have corresponding CORBA implementations [44]. From this point of view, CORBA is particularly important in large organizations, where many systems (probably legacy systems) must interact with each other. However, one of issues with CORBA is that being language neutral, some powerful features are lost [29]. One obvious example is that instead of being able to pass back and forth the actual objects as parameters, CORBA only allows primitive data types, and structures to be sent across network. Another major limitation with CORBA is that users must learn to use an interface definition language (IDL) for describing services.

From the above examination of two technologies, we can see both RMI and CORBA possess strengths and weaknesses, so users have to decide which technology is the most appropriate for their applications. With respect to our prototype system, RMI has been chosen as the communication protocol between the Web Server and the remote Pounamu Server. The fact that the existing Pounamu is written purely in Java is the main reason for this choice. Other tangible reasons include the ease use of RMI for experienced Java developers, and low cost. However, In the future, the system still needs to be extended to use CORBA, which is widely supported by vendors and therefore more mature than RMI.

5.1.3 Servlets and JSPs

Servlets are platform-independent, pure Java server-side modules that fit seamlessly into a Web server framework. They can be used to extend the capabilities of the Web Server with minimal overhead, maintenance, and support. Unlike other scripting languages, Servlets involve no platform-specific consideration or modifications, they are Java application components that are downloaded, on demand, to the part of the system that needs them [41]. Today, Servlets have become a popular choice for building interactive Web applications. Third-party Servlet containers are available for Tomcat, Apache Web Server, Microsoft IIS, and others [41].

Servlets are particularly useful when there is a lot of real programming and computing in your web application. For example, Servlets are normally used to store information between requests, track HTTP sessions, handle HTTP status codes, use cookies, access databases, generate dynamic GIF images, as well as perform many other computing intensive tasks [47]. However, outputting HTML with Servlets can be boring because every trivial HTML tags and text need to be printed out with code [47]. That's where JavaServer Page (JSP) comes to help. JSP is in fact an extension of the Servlet technology that allows you to mix standard and static HTML tags with dynamically generated HTML. By separating much of the presentation from the dynamic content, you can still write HTML in the normal way, and then insert simple java code into JSP in the form of JSP expressions, scriptlets, and declarations to generate dynamic content [47]. For even more complex applications, you can wrap up Java code inside beans or define your own JSP tags.

JSP with Beans and custom tags, although extremely powerful and flexible, are not suitable for applications in which request parameters are difficult to represent as HTML fields, as is the case with our system. For these sorts of applications, a proper solution would be the use of both JSP and Servlets, that is, a Servlet is responsible for handling the initial request, partially process the data, set up beans, then forward the results to one of JSP views, depending on the circumstance. When used together in this way, JSP and Servlets can complement each other to achieve the system's ultimate goals such as ease of use, enhanced performance, separation of logic from display, extensibility into the enterprise, and ease of maintenance [46].

5.1.4. XML

Extensible Markup Language (XML) is a markup language for the description of documents that contains structured information. Unlike HTML in which both the tag semantics and tag set have been standardized, XML defines neither semantics nor a tag set. XML is really a meta-language for describing markup languages and provides a facility to define tags and the structural relationships between them [45].

Derived from another markup language, namely SGML (Standard Generalized Markup Language [48]), XML was initially designed to address the challenges of large-scale electronic publishing. Today, XML is playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere [48]. Generally speaking, XML has developed into the new standard for data exchange, publishing, and developing intelligent web agents, etc. [49].

Compared with other data exchange methods, XML provides the following advantages [50]. First, due to its platform- and system-independent feature, data in XML format is readable and usable on any computer. Additionally, by being customizable, XML implementers can define their own tag sets to describe document content, and any XML-aware software will be able to work with these customizable tags. When dealing with XML, different clients can ask to have the same document displayed in customized ways with the help of associated style sheets [50]. A final advantage is that XML has adopted the Unicode standard, which allows documents to be coded in any language [50].

All of the above advantages add weight to the decision to select XML as the message exchange format between the web component and the Pounamu Server, though, there are other good reasons for doing so. The fact that Pounamu is an XML-based Meta-CASE tool is certainly one such reason. Additionally, the possibility of further expanding our work into web services that consume XML-like SOAP messages also contributes to our decision.

5.1.5 Web Graphics Format

To enable users' access to the Pounamu tool via a web browser, the first question that we need to answer is what web graphics format a PounamuView diagram should be displayed in. In order to get a satisfying solution, two most commonly used web graphics formats, GIF and JPEG, are investigated.

The GIF graphic format was developed by CompuServe to optimize the transmission of image data over networks [52]. To keep file sizes small, the designers limited the number of colors in a GIF image to 256, and this restricts its ability to handle the almost infinite color range found in most photographs [51]. In comparison, JPEG supports full-color (24-bit, "true color") images and therefore is often chosen for photographs, as well as complex "photographic" illustrations [51].

Despite GIF's limitation mentioned above, it's still favored by web developers for most page design elements. The reasons for this preference are generalized as follows:

- ♦ GIF is still the most widely supported graphics format on the Web. And almost all Web browsers support it.
- ♦ JPEG doesn't work well for graphic-based imagery (such as line drawing, illustrations and cartoons etc.) because its lossiness is clearly noticeable (and annoying) in a diagram of this sort. In comparison, GIFs of diagrammatic images look better.
- ♦ Due to its specific compression format, GIFs can be interlaced. Interlaced GIFs appear first with poor resolution and then improve in resolution until the entire image has arrived, allowing the viewer to get a quick idea of what the picture will look like while waiting for the rest [51]. In comparison, JPEGs can only arrive linearly, from the top row to the bottom row [51].

The above discussion shows GIF is much more suitable for diagram representation, so in our project, GIF is selected as the web graphics format.

5.2 Communication between the Web tier and the Pounamu Server

As mentioned before, because of its simplicity and our pure Java environment, RMI has been chosen as the communication protocol between the Web-tier and the Pounamu Server. Thus from the system structure's point of view, the Web-tier not only acts as the web server to be responsible for processing a user's requests and responding correspondingly, it also takes the role of an RMI client to make remote procedure calls. To be able to do this, an object reference to the RMI Pounamu Server should be available in our web-tier. When deciding where to lookup and store this RMI server reference, we have to consider the following factors:

- Our web-tier adopts the request-controller architecture;
- The RMI server object reference is an application-scope resource as it should be available for all JSPs/Servlets within the web application, and shared by all user sessions;

The first factor says a controller Servlet exists as an entry point for any user request in the web application. Actually, every Servlet contains an init() method which has two features:

- It is called only once in the Servlet's lifecycle;
- It has access to the web application's ServletContext object (an object which holds application-scope objects).

So it is easy to derive that the init() method of the controller Servlet would be an appropriate place to open and lookup any shared resource, such as an RMI remote server reference, JDBC connection, and application-level property files. The following code shows the remote object's lookup and store:

```
public class ControllerServlet extends HttpServlet {

  /**
  * Lookup shared remote RMI server in this init() method
  */
   public void init(ServletConfig config) throws ServletException {
     ServletContext context=config.getServletContext();
     RMIServerInterface remotePounamu=null;

     //lookup and store remote RMI Server
      try {
        remotePounamu=(RMIServerInterface)Naming.lookup("RMIPounamuServer");
       context.setAttribute("RMIRemoteServer", remotePounamu);
      }catch (Exception e) {
          e.printStackTrace();
      }
   }
    /* Rest of ControllerServlet code*/

 }
```

Whenever this server object reference is needed (i.e. remote RMI called are made), it can be retrieved from the ServletContext by calling:

```
RMIServerInterface obj = (RMIServerInterface) context.getAttribute("RMIPounamuServer");
```

5.3 Keeping and Tracking Session State Information

As with a typical e-commerce web application which needs a session object to keep the state of a user's shopping cart, our application also has to keep track of session state information. This is due to the following application-specific requirements:

(1) Most times, users continue editing a certain PounamuView diagram for a while. To avoid loading the same working view diagram for each individual editing action, names of working view and project and other useful data need to be kept in the session object for a period.
(2) Almost all the editing scenarios ("Remove Shape", "Change Element Property" etc.) need a series of associated requests or interactions with the web server. This means each interaction has to be stored for the final execution of the editing action.

Typically, there are three well-used solutions for maintaining sessions over the web, namely HTTP Cookies, URL Rewriting and Hidden form fields [53]. Although all of the above are excellent alternatives for session tracking, they have their drawbacks. For the first and second approaches, a common problem is that the server-side program has to do much straightforward but tedious processing, while for the third, the main disadvantage is that a HTML form with an entry looking like *<INPUT TYPE = "HIDDEN" NAME="session" VALUE="...">* is always needed.

Fortunately, Servlets provide a better technical solution, i.e., using the HttpSession API for session tracking [53]. Compared with the previous approaches, using HttpSession in Servlets is very straightforward, and a list of tasks involved is as follows:

a) Looking up the session object associated with the user's request,
b) Creating a new session object if it doesn't exist,
c) Looking up information associated with a session,
d) Storing information about a session,
e) Abandoning completed sessions.

For the reasons mentioned above, in this system, we depend on the HttpSession object for session tracking, and the detailed implementation can be generalized as the following:

- Define a data model class ("ApplicationSession") that represents the state of a user session;
- Look up the HttpSession object associated with a request by calling: *HttpSession session = request.getSession(true);*
- Associate the data model class with a session using *session.setAttribute()* method;
- Access session state through *session.getAttribute()* method.

The purpose of defining the *ApplicationSession* class is to avoid the messy and unnecessary operation of setting individual state attributes into a session. The code sample below shows the state attributes contained in this class:

```
public class ApplicationSession implements Serializable, HttpSessionBindingListener {
  /**
   * Individual session state data stored in this class
  */
   private String workingTool=null;
  private String workingProjectName=null;
   private String workingPounamuViewName=null;
   private String currentViewType=null;
   private Notifier sessionClientCallback=null;   //RMI client call back for events notification
   private ActionData currentActionData=null;     //current editing action data
   private byte[] imageByteArray=null;    //current in-memory image cache
   ServletContext context=null;   //variable used to retrieve remote RMI Server reference object

  /*Other data attributes and class methods*/
}
```

Here, two special attributes, namely *sessionClientCallback* and *currentActionData*, are worth special notice. The first one is a *Notifier* type instance object, and its role is to register itself to the RMI Pounamu server and receive events notification from the server. Sometimes this sort of notification mechanism would make web applications more powerful and efficient because clients can immediately know when and by whom the shared information or data has been changed. An example of using this *sessionClientCallback* for events notification is explained in next section "**Displaying a PounamuView Diagram on a Web browser**". The second attribute *currentActionData* has an interface type *ActionData*. It can refer to instance of one of the 7 subclasses such as *AddEntityActionData*, *EditPropertyActionData,* etc. Whenever users initiate an editing action by clicking the corresponding menu item, the action data of a certain type will be instantiated and assigned to *currentActionData*. Details of how to use this object are further discussed in the "**Interpreting and Execution of Editing Actions**" section.

5.4 Displaying a GIF-version of PounamuView Diagram on a Web browser

Upon a user's request to load a working PounamuView or refresh an updated one, the GIF-format image of the requested PounamuView diagram needs to be displayed on the web browser. At first sight, it is a simple dialogue between the browser and the web server. However, quite a lot processing happens underneath, which can be explained as answers to the following questions:

- Where would we want this GIF image to be generated? (The Pounamu Server or the Web server)
- If the generation takes place on the Pounamu server, how to send this binary data across RMI?
- How can we dynamically display the newly generated GIF image on the web browser?

The discussion below revolves around these questions.

To provide a good solution to the first question, three factors need to be considered:

- All the data required to generate the GIF version of a PounamuView diagram are stored on the Pounamu Server.
- A PounamuView diagram drawn with Java AWT and Swing components already exists on the Pounamu Server.
- Several useful classes in the Java API can be used to draw Swing components into an image buffer.

All of above factors indicate generating GIF images on the Pounamu server would be a better choice. With the help of the Java utility classes contained in packages such as javax.imageio.*, java.awt.*, and java.awt.image.*, the work for redrawing a canvas which contains a PounamuView diagram into a BufferedImage is not a difficult task.

Our answer to the first question brings the second one to the front. As stated in Java RMI reference [54], only classes that implement “Java.io.Serializable” can be sent across RMI. Unfortunately, BufferedImage is not one of them. So to send the generated buffered image across RMI, a helper class ImageFile with “Java.io.Serializable” as one of its parent classes has been defined, and the main attributes and methods of this class are shown below:

```
public class ImageFile implements java.io.Serializable{

/**
* The main attributes
*/
  private int[] m_data=new int[]{0};
  private int width;
  private int height;
  private int imageType=BufferedImage.TYPE_INT_RGB;
  private String sampleModelType="";
  private int[] bandOffsets=new int[]{0};
  private int[] bitmasks=new int[]{0};
  private int scanline=0;
  private int pixelStride=0;

  /**
  *Setter and getter methods for above attributes
  */
}
```

When the web tier receives an ImageFile instance object through RMI, it can recompose this object into a Java BufferedImage with another helper class BufferImageBuilder shown as the following:

```
public class BufferImageBuilder {
  ImageFile image=null;

  /*Constructor*/
  public BufferImageBuilder(ImageFile image) {
    this.image=image;
  }
  /**
  * Method to reconstruct BufferedImage from an ImageFile object
  */
  public BufferedImage constructBufferedImage()throws IOException{
    /*function body omitted here*/
  }
}
```

Having obtained satisfactory answers to the first two questions, we now move to the third one, “how to display the generated GIF image on the fly”. It is also a key question that needs to be addressed given the following application-specific features:

- Different users may access different PounamuView diagrams at the same time;

- Even a single user may frequently change the working PounamuView diagram.

Based on the resources in [55,56,57], the process of generating and outputting dynamic binary content from a Java Servlet is fairly simple. Generally speaking, there are two ways to do so (here we assume Java Servlet has already obtained a newly generated *BufferedImage*):

1. **Write the *BufferedImage* to a file on disk and provide a link to it.**

You do this by writing the *BufferedImage* to a temporary image file in the web server directory tree. Of course, in order to turn this *BufferedImage* into an image file or a bytestream, you may use Acme Labs' *GIFEncoder* class (for GIF) or the Java 2 *JPEGCodec* class (for JPEG). The image file can be downloaded to the web browser by embedding the HTML tag *"<IMG SRC='tempfile.gif'>"* in the requested JSP file.

2. **Output the image directly from the Servlet.**

This is done by setting the content-type header to image/gif (for GIFs), or image/jpeg (for JPEGs). Then, you open the HttpResponse output stream as a raw stream, not as a PrintStream, and send the bytes directly down this stream using the write() method. This image outputting Servlet is referenced by embedding something like "*<IMG SRC='/servlets/controllerServlet?action=generateGif'>"* in the JSP file.

The pros and cons of the above two methods are summarized as follows:

Method 1:

Pro: Images can be cached by the browser, and successive requests don't need to execute the Servlet again, thus reducing the server load.

Con: As image files will never be deleted from your disk, a script is needed to periodically clean out the image directory.

Method 2:

Pro: Compared with the first method, displaying the same binary content takes less time.

Con: Not caching image files means that every request needs to generate and output the image again.

By analyzing the above pros and cons, it can be seen that the first method is more suitable for those circumstances in which images may change less frequently, while the second method would be better for situations in which there is a high variation rate in the displayed images. Our application fit into the second category. However, our application still needs some caching capabilities not provided by the second method. The reason can be seen from the following GIF image generation and displaying process:

- Whenever a user make requests for JSP pages which embed the Servlet to output a GIF image of a PounamuView diagram, part of each request will be transferred to and handled by the *GenerateGif* request handler;
- Since the *BufferedImage* of the working PounamuView is generated on the Pounamu server, the *GenerateGif* request handler has to make a remote call to ask for generation and return of the corresponding *BufferedImage*;

- This returned *BufferedImage* is encoded by the appropriate *GIFEncoder* and outputted directly by the Servlet.

Assuming no caching is allowed on the web server, any request of the working PounamuView diagram always needs to go through the above three steps even if the diagram has undergone no change since the previous visit and display. Obviously, the servers' (the Pounamu Server and Web Server) working loads are unnecessarily increased. To avoid over-loading servers, we have designed our own cache (in-memory data array variable "*imageByteArray*" in *ApplicationSession*) where the last requested diagram can be kept. A dirty flag bit that indicates whether this diagram needs changes or not is set whenever any user has made edits on it. The notification that this diagram has been changed since the user's last visit can be implemented with the RMI client Callback mechanism mentioned earlier.

The only important method in the GenerateGif request handler class is shown as the following:

```
public class GenerateGif extends RequestHandler {

 /**
  *A method used to handle requests for displaying GIF image of a PounamuView diagram
  *on the web browser
 */
 public void handleRequest(ServletContext context, HttpServletRequest req,
                                HttpServletResponse resp) throws ApplicationException {
  ApplicationSession appSession = RequestController.FindApplicationBean(req, this);
  resp.setContentType("image/gif");   //set response content type
  try{
   ServletOutputStream out =resp.getOutputStream();
   ByteArrayOutputStream out1=new ByteArrayOutputStream();

   //pulling to see if the PounamuView diagram  is updated
   if (!appSession.getViewUpdated()){
     //get the image byte array from the memory cache and send it to the browser
      byte[] gif=appSession.getImageByteArray();
      out.write(gif,0,gif.length);
   }
   else{ //should call the remote function to generate the updated image
     RMIServerInterface obj = (RMIServerInterface) context.getAttribute("RMIRemoteServer");
     ImageFile image = obj.generateImage(appSession.getCurrentProject(),
                    appSession.getCurrentViewType() appSession.getCurrentPounamuView(),
                    appSession.getOpened(), null);
     if (image != null) {
        BufferImageBuilder imageBuilder = new BufferImageBuilder(image);
       BufferedImage bi = imageBuilder.constructBufferedImage();

        // Encode the off screen image into a GIF and send it to the client
        GifEncoder encoder = new GifEncoder(bi, out);
        encoder.encode();
        GifEncoder encoder1 = new GifEncoder(bi, out1);
        encoder1.encode();
        //put out1 to a cache array
        appSession.setImageByteArray(out1.toByteArray());
         //send out to browser
        out.flush();
        out.close();
       }
     }
   }catch(RemoteException e){
     throw new ApplicationException(e.toString());
   }catch(IOException e){
     throw new ApplicationException(e.toString());
   }
 }
}
```

5.5 Interpreting and Execution of Editing Actions

The purpose of this project is to design a thin-client user interface for our existing Pounamu tool so that it can be run on a web browser with no special software plug-ins installed. As we already know, the web browser itself provides very limited user interaction facilities through HTML form elements, and the small degree of interaction it permits is far below the requirement for diagram editing scenarios, where users at least should be equipped with some means of interacting with the PounamuView diagram displayed on a browser. The solution that we propose to allow such interactions is to embed an assumed image map within HTML. Here we use words “assumed image map” to differentiate from real image maps. Before discussing how to use an assumed image map in our application, it is necessary to briefly introduce image maps first. Several steps involved in using server-side image maps are as follows [58]:

- First, define an ASCII-text image map file (with file extension .map).

Below is an example of a image map file where each line contains a rectangular link and the X, Y coordinates that define the upper left corner and lower right corner of the link area:

default http://www.projectcool.com

rect http://www.devx.com/projectcool/sightings 141,114 372,143

rect http://www.devx.com/projectcool/coolest 141,154 372,183

rect http://www.devx.com/projectcool/developer 141,194 372,223

rect http://www.devx.com/projectcool/focus 141,234 372,263

- Second, Store this file in a known place of web server.
- Last, build an HTML page and incorporate this map by embedding special HTML tags into it.

An example of using an image map in an HTML file is shown below:

<A href="/mapservlet/main.map">

<IMG SRC="images/myImage.gif" width=411 height=399 ISMAP></A>

where *<A>* element’s *HREF* identifies the name of image map file and a Servlet program used to decode the image map, *<IMG>* identifies the image which the links are mapped onto. In addition, *<IMG>* element must contain the attribute *ISMAP* so the browser can decode the image appropriately.

Instead of mapping X, Y coordinates of a GIF image to special links, an assumed image map in this application is only used to capture X, Y coordinates of a mouse click. So no image map file needs to be created and referred in the anchor reference tag. Actually, in the implementation, we have used HTML tags that look like:

< A href= "/controllerservlet?action=locateMousePosition">

<IMG SRC= "/controllerservlet?action=generateGif" ISMAP"> </A>

Then, whenever a user clicks somewhere in the image, the *locateMousePosition* request handler class helps to obtain X, Y values of the point being clicked.

Knowing how to interact with the system, now we will use an example to illustrate what actually happens when a user initiates an edit command. Figure 5-1 shows a sequence of actions involved in the "moving an entity shape" editing scenario, and this is followed by a brief description for each action.

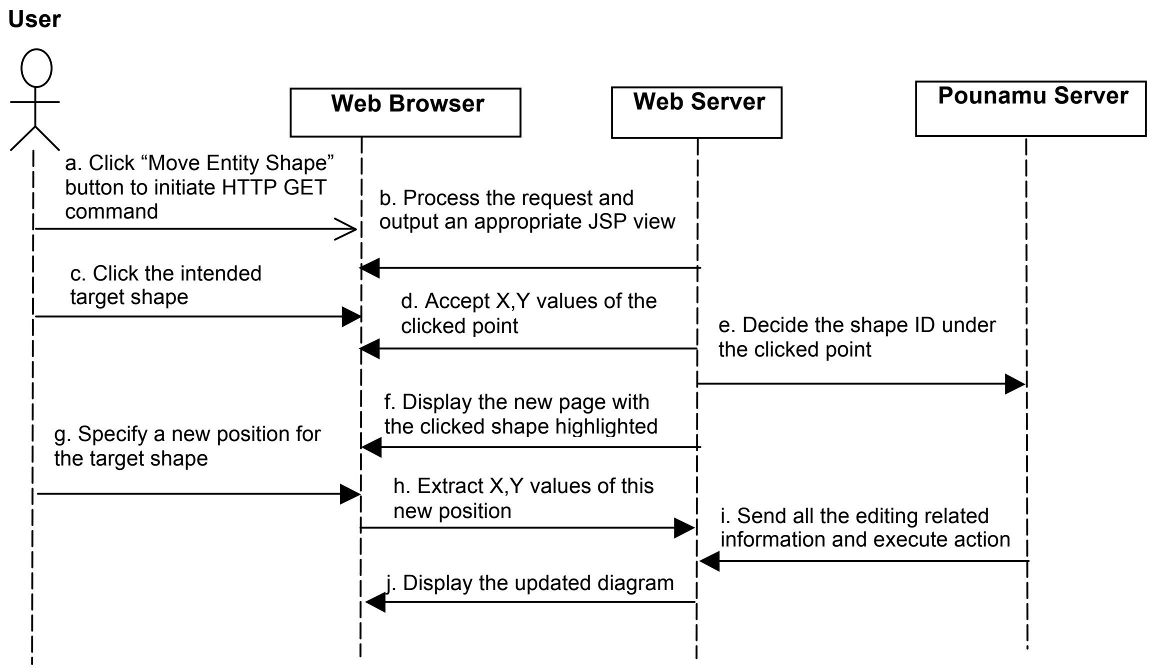

User
Web Browser
Web Server
Pounamu Server
a. Click "Move Entity Shape" button to initiate HTTP GET command
b. Process the request and output an appropriate JSP view
c. Click the intended target shape
d. Accept X,Y values of the clicked point
e. Decide the shape ID under the clicked point
f. Display the new page with the clicked shape highlighted
g. Specify a new position for the target shape
h. Extract X,Y values of this new position
i. Send all the editing related information and execute action
j. Display the updated diagram

Figure 5-1 Dialogues Communicated in Moving Entity Editing Action

a. A user clicks "Move Entity" button to enter editable state;
b. The corresponding JSP view with the editable PounamuView diagram appears;
c. In this JSP page, the user can click a model entity shape that needs to be moved;
d. The web server accept the clicked point and extract the corresponding X,Y values;
e. The X, Y values of the click point are sent back to the Pounamu server through a remote RMI call, and it's there that the shape ID is determined and returned;
f. The working view diagram is re-displayed with the selected shape highlighted;
g. Then the user needs to specify a new location for the selected model element;
h. Again, the web server has to extract the X,Y values of the clicked new position;
i. Any information related with this move action (including element ID, old location and new location) will be sent to the Pounamu Server, and there this "move entity" edit command is executed;

j. The updated PounamuView diagram is redisplayed on the web browser;

By analyzing the above steps, we see that a class is required to collect all the information needed to perform this "move entity shape" command, including the name of PounamuProject, the name of PounamuView, modelElementID, modelElementType, old location and new location of this shape, etc. Since most of these information is entered by interacting with the displayed GIF image, the traditional JavaBean method is not suitable for this task. For this reason, we have designed an interface (namely *ActionData*), together with 7 specific classes that implement this interface. Each subclass acts as information container for one type of edit command. In addition, the fact that action-specific information is collected through a series of consecutive interactions means the current *ActionData* should be a Session-scope object. So as mentioned in section 5.3, a reference to the current *ActionData* is contained in the *ApplicationSession* class.

Step(i) of the above "moving a entity shape" process also reveals another implementation detail, i.e., edit command-related information that has been collected is sent to the Pounamu server and executed there. To do this, a very important method, *buildCommandXML(),* is provided in interface *ActionData*. Each of the 7 subclasses has its own implementation of this method, depending on the data collected and required to execute the corresponding edit command. As an example, *buildCommandXML()* of class *MoveShapeActionData* is listed as below:

```
public class MoveShapeActionData implements ActionData {

 /**
  *Data attributes that will be set and used for XML command string
 */
 static final String commandName="MoveShape";
 String currentProject=null;
 String currentView=null;
 String currentViewType=null;
 String modelCompName=null;
 String compType=null;
 Point oldLocation=null;
 Point newLocation=null;

 /**
  *Build XML editing action string from data attributes set during the previous interactions
 */
 public String BuildCommandXML(){
   String commandXML=null;
   StringBuffer buf = new StringBuffer (400000);
   String space = "     ";
   buf.append("<?xml version=\"1.0\" standalone=\"yes\"?>");
   buf.append("<command>\n");
   buf.append(space+"<commandname>");
   buf.append(commandName);
   buf.append("</commandname>\n");
   buf.append(space+"<modelProjectName>");
   buf.append(this.currentProject);
   buf.append("</modelProjectName>\n");
   buf.append(space+"<viewTypeName>");
   buf.append(this.currentViewType);
   buf.append("</viewTypeName>\n");
```

```
    buf.append(space+"<viewName>");
    buf.append(this.currentView);
    buf.append ("</viewName>\n");
    buf.append(space+"<modelelement>\n");
    buf.append (space+space+"<name>");
    buf.append(this.modelCompName);
    buf.append("</name>\n");
    buf.append (space+space+"<type>");
    buf.append(this.compType);
    buf.append("</type>\n");
    buf.append(space+space+"<newlocation>\n");
    buf.append(space+space+space+"<XPos>");
    buf.append(this.newLocation.x);
    buf.append("</XPos>\n");
    buf.append(space+space+space+"<YPos>");
    buf.append(this.newLocation.y);
    buf.append("</YPos>\n");
    buf.append(space+space+"</newlocation>\n");
    buf.append(space+"</modelelement>\n");
    buf.append("</command>\n");
    commandXML=buf.toString();
    return commandXML;
  }

  /*Rest of code*/
}
```

There were several reasons behind the decision to build XML-message like edit commands rather than making direct remote editing procedure calls. For example, the fact that XML is platform and language independent was of importance. Another obvious reason would be data type and quantity that can be carried in XML is very flexible, and it even can contain binary data. In addition, using XML eases the subsequent implementation of other remote message passing technologies, for example Simple Object Access Protocol (SOAP) [30], which is an XML based technology.

Now we review what the Pounamu application server does after receiving an XML-format edit command from the web tier. In the design phase introduced in Chapter 4, interface *RemotePounamuCommand* and its 8 subclasses have been designed to commit specific edit commands on the working PounamuView diagram. For example, the execution of *execute()* method of *RemoteAddEntityCommand* class adds a new entity model element to the PounamuView diagram. So, one way to get XML like edit command messages from the web tier executed is to convert them into the corresponding *RemotePounamuCommand* objects first.

An interface, *RemotePounamuCommandDeserializer*, is used as API for the implementation of parsing XML-format editing commands into the corresponding remote Pounamu command objects. This interface contains one method named *extractCommandFromXML()*, which takes the received XML string and returns a *RemotePounamuCommand* type object. The code below shows this interface:

```
public interface RemoteCommandDeserializer {
  public RemotePounamuCommand extractCommandFromXML(Node commandXML);
}
```

There are 8 specific command deserializer classes that implement this *RemotePounamuCommand-Deserializer* interface, one for each possible remote command object in Pounamu. All these deserializer classes have their own implementation of the *extractCommandFromXML()* method. Below is an example of a deserializer class, namely the *RMSDeserialier* (the deserializer class for the *RemoteMoveShape command*), and its implementation of *extractCommandFromXML()* method:

```
public class RMSDeserializer implements RemoteCommandDeserializer{
  public RemotePounamuCommand extractCommandFromXML(Node root){
    RemoteMoveShape remoteMSCommand=null;
    NodeList nl=null;    Node n=null;
    nl=((Element)root).getElementsByTagName("modelProjectName");
    n=nl.item(0);
    String projectName=n.getFirstChild().getNodeValue();
    nl=((Element)root).getElementsByTagName("viewTypeName");
    n=nl.item(0);
    String viewType=n.getFirstChild().getNodeValue();
    nl=((Element)root).getElementsByTagName("viewName");
    n=nl.item(0);
    String viewName=n.getFirstChild().getNodeValue();
    nl=((Element)root).getElementsByTagName("modelelement");
    n=nl.item(0);
    NodeList nnll=((Element)n).getElementsByTagName("name");
    Node k=nnll.item(0);
    String modelElementName=k.getFirstChild().getNodeValue();
    nnll=((Element)n).getElementsByTagName("type");
    k=nnll.item(0);
    String modelElementType=k.getFirstChild().getNodeValue();
    nnll=((Element)n).getElementsByTagName("newlocation");
    k=nnll.item(0);
    NodeList nnnl=((Element)k).getElementsByTagName("XPos");
    Node p=nnnl.item(0);
    int xPos=Integer.parseInt(p.getFirstChild().getNodeValue().trim());
    nnnl=((Element)k).getElementsByTagName("YPos");
    p=nnnl.item(0);
    int yPos=Integer.parseInt(p.getFirstChild().getNodeValue().trim());
    PounamuModelElement pme=null;
    PounamuModelProject modelProject = (PounamuModelProject) manager.getOpenedModels().get(projectName);
    Hashtable openModelViews = (Hashtable) modelProject.getOpenedModelViews();
    Hashtable hash = (Hashtable) openModelViews.get(viewType);
    PounamuView view = (PounamuView) hash.get(viewName);
    ModellerPanel mp = (ModellerPanel) view.getDisplayPanel();
    PounamuPanel relocatePanel=null;
    int oldX=0;int oldY=0;
    Vector shapes=mp.getShapes();
    for(int i=0;i<shapes.size();i++){
      PounamuPanel panel=(PounamuPanel)shapes.get(i);
      PounamuShape shape=panel.getPounamuShape();
      If(shape.getName().equals(modelElementName)){
        relocatePanel=panel;
        break;
      }
    }
    if(relocatePanel!=null){
      oldX=relocatePanel.getBounds().x;
      oldY=relocatePanel.getBounds().y;
      remoteMSCommand=new RemoteMoveShape(mp, relocatePanel, oldX, oldY, xPos, yPos);
    }
    return remoteMSCommand;
  }
  /*Rest of code*/
}
```

5.6 Concurrent Editing Issues

Almost all software designers in industry know that the early phase of system development often requires collaboration and review of designs by a group of members and by other stakeholders. This review often takes place in different locations, and sometimes informally [12]. Past experience tells us that a “heavy-weight” infrastructure is required to enable collaboration support in standalone software engineering tools. However, even with such an infrastructure, collaboration is still restricted to those people who have special software installed on their machines. In comparison, another cheaper and less-effort way to implement a collaboration mechanism is to make use of the web medium. Our Pounamu/Thin tool is in this second class. Below we will address implementation details that allow multiple users to concurrently edit the same PounamuView diagram.

The first thing that we should point out is that a multithreaded *ControllerServlet* has been used in the web tier. In order to make it thread-safe, we have avoided using instance and static variables in this Servlet. Actually, the only instance variable that was defined in *ControllerServlet* is an object of *RMIServerInterface* type. But, as this object variable is defined inside the Servlet’s *init()* (which is executed only once) and never changed by any client request, the code referring this variable needs no protection.

We now move to the remote Pounamu server tier. This is where edit commands from the web tier are actually executed, and therefore the source code in this tier needs to be carefully designed to be thread-safe. As mentioned in the previous “**Interpreting and Execution of Editing Actions”** section, when all the information required to perform a edit command has been collected in the web tier, the *receiveCommandMsg* method of *RMIPounamuServer* is called to decode and execute the remote editing command. So an easy approach of enforcing thread-safe code is to synchronize this method, and the code sample showing how to do this is displayed below:

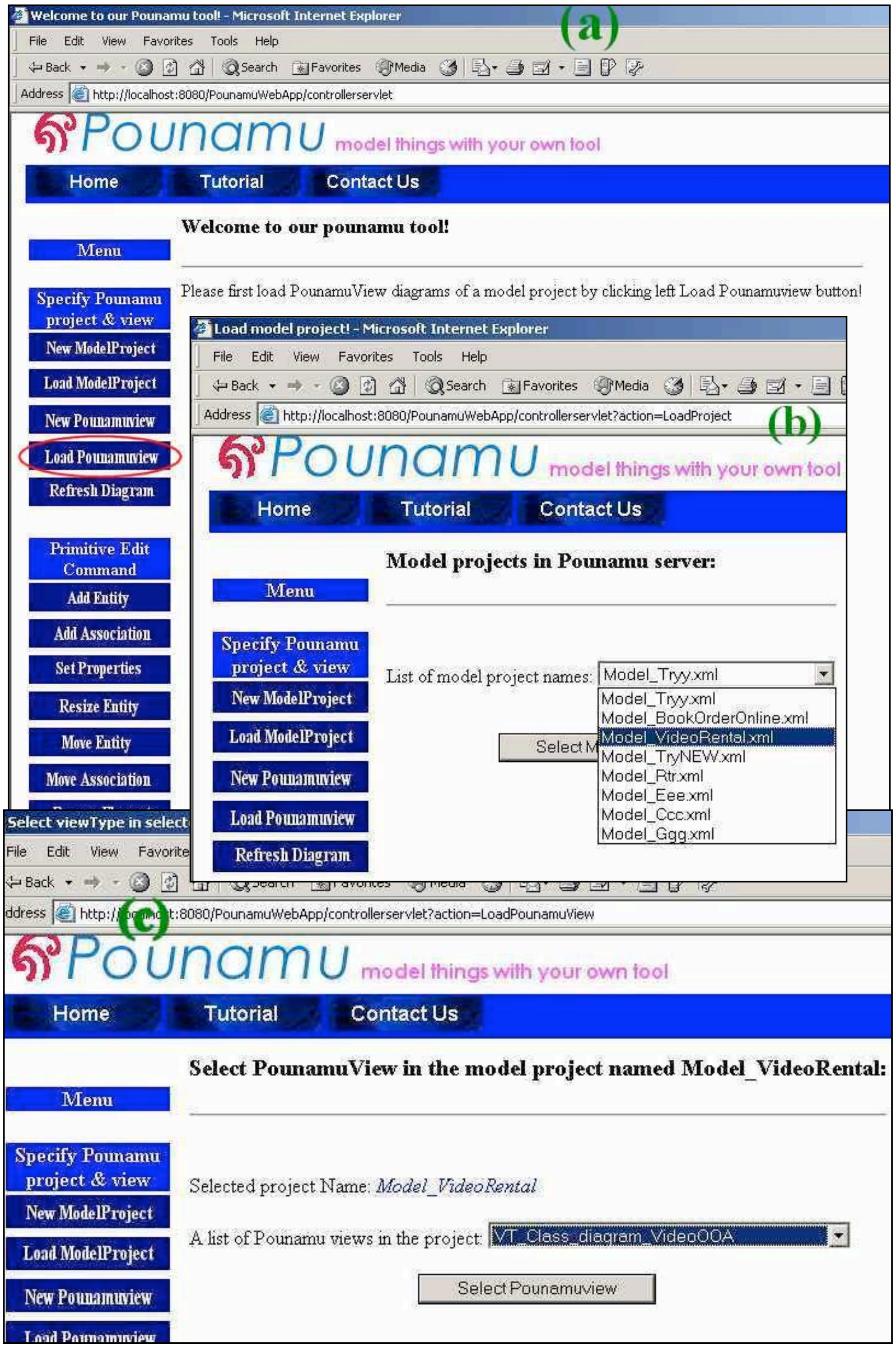
```
public class RMIPounamuServer extends UnicastRemoteObject implements RMIServerInterface{

 public synchronized boolean receiveCommandMsg(String viewName,String commandXML){
   boolean commandExecuted=remoteOperator.receiveCommandMsg(commandXML);
   if(commandExecuted)
      notifyAllClients(viewName);
  return commandExecuted;
 }
 /*Rest of code*/
}
```

5.7 Screen Dumps of System Running

In this section, screen dumps are used to depict all the main functionalities that this prototype provides, including Adding Entity shape, Adding Association connector, Moving shape, Setting properties, Resizing shape and Removing Shape/Connector etc. For each of the editing operations, an example is given to show the basic user interfaces of the system, as well as users' interaction with it. Here, one thing that need be clarified is that all of our editing scenarios are carried out on the OOA class diagram (stored in the Pounamu server with name *VT_Class_diagram_VideoOOA*) modelled for a Video Store system (an example application designed for teaching software engineering practices). In addition, you can refer to Appendix A to obtain a detailed description of the UML tool specified for the modelling work.

5.7.1 Example of Loading a PounamuView Diagram

When a user uses a web browser to access the Pounamu tool, a welcome page appears. On this page (shown in Figure 5-2(a)), the user can click the left menu bar to specify and load a PounamuView diagram. However, before doing that, the user must specify in (b) which modelling project this view diagram belongs to. Then in (c) the PounamuView diagram *VT_Class_diagram_VideoOOA* is chosen to be loaded into the web browser, and the result of this loading operation is shown in (d).


(a)
Welcome to our Pounamu tool! - Microsoft Internet Explorer
Address http://localhost:8080/PounamuWebApp/controllerservlet
Pounamu model things with your own tool
Home
Tutorial
Contact Us
Menu
Specify Pounamu project & view
New ModelProject
Load ModelProject
New Pounamuview
Load Pounamuview
Refresh Diagram
Primitive Edit Command
Add Entity
Add Association
Set Properties
Resize Entity
Move Entity
Move Association
(b)
Load model project! - Microsoft Internet Explorer
Address http://localhost:8080/PounamuWebApp/controllerservlet?action=LoadProject
Model projects in Pounamu server:
Model_Tryy.xml
Model_BookOrderOnline.xml
Model_VideoRental.xml
Model_TryNEW.xml
Model_Rtr.xml
Model_Eee.xml
Model_Ccc.xml
Model_Ggg.xml
(c)
Select viewType in select
Address http://localhost:8080/PounamuWebApp/controllerservlet?action=LoadPounamuView
Select PounamuView in the model project named Model_VideoRental:
VT_Class_diagram_VideoOOA
Select Pounamuview

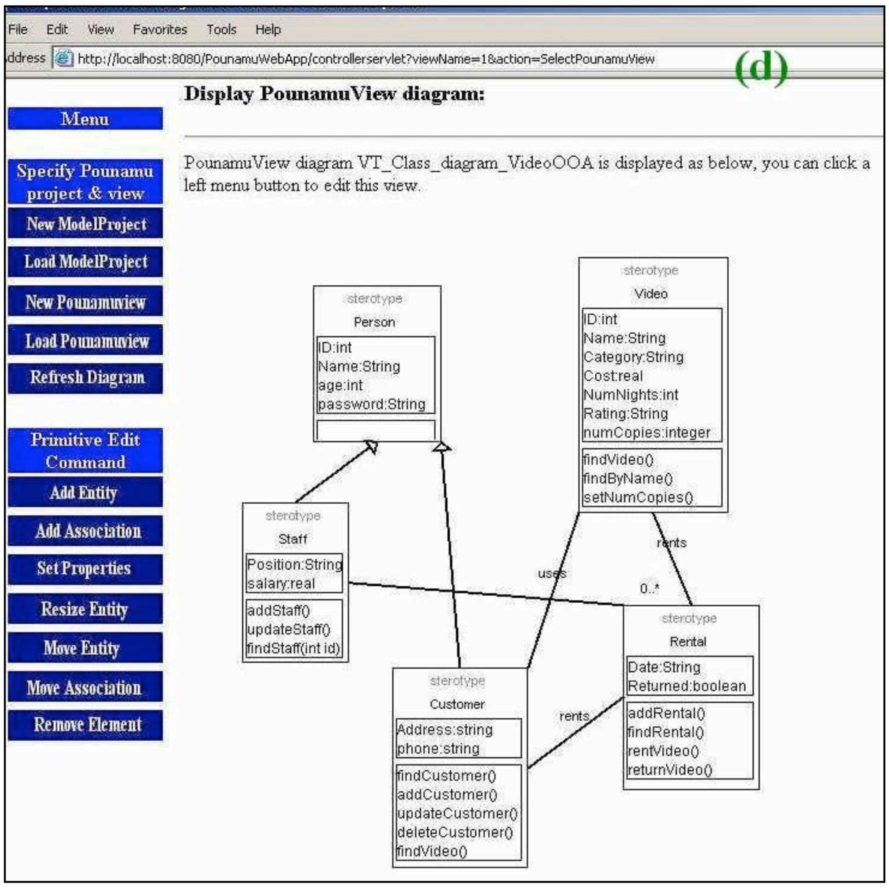
Welcome to our pounamu tool!

Please first load PounamuView diagrams of a model project by clicking left Load Pounamuview button!

List of model project names: Model_Tryy.xml

Selected project Name: *Model_VideoRental*

A list of Pounamu views in the project: VT_Class_diagram_VideoOOA



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 5-2 Screen Dumps of Loading and Specifying a PounamuView Diagram

5.7.2 Example of Removing an Entity Shape

To demonstrate how the removing action works, we decide to delete the “Staff” class object from the diagram shown in Figure 5-3(a). First, when the user clicks the “Remove Element” button on the menu bar, a web page (b) appears to suggest that the user can choose a Pounamu model element for removal. After making the selection, the target object is highlighted by surrounding red rectangles. In addition, a message related to this removal editing is displayed on the top of the web page (c). Next, the user needs to confirm the operation before submitting the action. Finally, a result page that looks like (d) is provided.

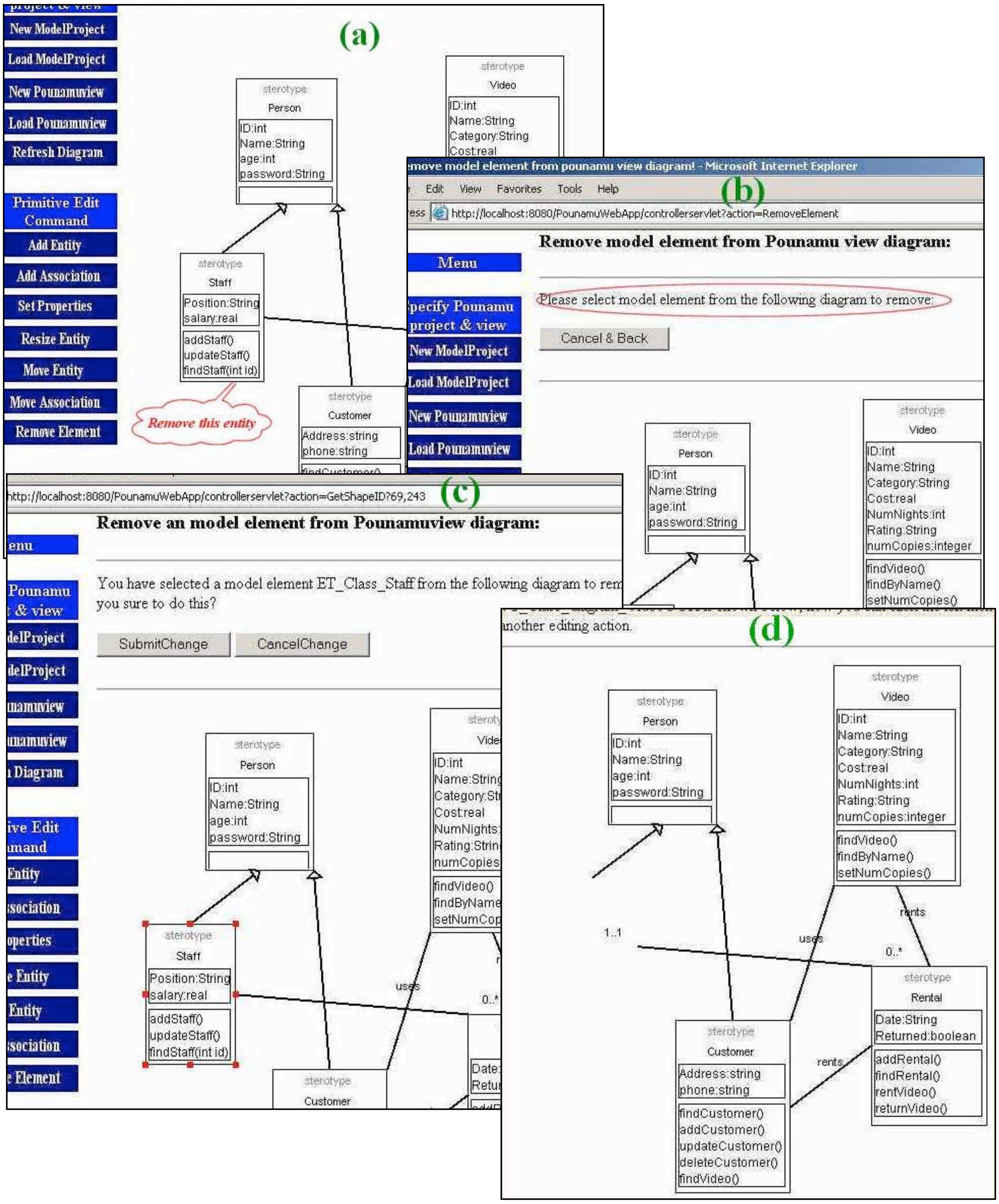


Figure 5-3 Screen Dumps of Removing an Entity Shape

5.7.3 Example of Removing an Association Connector

Since the steps to perform this "Remove Association Connector" are similar to the previous one, the detailed description is not pursued. However, we do give figures (shown in 5-4(a)-(c)) to demonstrate this type of editing action.

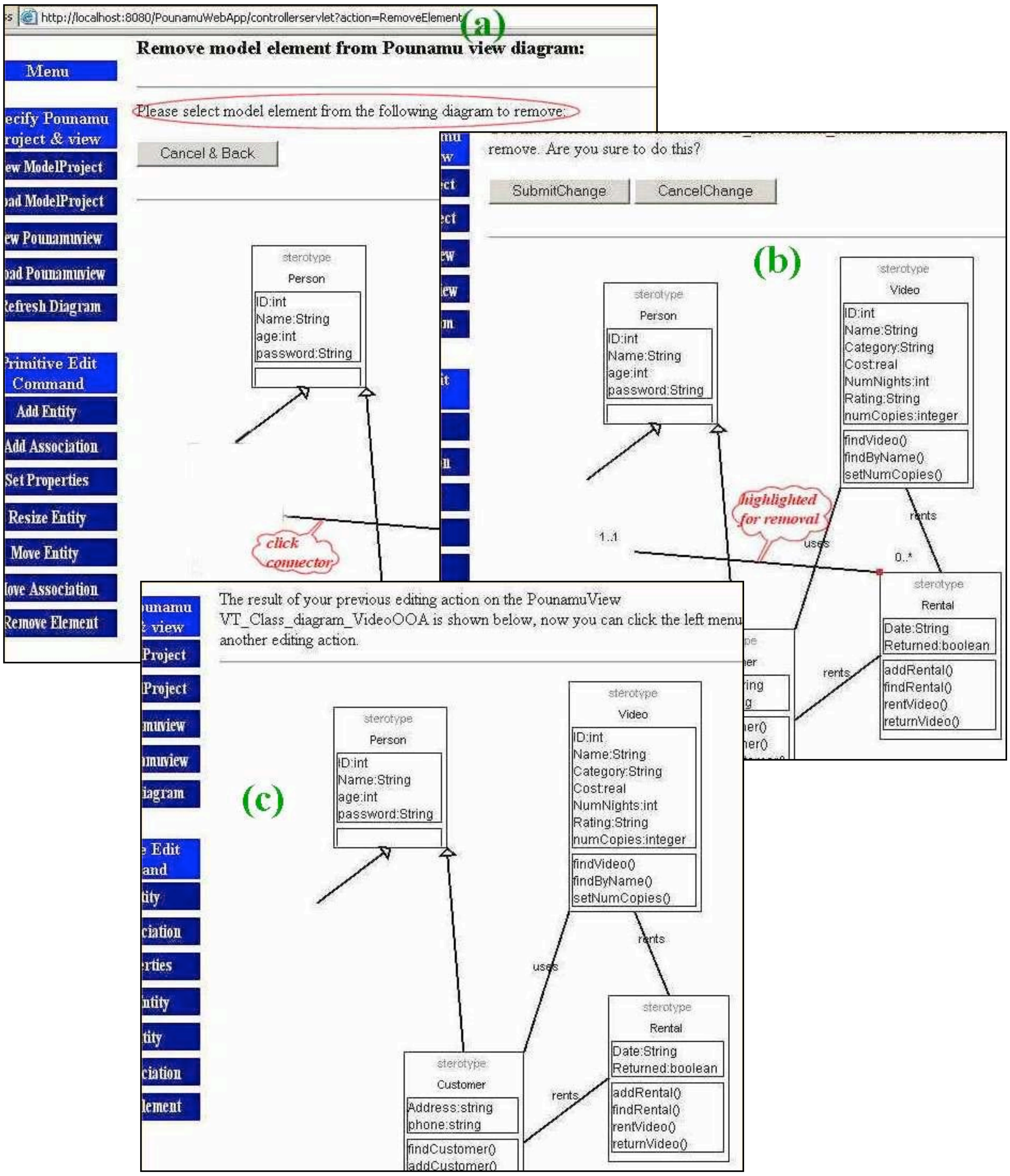
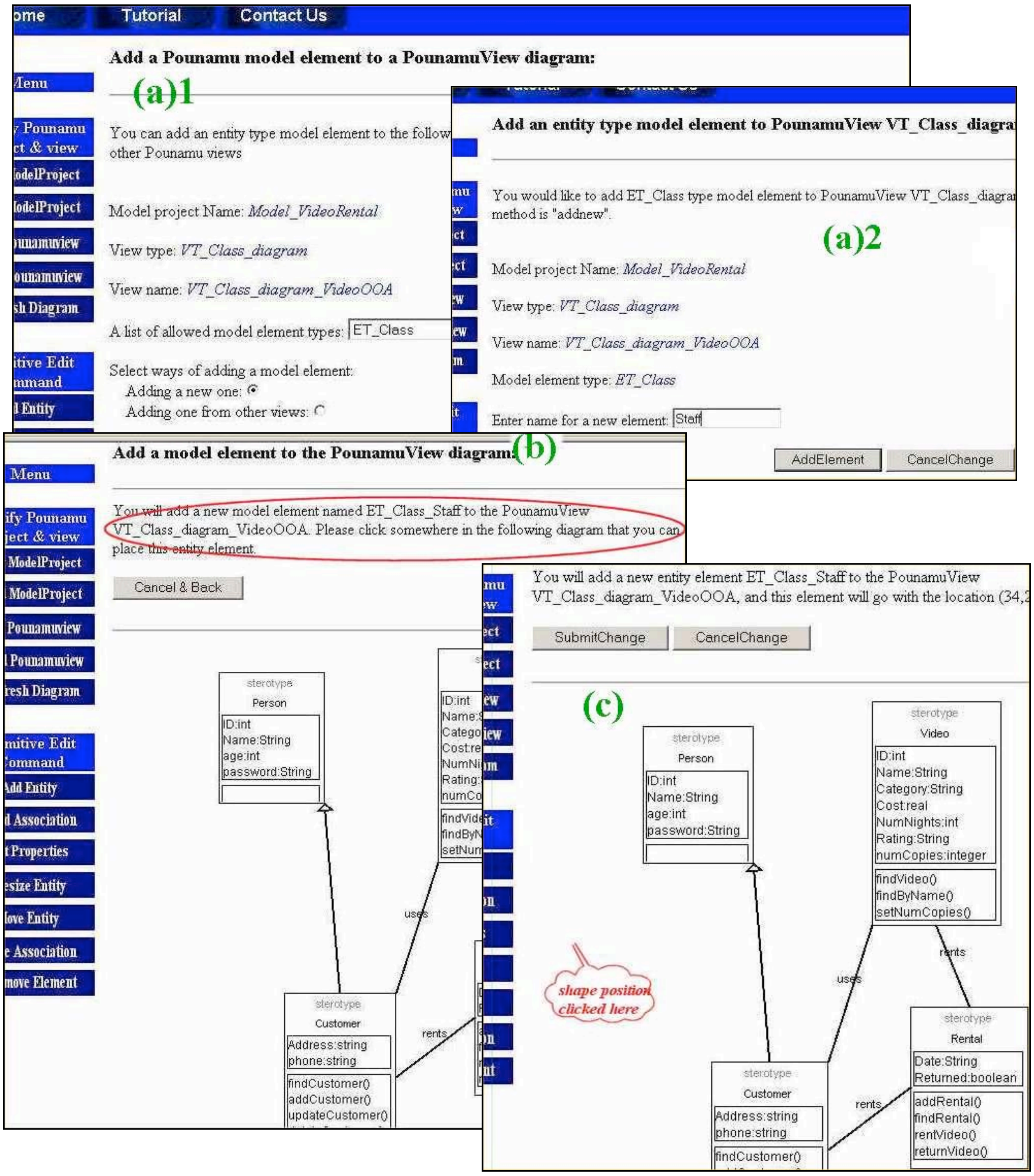


Figure 5-4 Screen Dumps of Removing an Association Connector

5.7.4 Example of Adding an Entity Shape

Here we add back the entity shape that has been deleted in 5.7.2. To do that, the “Add Entity” menu item has to be clicked first, and a page as shown in Figure 5-5(a) appears to allow the user to specify entity parameters (type, name etc.). After entering proper values for those parameters, a page (b) comes up suggesting what to do next. A message displayed on the top of this page requests specification of a location for this new entity object by clicking a preferred position in the diagram. The result of clicking is shown in (c). Finally, the updated diagram is presented in (d).



Add a Pounamu model element to a PounamuView diagram:

You can add an entity type model element to the follow
other Pounamu views

Model project Name: *Model_VideoRental*

View type: *VT_Class_diagram*

View name: *VT_Class_diagram_VideoOOA*

A list of allowed model element types: ET_Class

Select ways of adding a model element:
Adding a new one: [✓ ●]
Adding one from other views: ○

Add an entity type model element to PounamuView VT_Class_diagra

You would like to add ET_Class type model element to PounamuView VT_Class_diagra
method is "addnew".

Model project Name: *Model_VideoRental*

View type: *VT_Class_diagram*

View name: *VT_Class_diagram_VideoOOA*

Model element type: *ET_Class*

Enter name for a new element: Staff

AddElement CancelChange

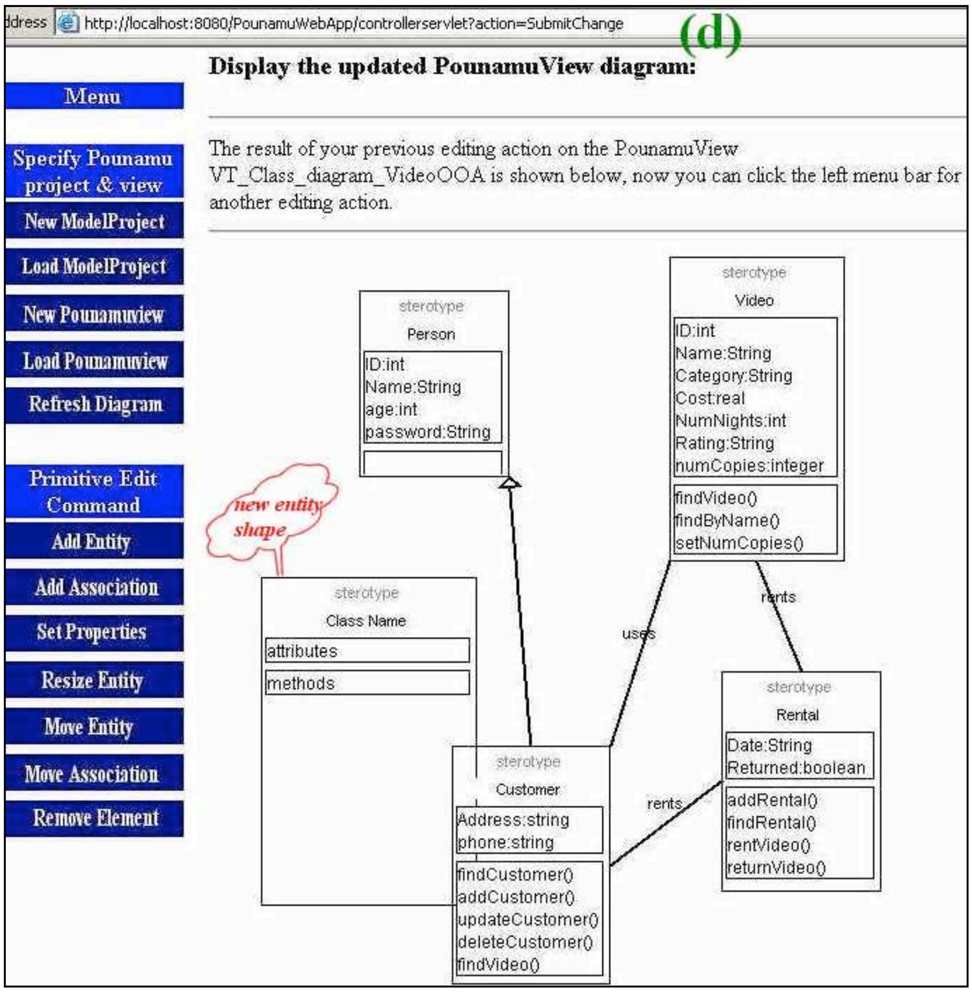
Add a model element to the PounamuView diagram:

You will add a new model element named ET_Class_Staff to the PounamuView VT_Class_diagram_VideoOOA. Please click somewhere in the following diagram that you can place this entity element.

Cancel & Back

You will add a new entity element ET_Class_Staff to the PounamuView VT_Class_diagram_VideoOOA, and this element will go with the location (34,2

SubmitChange CancelChange


ddress http://localhost:8080/PounamuWebApp/controllerservlet?action=SubmitChange

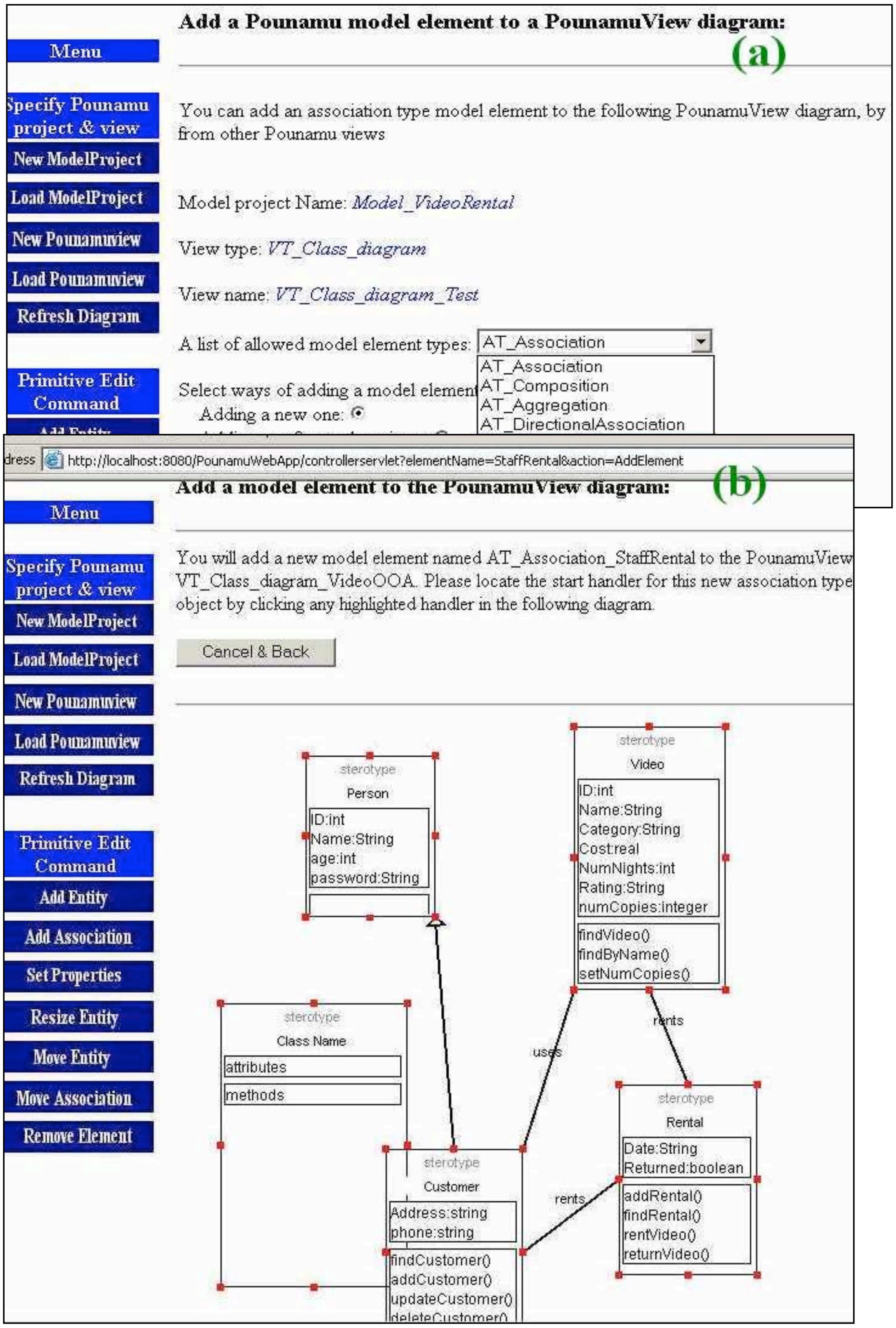
Display the updated PounamuView diagram:

The result of your previous editing action on the PounamuView VT_Class_diagram_VideoOOA is shown below, now you can click the left menu bar for another editing action.

Figure 5-5 Screen Dumps of Adding an Entity Shape

5.7.5 Example of Adding an Association Connector

In this subsection, we introduce how to add an association connector between two entity shapes, and the whole process is presented in Figure 5-6. First, the "Add Association" button is clicked to initiate this type of editing action. A page (a), seen before, appears so that the connector's parameters (name, type etc.) can be entered. After specifying appropriate parameters for this object, a view diagram (b) is shown with all shapes being highlighted by red handles. The next thing is to decide which two shapes are to be connected. This is done by specifying the start handle from the first shape (c), and the end handle from the second shape (d). Finally, you can see the result of this operation in (e).



Add a Pounamu model element to a PounamuView diagram:

You can add an association type model element to the following PounamuView diagram, by from other Pounamu views

Model project Name: *Model_VideoRental*

View type: *VT_Class_diagram*

View name: *VT_Class_diagram_Test*

A list of allowed model element types: AT_Association

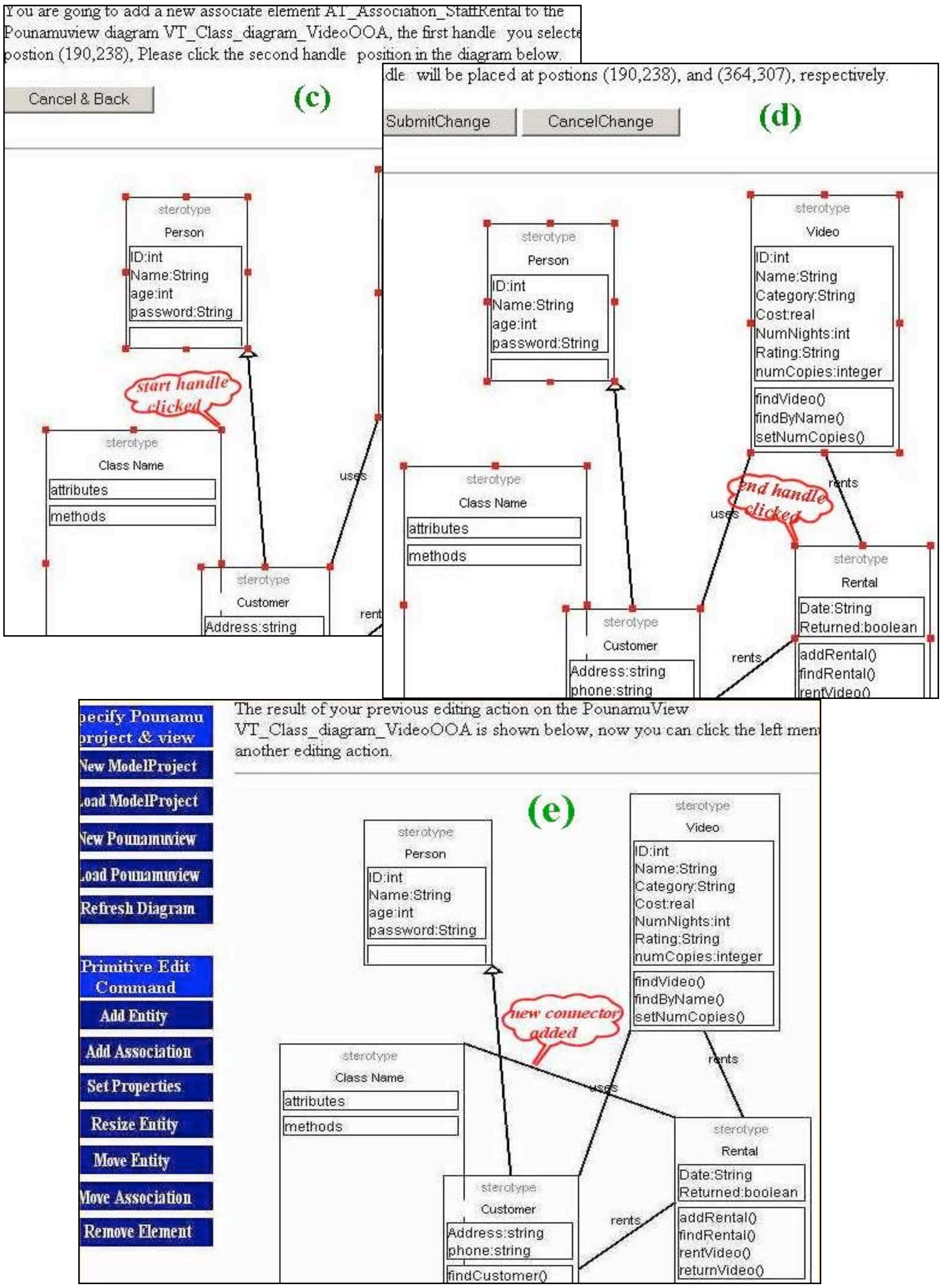
Select ways of adding a model element

Adding a new one: ◉

dress http://localhost:8080/PounamuWebApp/controllerservlet?elementName=StaffRental&action=AddElement

Add a model element to the PounamuView diagram:

You will add a new model element named AT_Association_StaffRental to the PounamuView VT_Class_diagram_VideoOOA. Please locate the start handler for this new association type object by clicking any highlighted handler in the following diagram.



You are going to add a new associate element AT_Association_StaffRental to the Pounamuview diagram VT_Class_diagram_VideoOOA, the first handle you selecte postion (190,238), Please click the second handle position in the diagram below.

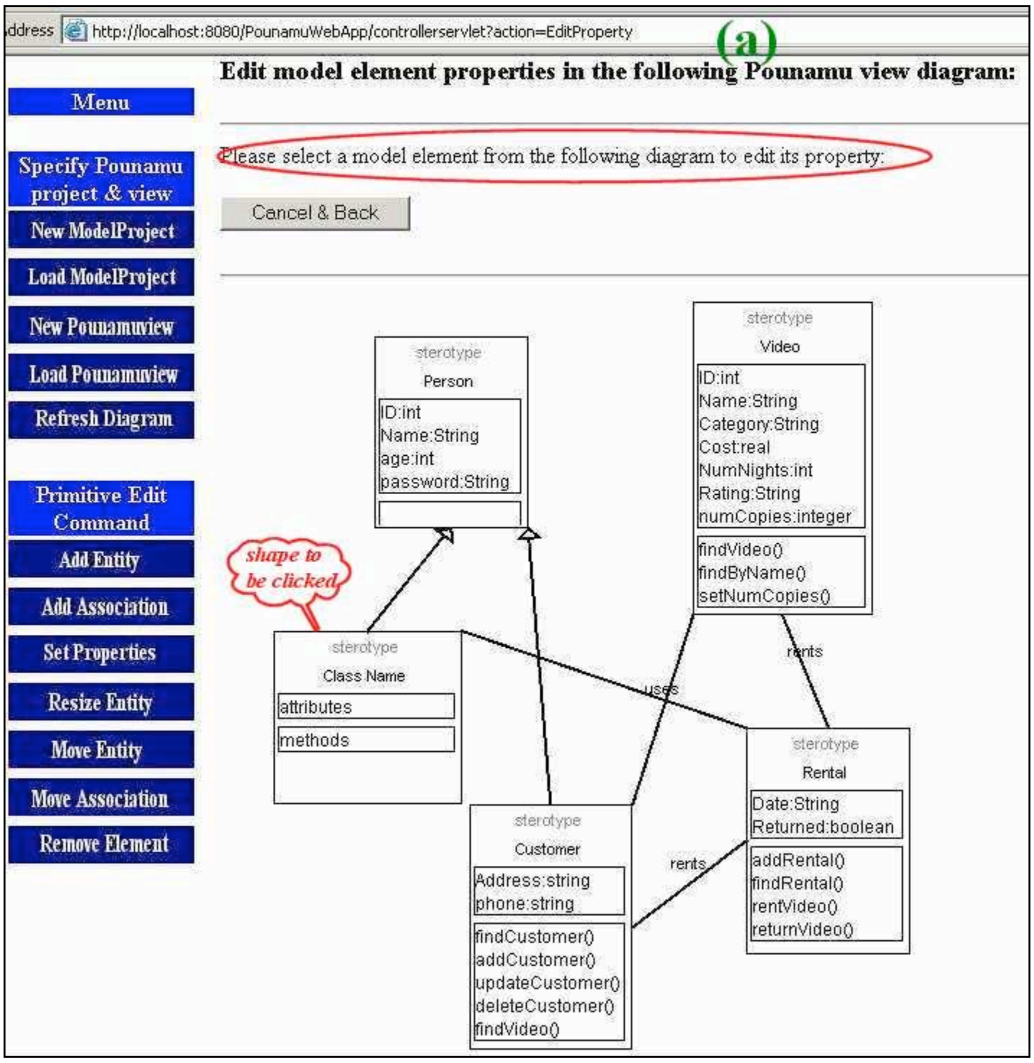
dle will be placed at postions (190,238), and (364,307), respectively.

The result of your previous editing action on the PounamuView VT_Class_diagram_VideoOOA is shown below, now you can click the left men another editing action.

Figure 5-6 Screen Dumps of Adding an Association Connector

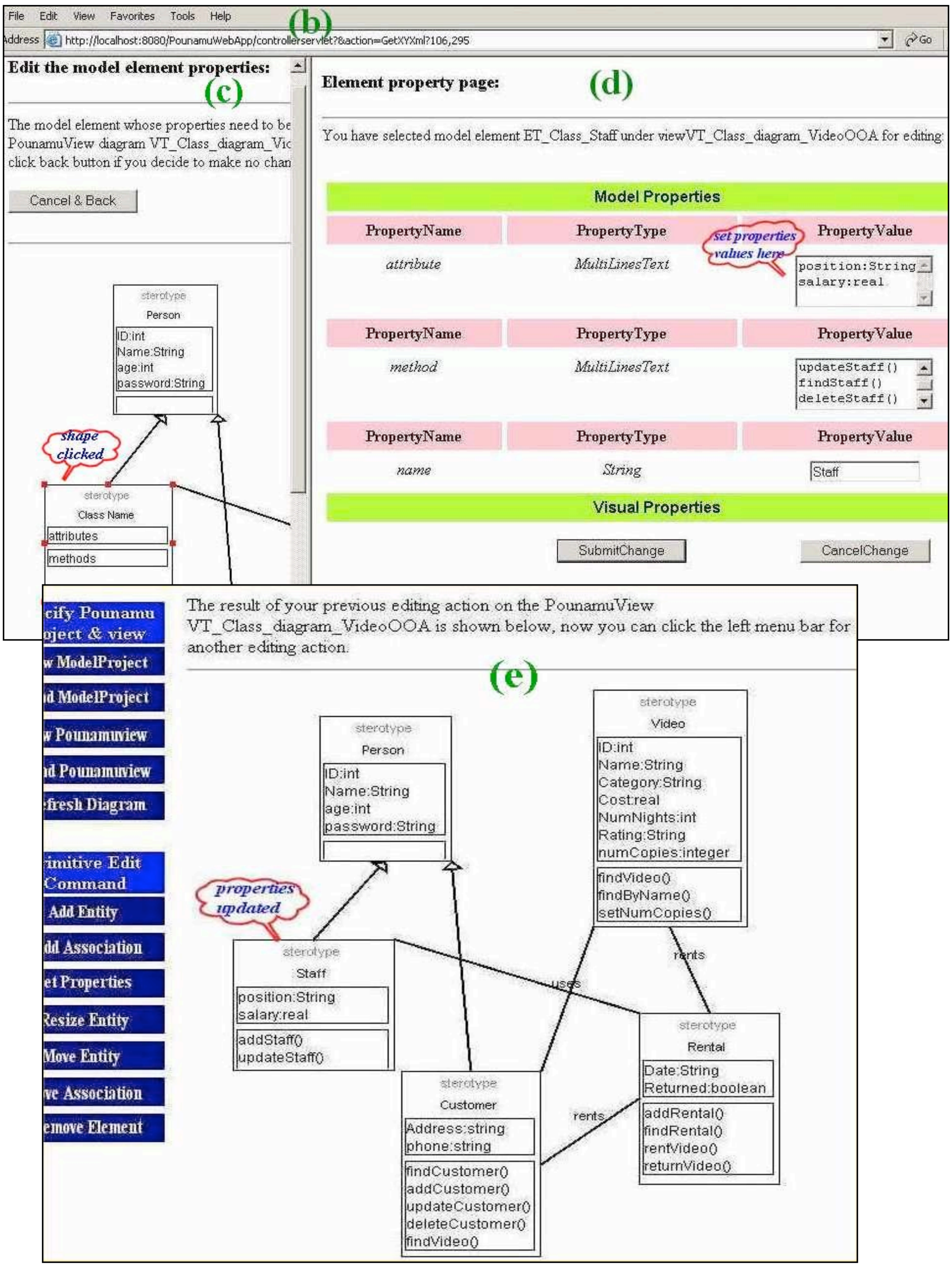
5.7.6 Example of Property Setting

After adding back the previous entity shape or association object, the next thing is to set their model and visual properties. While model properties correspond to data contained within an entity or association object, visual properties influence its visual representation. The whole process for this property setting action is illustrated in Figure 5-7. First, this action is initiated by clicking the “Set Properties” menu item, and a page (a) appears waiting for the user’s choice of the target object. Similar to those editings that have been described, the selection is made through clicking the correct shape/connector in the displayed diagram. After that, the selected target is annotated by red rectangle handles in the left frame (c) of the web page (b), while its properties (name, type and value) are listed in the right frame (d) of the web page (b). By entering new values into the property list and submitting them, you can obtain a new version of diagram with the target’s properties values updated. Figure 5-7(e) shows the result of the property setting editing action.


ddress http://localhost:8080/PounamuWebApp/controllerservlet?action=EditProperty

Edit model element properties in the following Pounamu view diagram:

Please select a model element from the following diagram to edit its property:



Edit the model element properties:

The model element whose properties need to be PounamuView diagram VT_Class_diagram_Vic click back button if you decide to make no chan

Element property page:

You have selected model element ET_Class_Staff under viewVT_Class_diagram_VideoOOA for editing

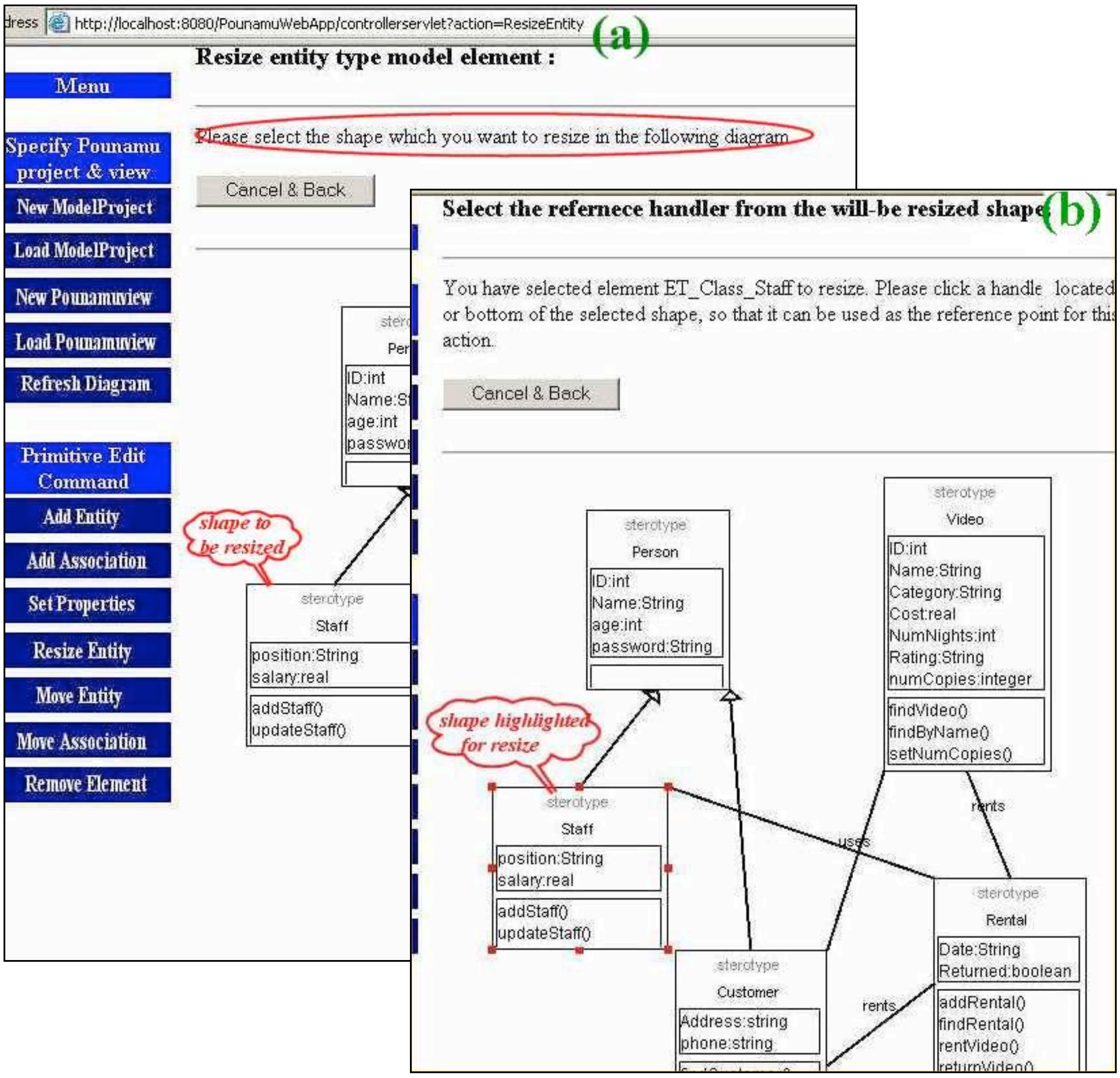
Model Properties		
PropertyName	PropertyType	PropertyValue
attribute	MultiLinesText	position:String salary:real
PropertyName	PropertyType	PropertyValue
method	MultiLinesText	updateStaff() findStaff() deleteStaff()
PropertyName	PropertyType	PropertyValue
name	String	Staff
Visual Properties		

The result of your previous editing action on the PounamuView VT_Class_diagram_VideoOOA is shown below, now you can click the left menu bar for another editing action.

Figure 5-7 Screen Dumps of Property Setting Example

5.7.7 Example of Resizing an Entity Shape

From Figure 5-7(e), you can see part of the property values have been blocked by the boundary of the entity shape, so the resizing shape action comes in and is used to adjust a shape to its appropriate size. To initiate this action, the user needs to click the “Resize Entity” menu item, and then a web page as shown in Figure 5-8(a) appears to remind the user to click the target shape. As before, the selection is highlighted with red handles (b). The next task is to click a red rectangle handle of the target so that it can serve as the resizing reference point (c). After that, the user needs to click somewhere in the diagram that the reference point is expected to reach as the result of the resizing (d). The system calculates the resizing proportion based on the values provided and acts accordingly. Finally, the updated diagram is as shown in Figure (e).

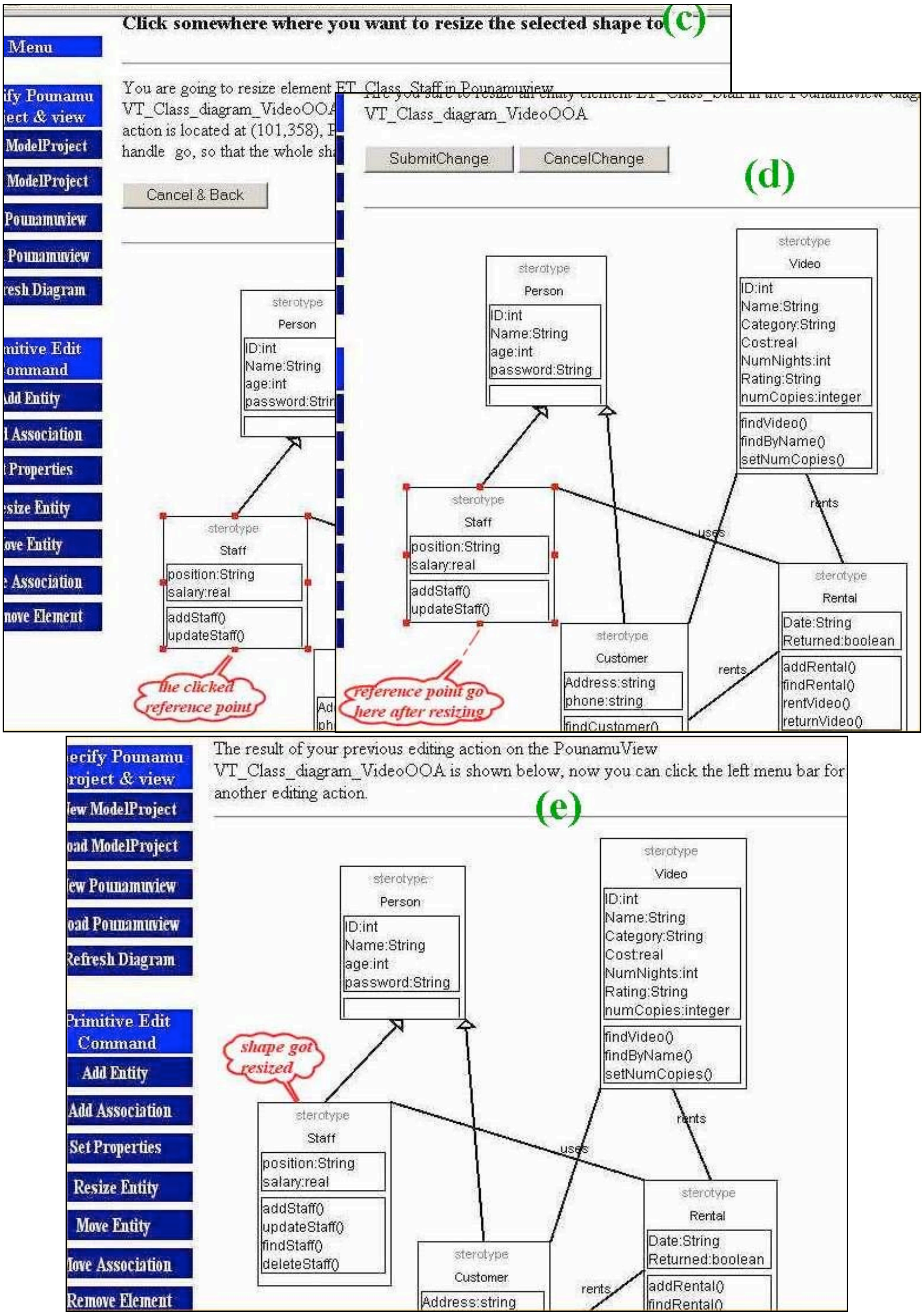

dress http://localhost:8080/PounamuWebApp/controllerservlet?action=ResizeEntity

Resize entity type model element :

Please select the shape which you want to resize in the following diagram

Select the refernece handler from the will-be resized shape

You have selected element ET_Class_Staff to resize. Please click a handle located or bottom of the selected shape, so that it can be used as the reference point for this action.



Click somewhere where you want to resize the selected shape to

You are going to resize element ET_Class_Staff in Pounamuview
VT_Class_diagram_VideoOOA
action is located at (101,358), P
handle go, so that the whole sh

VT_Class_diagram_VideoOOA

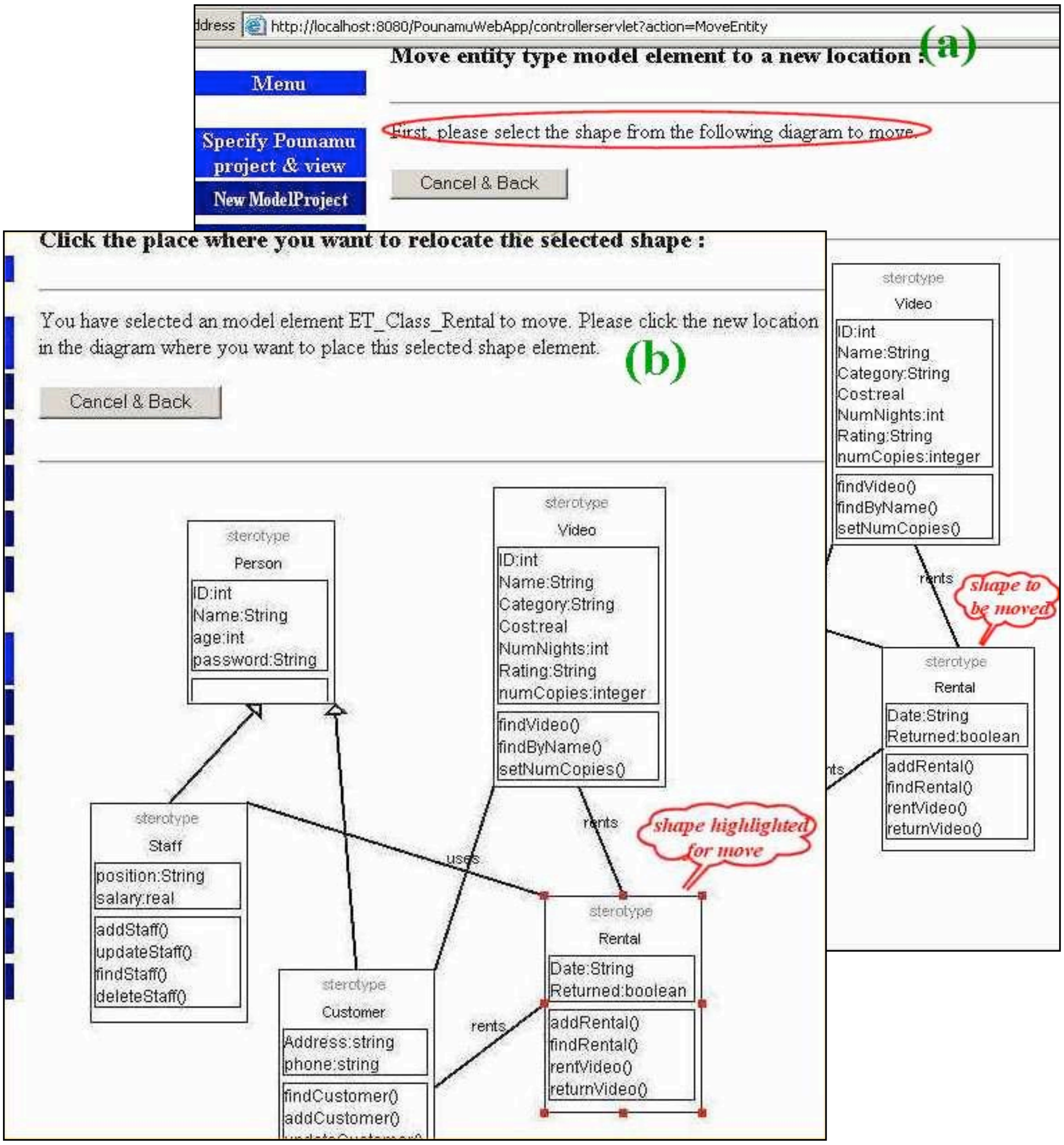
The result of your previous editing action on the PounamuView VT_Class_diagram_VideoOOA is shown below, now you can click the left menu bar for another editing action.

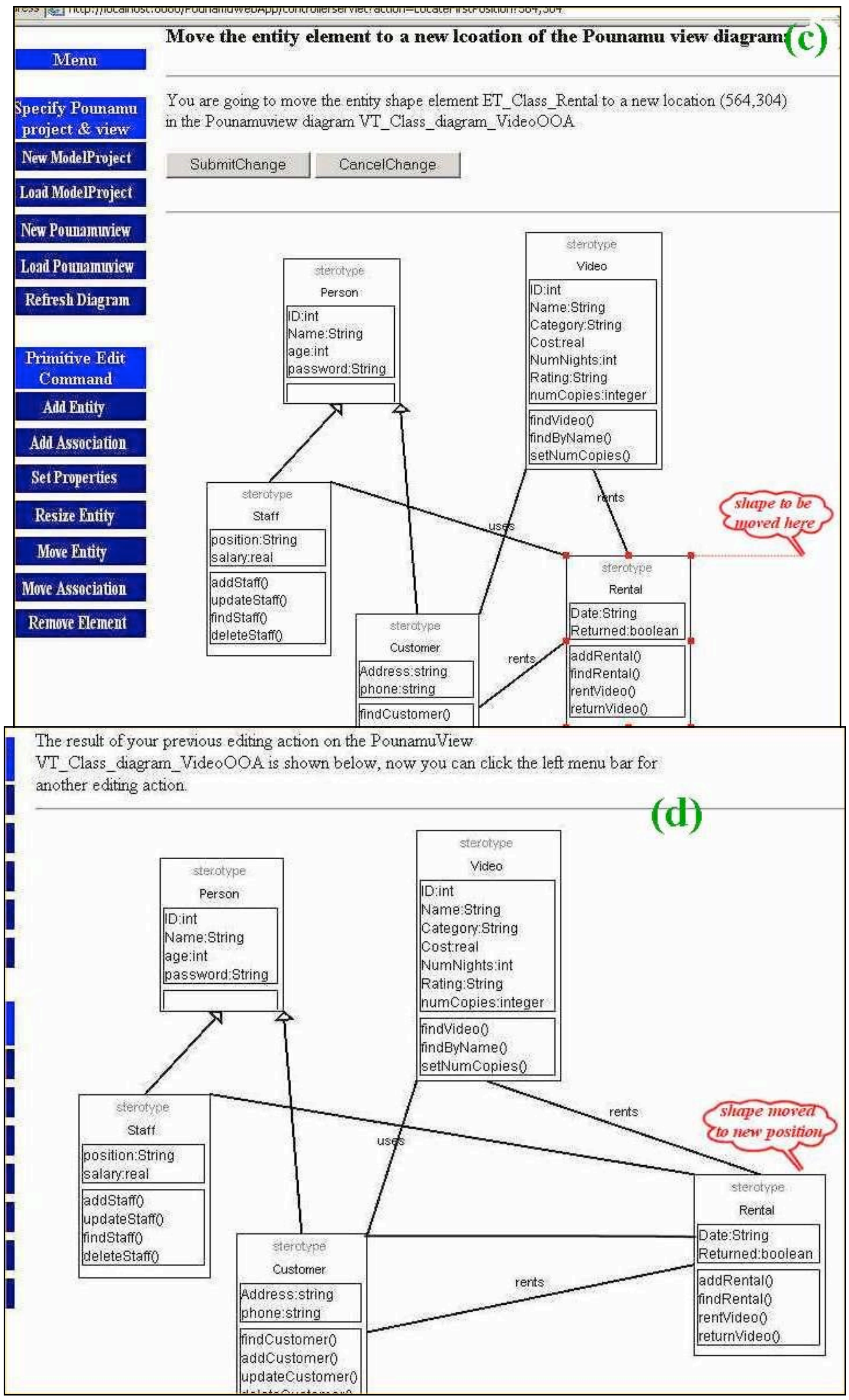
(c)
Menu
ify Pounamu ject & view
ModelProject
ModelProject
Pounamuview
Pounamuview
resh Diagram
mitive Edit ommand
Add Entity
Association
Properties
esize Entity
ove Entity
Association
nove Element
Cancel & Back
sterotype Person ID:int Name:String age:int password:String
sterotype Staff position:String salary:real addStaff() updateStaff()
the clicked reference point
(d)
SubmitChange
CancelChange
sterotype Video ID:int Name:String Category:String Cost:real NumNights:int Rating:String numCopies:integer findVideo() findByName() setNumCopies()
sterotype Rental Date:String Returned:boolean addRental() findRental() rentVideo() returnVideo()
sterotype Customer Address:string phone:string findCustomer()
uses
rents
rents
reference point go here after resizing
(e)
ecify Pounamu roject & view
New ModelProject
oad ModelProject
ew Pounamuview
oad Pounamuview
Refresh Diagram
Primitive Edit Command
Add Entity
Add Association
Set Properties
Resize Entity
Move Entity
Move Association
Remove Element
sterotype Staff position:String salary:real addStaff() updateStaff() findStaff() deleteStaff()
shape got resized

Figure 5-8 Screen Dumps of Resizing an Entity Shape

5.7.8 Example of Moving an Entity Shape

There are times when the user wants to change the layout of a PounamuView diagram, so the moving action is another indispensable editing action. Steps involved in the moving shape action are shown in Figure 5-9. First, similar to all the other editings, this action is initiated by clicking the corresponding menu item, and the page shown in (a) comes up. Following is a click on the target shape, causing the target to be highlighted in the usual way (b). Next, the user needs to click somewhere in the diagram to set the preferred position for the target object(c). After committing this editing by clicking the "SumbitChange" button, you obtain the result as in (d).




Move the entity element to a new lcoation of the Pounamu view diagram

You are going to move the entity shape element ET_Class_Rental to a new location (564,304) in the Pounamuview diagram VT_Class_diagram_VideoOOA

The result of your previous editing action on the PounamuView VT_Class_diagram_VideoOOA is shown below, now you can click the left menu bar for another editing action.

Figure 5-9 Screen Dumps of Moving an Entity Shape

5.8 Implementation Experiences

Implementation experiences accumulated in the development of this GIF-version thin-client Pounamu user interface are summarized below:

Good Experience in Using Pounamu API

We have experienced the rich API provided by Pounamu for its further extension. For example, our remote editing component is completely built on the existing Pounamu API and can be arbitrarily plugged in or out without influencing other functionalities. However, since Pounamu is still undergoing constant improvement in functionality and the overall structure, we cannot ensure that APIs used here aren't changed. In summary, our experience in using the Pounamu API for the design of additional components is very good.

On-the-fly Gif Image Generation using Package Java.awt.image.*

I had expected that generating GIF images for Java AWT or Swing components would be a tough task until the useful java class "java.awt.image.BufferedImage" was discovered. With this class, it's easy to write any AWT or Swing components into a buffered image, and then encode the buffered image into the required formats (e.g. GIF or JPEG). Another thing worth mentioning is that the GIF encoder class from ACME LABS [59] has been used in this application since there are no similar API classes provided by SUN Java.

Using XML As Message Exchange Format

As said many times, XML has been chosen as the message exchange format between the remote Pounamu server and the Web server. Here we won't repeat XML's advantages over other similar technologies, however, we do feel it is necessary to mention XML's capability to transfer binary data as part of itself. With this feature, the generated GIF image can be encoded in an XML document to be sent across RMI, so that the necessity of writing a serializable java object for the image is avoided. The base64-encoding scheme [60], specified in RFC 2045 - MIME (Multipurpose Internet Mail Extensions), is commonly used to encode and embed binary data in an XML document.

Limited Html Resources to Represent Complex Element Properties

In this prototyping system, there is no client-side scripting (such as Jscript, JavaScript) used. The purpose is to see how far this diagramming tool can go as a strict thin-client computing environment. The result shows that it's only possible to implement a simplified version of the original Pounamu tool. One obvious problem is that HTML is very limited in its ability to vividly represent some of model element's visual properties. For example, visual properties such as colour or line width can only be represented with HTML form combo-boxes. So, to improve the system usability, small quantities of client-side scripts should be tolerable.

RMI Security Issues

As mentioned before, RMI is chosen for its ease of use and being able to send java objects across the network. However, my personal experience shows that RMI's ease of use is true only under the condition that you have worked around its security manager mechanism. If not, nasty error messages such as "Access Denied" occur. You may ask why this security mechanism is needed. Actually, RMI's security mechanism is designed to secure interprocess communication over a network. With this mechanism, you can restrict the actions performed by remotely loaded classes. Otherwise, you may inadvertently allow unsecure code to access private system resources. The suggested solution to work out this security barrier is to include a policy file in the application. This policy file is accessed by the RMI Manager to determine which activities are acceptable.

5.9 Summary

In this chapter, we have discussed implementation details involved in building the GIF version of thin-client Pounamu user interface. The discussion focused on three areas. First we briefly introduced tools and technologies that have been used to assist in implementation, including Tomcat, XML, RMI/CORBA, and JSP/Servlets. Following that, we described how the critical parts of the prototype system are implemented, and code samples have been used to make the description clear. Finally, examples of the system running were presented to demonstrate how users interact with it.

In the next chapter, we will present how the SVG version thin-client Pounamu tool is implemented. You will notice that some implementation details are omitted because of the close similarity to the GIF counterpart.

Chapter 6: Implementation of the SVG-based Prototype

The purpose of this chapter is to discuss design and implementation concerns in building the SVG version of the thin-client Pounamu user interface. The structure of this chapter is similar to the previous one, and consists of an explanation of the required technologies and tools, description of how the different parts are implemented with code samples, as well as presentation of the system running with screen dumps. Since part of the implementation of the previous GIF version was reused here, for the sake of conciseness, only those SVG-specific technologies and implementation details are discussed.

6.1 Technologies to Use

In this section, the discussion focuses on what SVG is, and its advantages and disadvantages compared with other web graphics formats. Additionally, we give a brief introduction to the available SVG viewer plug-ins.

6.1.1 SVG

Scalable Vector graphics (SVG) is a language for describing two-dimensional graphics in XML [61]. SVG was recommended and created by the World Wide Web Consortium (W3C), who also created HTML and XML, among other important standards and vocabularies. More than twenty organizations, including Sun Microsystems, Adobe, Apple, IBM, and Kodak, have been involved in defining SVG [61]. Its emergence is changing the role of graphics on the Web.

Today, GIF and JPEG, as two most commonly used graphics on the web, are pixel-based. These graphics formats contain information about each and every pixel in the image. Although pixel-based systems have their own strength, such as the ability to faithfully recreate photographic images [64], they also have several weaknesses. First, zooming and scaling a pixel-based image often results in jagged edges because values for pixels that don't exist in the original image have to be interpolated [64]. Second, a pixel-based image file is normally big since a colour value is needed for each and every pixel in the image. Another weakness is that the binary nature of pixel-based image formats make it difficult (although not impossible) to create "on the fly" images based on database information [64].

SVG, on the other hand, overcomes the above limitations. The fact that SVG files are text files with XML syntax tells that they can be easily manipulated through standard APIs (e.g. DOM) and transformed through XML Stylesheet Language Transformation (XSLT). Integration of SVG into the existing DOM enables the developer to fully access SVG elements by the usual Java/JavaScript interfaces [63]. Given that SVG is a vector-based standard, SVG graphics keep the same high quality no matter what display

```
<circle
   cx= “100”
   cy= “250”
    r= “50”
   style= “fill:green; stroke:red; stroke-width:4;”>
</circle>
```

devices (including PDA, laptops or TVs) are used [63]. Compared with pixel-based images such as JPEG and GIF, SVG files are much smaller and more compressible, which is very helpful to optimize the browser’s performance, and therefore often results in faster download speed [63]. In addition to the above features, SVG has other advantages over the pixel-based formats, which are generalized as follows [62]:

Compact and Flexible file format. An SVG file, being a text-only collection of XML commands, is editable in any simple text editor and can be easily generated from server-side tools such as CGI and JSP/Servlet.

Same high resolution at any zoom or resize. This is an important feature for any vector-based image, including SVG, VRML, flash etc.

Compatibility and integration: SVG, as a web graphics standard specified by W3 organization, can be easily integrated with other web technologies such as HTML, Java Scripts, CGI and JSP etc. Therefore, great flexibility is allowed in design and implementation of the web-based applications.

Dynamic interactivity: By using a supplemental scripting language to access the SVG DOM (which provides complete access to an individual SVG graphical object), users can enjoy rich interaction with web graphics. Generally speaking, such a degree of interactivity is difficult to achieve with any pixel-based image format.

High-quality printing: SVG images are printed with the same colors as are displayed on a screen and at full printer resolution.

6.1.2 SVG in practice

Having discussed some of the features of SVG, we now review some SVG examples. Figure 6-1(a) shows the SVG code defined for a simple 2D circle, and the rendered result is in Figure 6-1(b). The attributes *cx* and *cy* give the centre of this circle while *r* and *style* specify its radius and visual appearance respectively.

Figure 6-1(a) SVG code for a 2D circle

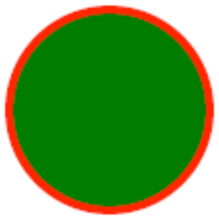
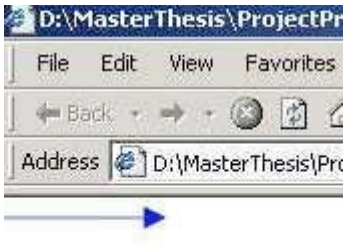


Figure 6-1(b) the Rendered result

Another example that is used to demonstrate some advanced SVG graphic elements is shown in Figure 6-2. In this example, an arrow has been defined by grouping a *line* and a closed *path* (SVG low-level element) together into a high-level *g* element. Then by defining this arrow group as a *symbol*, it can be used as a whole to describe a more complex diagram. In addition, any kind of transformation such as translation, stretch or rotation can be applied to this arrow element when it is used. An example of using this custom-defined group element is shown in Figure 6-3(a) and Figure 6-3(b).



```
<g style="fill:blue; stroke:blue">
  <line x1="0" x2="70"  y1="10" y2="10"/>
  <path d="M70 5 L70 15  L80 10 z"/>
</g>
```

Figure 6-2 An Arrow Defined in SVG group Element

```
<symbol id="Arrow" preserveAspectRatio="none" viewBox="0 0 80 20">
  <g style="fill:blue; stroke:blue">
  <line x1="0" x2="70" y1="10" y2="10"/>
  <path d="M70 5 L70 15 L80 10 z"/>
  </g>
</symbol>
<use height="20" width="80" x="50" y="0" xlink:href="#Arrow"/>
<use height="20" width="80" x="50" y="0" xlink:href="#Arrow"
  transform="rotate(190 70 30)"/>
<use height="20" width="80" x="50" y="0" xlink:href="#Arrow"
  transform="translate(0,20)"/>
```

Figure 6-3(a) SVG Code Examples Using the Arrow Defined in Figure 6-2

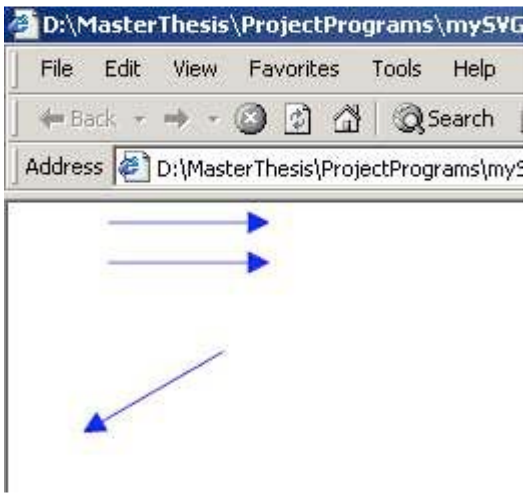

D:\MasterThesis\ProjectPrograms\mySVG
File Edit View Favorites Tools Help
Back Search
Address D:\MasterThesis\ProjectPrograms\myS

Figure 6-3(b) the rendered result from SVG code shown in (a)

Since most editing actions required in the thin-client Pounamu tool are realized through interacting with diagrams, the last example is given to highlight SVG's interactivity. Before doing that, we will look at how interaction is possible with SVG. First, SVG specifies that event handlers such as *onclick*, *onmouseover* can be associated with individual SVG elements. Second, as stated earlier, script elements written in a scripting language are allowed to be included into SVG documents, and these scripts can manipulate all aspects of SVG by interacting with the SVG DOM. When registering these scripts as event handlers

(a) Circle with radius 100 ***(b) Circle with radius 200***

through assignment, such as "*onclick=scriptfunction*", script functions will be executed when corresponding events are triggered. For example, the code sample below shows the script function *circle_click(evt)* has been registered with the *onclick* event of a *circle*, so with each mouse click the radius of the circle is toggled between 100 and 200. Figure 6-4(a) and (b) present a running instance.

```
<!-- ECMAScript to change the radius with each click -->
  <script type="text/ecmascript"> <![CDATA[
    function circle_click(evt) {
      var circle = evt.target;
      var currentRadius = circle.getAttribute("r");
      if (currentRadius == 100)
        circle.setAttribute("r", currentRadius*2);
      else
        circle.setAttribute("r", currentRadius*0.5);
    }   ]]> </script>

  <!-- Outline the drawing area with a blue line -->
  <rect x="1" y="1" width="598" height="498" fill="none" stroke="blue"/>

  <!-- Act on each click event -->
  <circle onclick="circle_click(evt)" cx="300" cy="225" r="100"  fill="red"/>

  <text x="300" y="480"  font-family="Verdana" font-size="35" text-anchor="middle">
    Click on circle to change its size
  </text>
```

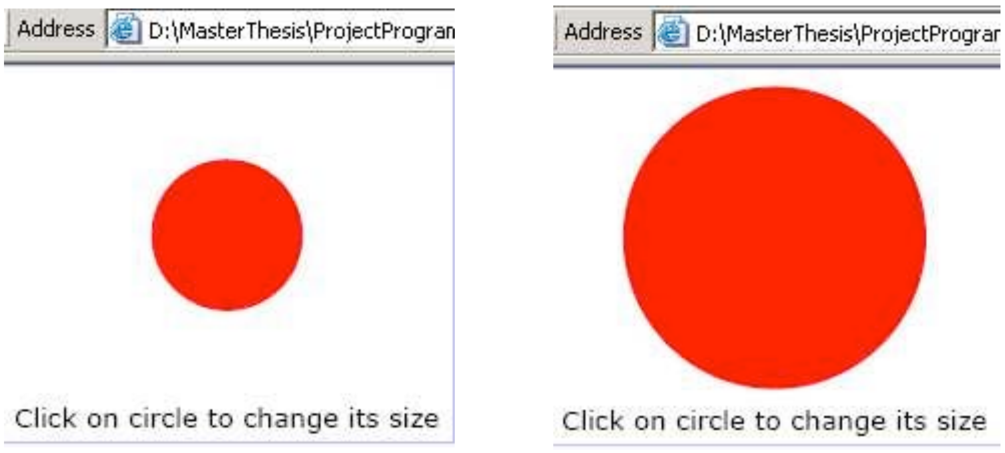

Address
D:\MasterThesis\ProjectProgran
Click on circle to change its size
Address
D:\MasterThesis\ProjectProgrär
Click on circle to change its size

Figure 6-4 SVG event handler examples

Although the SVG specification doesn't clearly indicate what scripting languages are supported in SVG, ECMAScript [65] seems to be the de-facto standard. Actually several ECMAScript compatible scripting languages (e.g. JavaScript, Jscript and ActionScript) are often seen binding with SVG.

6.1.3 SVG Implementations

Despite all of the advantages mentioned before, one big disadvantage that holds SVG back from becoming a standard web graphic is that it has not been directly supported by most web browsers. Viewing SVG graphics on a web page often requires a browser plug-in or other special software to be installed. Currently, there are numerous SVG implementations, including Adobe SVG plug-in [66], a pure java-based standalone SVG viewer [67], a KDE plug-in [68], the first native web-browser implementation in Mozilla [69] and Corel SVG renderer [70]. Here, only the first two of them, commonly used to view SVG images, are given a brief introduction.

Today, the most mature and widely-deployed SVG implementation is Adobe SVG Viewer plug-in [71]. For this reason, this SVG plug-in is employed in our project. Currently in *version 3.0*, Adobe SVG viewer is available for a wide range of platforms, including Macintosh, Microsoft Windows, Solaris and Linux operating systems. However, there are still some SVG elements, attributes, properties, and the DOM interfaces that have been proposed in the SVG Recommendation not implemented in this viewer (see [73] for details). For this reason, Adobe continues to innovate its SVG Viewer to comply with more of the SVG standard and enhance performance. It's said that *version6.0* will support most of the SVG specification, as well as advanced SVG DOM features [72].

A nice feature that Adobe SVG Viewer has is the inclusion of a JavaScript engine. Using this embedded scripting engine offers a few significant advantages. The main one is that its compliant implementation of the ECMA specification guarantees all the scripting code runs the same whether it is viewed on IE or Netscape, and whether on a Mac or a Windows machine [74]. Another nice feature is that server-connection facilities provided by Adobe SVG Viewer enable developers to open sockets to load from external data sources or store modified data [74]. The relevant methods for this server connection are getURL() and postURL(), both defined in the DOM Window object. An in-depth description of these methods is given when they are involved in implementation details.

The other main SVG implementation is Batik, the Java-based and open source XML project from Apache. There are two versions (1.1 and 1.5) of Batik that are downloadable from the Apache web site [67]. While Batik1.1 only supports static features of the SVG 1.0 specification, Batik1.5 (released in 2003) is focused on supporting SVG scripting fully. Actually Batik is more than a viewer, it really is a toolkit that provides an SVG to raster converter, server-side generation through DOM and APIs, a font converter, and a pretty printer [75]. Because of these features, as well as its java-based nature, Batik is suitable to be used in a Java-based web application. For example, Batik is obtainable on any Java platform and has been integrated most noticeably in Apache FOP [76] and ILOG Views [77].

6.2 SVG-Specific Implementation Details

In this section, some important issues relating to implementing this SVG-based thin-client Pounamu UI are examined. The discussion is mainly focused on three areas: how to dynamically display an SVG-version of the requested PounamuView diagram on a web browser, how to design and translate SVG events into the diagram editing commands, and how to implement multiple editing.

6.2.1 Displaying an SVG-version of a PounamuView Diagram on a Browser

First and foremost an SVG image of the requested PounamuView diagram has to be generated. As SVG conforms to the XML grammar, an SVG image can be written in the same way as the construction of normal XML documents. The difference is that most elements contained in the former come from one special namespace, SVG.

In our system, a special helper class *PounamuViewSVGTemplateGenerator* was coded for generating SVG images for a specified PounamuView diagram. Considering the tree structure of a typical PounamuView diagram (shown in Figure 6-5), this helper class contains seven *createSVGElementFor***()* methods with each designed for a special primitive Pounamu visual component (e.g. *PounamuConnector*, *PounamuPanel*, *PounamuMultipleLinesInput, PounamuHandle*, *PounamuLabel*, *PounamuButton*, *PounamuSingleLineInput*). In doing so, a universal generation solution has been found to construct an SVG image for any PounamuView diagram.

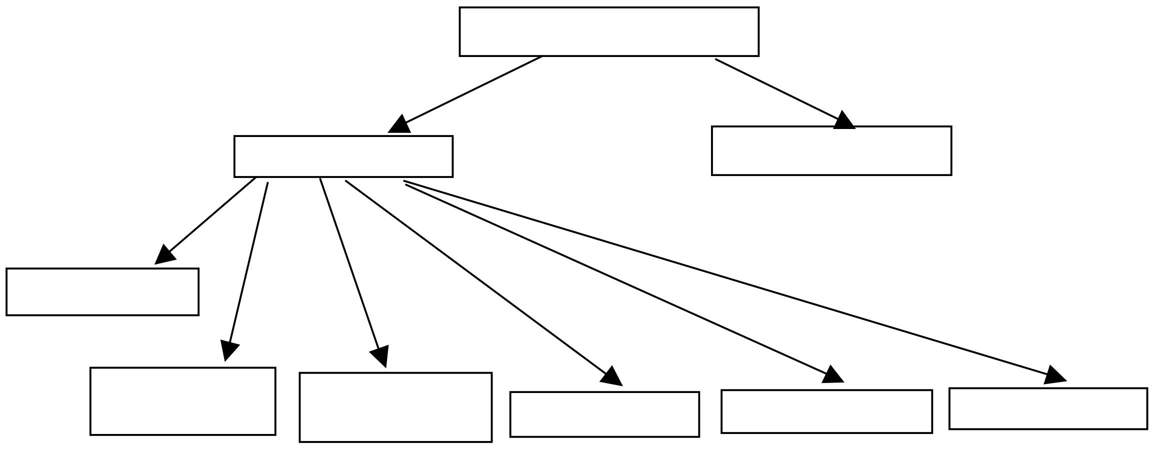


Figure 6-5 the Tree Structure of a typical PounamuView Diagram

Figure 6-6 illustrates an example of a PounamuView diagram (a) and the corresponding SVG image (b) obtained as the result of our program. Additionally, indications are given to show the mappings between a Pounamu icon and SVG elements. Here one thing worth special attention is that SVG elements for describing a Pounamu icon (shape/connector) have been grouped together as a *g* element with a certain ID. This arrangement suits our purpose of identifying all the SVG elements used to represent a certain

Pounamu icon as a whole. Moreover, by assigning event attributes such as "*onclick*" and "*onload*" to this *g* element (highlighted in green), mouse or keyboards events will be intercepted on a per Pounamu shape/connector basis. To complete this discussion, we presented below the skeleton code of *PounamuViewSVGTemplateGenerator* class.

```
public class PounamuViewSVGTemplateGenerator {

  /*PounamuViewSVGTemplateGenerator constructor*/
  public PounamuViewSVGTemplateGenerator(PounamuView view) {
  }

  /*Here generate SVG document string for the specified PounamuView diagram*/
  public String CreateSVGTemplateForView() {
     //When needed other primitive createSVGElementFor**** are called here
  }

  /*Create corresponding SVG element for PounamuConnector*/
  protected void CreateSVGElementForPounamuConnector(PounamuConnector conn, Element group){
     //write SVG element for a PounamuConnector and append it to group Element
   }

 /*Create corresponding SVG element for PounamuPanel*/
 protected void CreateSVGElementForPounamuPanel( PounamuPanel panel, boolean Outmost, Element
                    group,String clipPath){
     //write SVG element for a PounamuPanel and append it to group Element
  }

/*Create corresponding SVG element for PounamuHandle*/
 protected void CreateSVGElementForPounamuHandle( int[] gemoValues, String handleID, String role,
                  Element group){
    //write SVG element for a PounamuHandle and append it to group Element
  }

/*Create corresponding SVG element for PounamuMultiLinesInput*/
 protected void CreateSVGElementForPounamuMultiLines(PounamuMultiLinesInput comp, int xOffset,
            int yOffset,Element group){
     //write SVG element for a PounamuMultiLinesInput and append it to group Element
  }

/*Create corresponding SVG element for PounamuOneLineInput*/
 protected void CreateSVGElementForPounamuOneLine(PounamuOneLineInput comp, int xOffset, int
           yOffset, Element group){
    //write SVG element for a PounamuOneLineInput and append it to group Element
  }

/*Create corresponding SVG element for PounamuLabel*/
 protected void CreateSVGElementForPounamuLabel( PounamuLabel label, int xOffset, int yOffset,
            Element group){
    //write SVG element for a PounamuLabel and append it to group Element
 }

/*Create corresponding SVG element for PounamuButton*/
 protected void CreateSVGElementForPounamuLabel(PounamuButton button, int xOffset,  int yOffset,
           Element group){
    //write SVG element for a PounamuButton and append it to group Element
   }
} //end of class
```

Shape B' SVG Path
Canvas
Representation

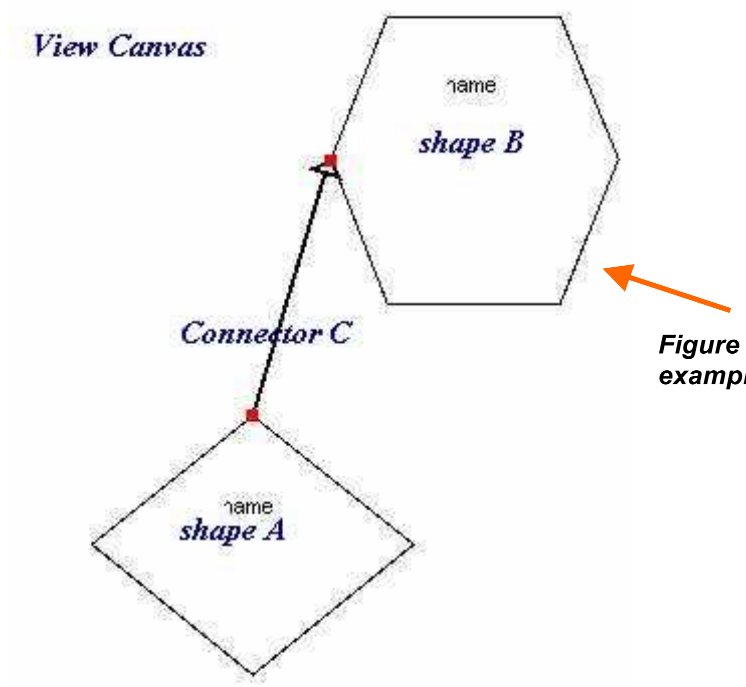


Figure 6-6(a) A PounamuView diagram example

```
<rect  x="0" y="0" width="800" height="800" style="fill: none; stroke:#000000;  stroke-
        width:1.0 " id="VT_TryVisualPro_1" pointer-events="none"
       onclick="locateCanvasPosition(evt)"/>
```

```
<clipPath id="ET_Try1_A_Clip"><polygon points="83,3 163,67 83,132 3,67"/>
</clipPath>
<clipPath id="ET_Try_B_Clip"><polygon points="31,3 117,3 146,74 117,146 31,146 3,74"/>
</clipPath>
```

```
<g   id="ET_Try1_A" transform="translate(40,227)"  modelType="ET_Try1"
      icontype="Rhomb" panelWidth="167" panelHeight="136" pointer-events="none"
      onclick="getShapeID(evt)" style="clip-path:url(#ET_Try1_A_Clip)">
<polygon  style="fill: none; stroke:#000000;  stroke-width:1.0; " points="83,3 163,67 83,132 3,67"
           basePanel="true"/>
<text  x="71" y="50" style="font-family:sansserif; font-size:11; fill:#000000;">name</text>
</g>

<rect  x="296" y="283" width="6" height="6" style="fill:none; stroke:white; stroke-width:0"
        id="Handle0_ET_Try1_A"  handle="true" role="general" cursor="default"
        onclick="locateShapeHandle(evt)" pointer-events="none"/>
……………………..other shape handles are skipped here
```

```
<g   id="ET_Try_B"   transform="translate(159,28)"   modelType="ET_Try"
      icontype="Hexagon" panelWidth="150" panelHeight="150" pointer-events="none"
      onclick="getShapeID(evt)" style="clip-path:url(#ET_Try_B_Clip)">
<polygon     style="fill: none; stroke:#000000;   stroke-width:1.0; " points="31,3 117,3 146,74 117,146 31,146
3,74" basePanel="true"/>
<text  x="63" y="39" style="font-family:sansserif; font-size:11; fill:#000000;">name</text>
</g>

<rect x="303" y="100" width="6" height="6" style="fill:none; stroke:white; stroke-width:0"
       id="Handle0_ET_Try_B" handle="true" role="general" cursor="default"
       onclick="locateShapeHandle(evt)" pointer-events="none"/>
……………………..other shape  handles are skipped here
```

Connector C SVG
Representation

```
<g  id="AT_TryConnector_C"  transform="translate(236,126)"  modelType="AT_TryConnector"
       startHandleID="Handle0_ET_Try1_A"   endHandleID="Handle3_ET_Try_B"
       pointer-events="none"  onclick="getShapeID(evt)">
<line  x1="50" y1="177" x2="86" y2="56" style="fill:#ffffff; stroke:#000000;  stroke-width:2
        body="true"/>
<polygon style="fill: none; stroke:#000000;  stroke-width:2.0;" points="80,54 93,58 89,50"/>
<text  x="65" y="167" style="font-family:sansserif; font-size:11; fill:#000000;"> </text>
<text  x="69" y="103" style="font-family:sansserif; font-size:11; fill:#000000;"> </text>
<text  x="74" y="40" style="font-family:sansserif; font-size:11; fill:#000000;"> </text>
</g>
```

Figure 6-6(b) SVG transformation of the PounamuView Diagram shown in 6-6(a)

Being able to construct an SVG image of a PounamuView diagram is only the first step towards dynamically displaying it on a web browser. The other tasks involve sending the generated SVG back to the web server and embedding it in a web page. Due to SVG's XML nature, transferring it via an RMI connection is an easy job. Again, as has been done in the GIF version tool, the web server caches the received SVG image in the client's session object for later use. It forwards the SVG code to the browser through the Servlet's *OutputStream* or *PrintWriter* object. Below are code samples for outputting the server-side SVG to a web browser:

```
/*This request handler is dispatched by ControllerServlet for SVG image output*/
public class GenerateSVG extends RequestHandler{
  public GenerateSVG() {
     /*Constructor*/
  }

  /*handleRequest method*/
  public void handleRequest(ServletContext context, HttpServletRequest req,
                                                    HttpServletResponse resp) throws ApplicationException{
        String svgImage=null;
       ApplicationSession appSession = RequestController.FindApplicationBean(req, this);

      //send servlet response header infomation
       resp.setHeader("Cache-Control","no-cache, no-store, must-revalidate");
       resp.setHeader("Cache-Control","post-check=0, pre-check=0");
       String agent = req.getHeader("User-Agent").toLowerCase();
       resp.setContentType( "image/svg+xml" );

       try{
         //Get a PrintWriter object from Servlet response
         PrintWriter out=resp.getWriter();

         //Call remote Server function to create SVG image for the specified view
         RMIServerInterface obj = (RMIServerInterface)context.getAttribute("RMIRemoteServer");
         String svgImage = obj.generateSVGTemplate(appSession.getCurrentProject(),
                              appSession.getCurrentViewType(), appSession.getCurrentPounamuView());

         //When svgImage is not null, send it to web browser via Servlet's PrintWriter object
          if (svgImage!= null) {
                out.write(svgImage);
                out.flush();
                out.close();
          }
         else
```

```
            throw new ApplicationException("The PounamuView you asked can’t be created");
      }catch(RemoteException e){
        throw new ApplicationException(e.toString());
      }catch(IOException e){
        throw new ApplicationException(e.toString());
      }
    }
 }
```

Finally, we need to embed a reference to the above output Servlet in a JSP view where an SVG image should appear, and this is done with the following scriptlets and HTML tags:

```
<% String svgSrc="/PounamuSVGApp/controllerservlet?action=GenerateSVG; %>
<embed name= “SVGObject”  width="800" height="800" src=<%=svgSrc%>
          type="image/svg+xml">
</embed>
```

6.2.2 Capturing and Interpreting SVG Events

Another difficulty that needs to be addressed in implementing this SVG version of the Pounamu/Thin tool is how to translate SVG events (e.g. *mouseover*, *mouseclick*, and some keyboard events) into system recognizable editing commands.

As stated in Chapter 4, three SVG event handlers, including *getShapeID*, *locateCanvasLocation* and *locateShapeHandle*, are enough to implement all system functionalities. Also, from the SVG image construction example shown in Figure 6-6 (b), you will have a general idea of what events these handlers are registered with. Here, a summary is drawn as below:

- *GetShapeID*: Being associated with click events on a PounamuIcon’s (may be shape or connector).
- *LocateCanvasLocation*: Being associated with click events on a PounamuView canvas, where the canvas is represented with an SVG *rect* element having a certain ID value.
- *LocateShapeHandle*: Being associated with click events on a shape handle, where shape handles are red small rectangles used to highlight the shape’s selected state.

We certainly can write client-side scripting for the above event handlers so that they are executed at runtime without the web server’s interference. However, for the sake of our thin-client computing scenario (where most computing is done on the server side) that we are trying to achieve in this project, the client shouldn’t be burdened too much. Fortunately, both Adobe and batik SVG implementations provide server-connection facilities so that SVG events captured on the client-side can be propagated to the web server. In the following paragraphs, we use Adobe’s *getURL()* as an example to demonstrate how this can be done.

getURL() is a function that makes a request for data to a given URL location. It is very useful for generating dynamic SVGs that need updates of data based on user interaction/input while calculations may be too complex to be handled with the client-side scripting. It's syntax, according to Peter Sorotokin from Adobe, is as follows [78]:

getURL(URL, callback)
URL: source to request data from, for security reasons, must be on the same server as SVG document
callback: a function which processes reply status and the returned data contents from URL

When *getURL()* is used for the purpose of event propagation, event-related information can be appended to the URL in the form of request parameter name-value pairs. For example, in our *getShapeID()* event handler, the ID and type of the shape which has been clicked are appended to the requested URL so that they can be sent back to the web server. After receiving the event information, the application logic on the web server decides the next JSP page to be displayed and sends the name of the page back. Then, *Callback* scripts on the client side retrieve the page name and sequentially connect it. To make the above explanation much clearer, a sequence diagram (shown in Figure 6-7) is used to describe objects involved in this *getShapeID* operation and messages exchanged between these objects. Following that is a code sample for *getShapeID()*.

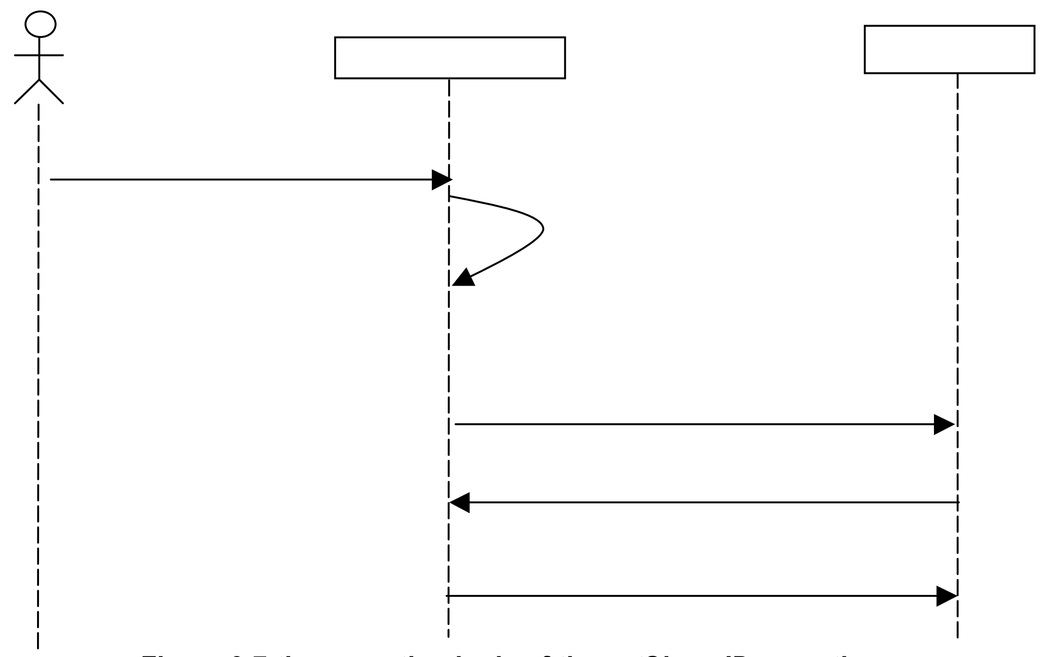

User
Web Browser
Web Server
1. Click a Pounamu Shape or Connector
2. ID and X,Y values of the clicked icon is retrieved
3. Connect to a designated URL (with the retrieved information appended) via the SVG built-in getURL()
4. The URL of the next page is sent back
5. Connect the returned next URL in the Callback function of getURL()

Figure 6-7 the execution logic of the getShapeID operation

Here is a code sample for getShapeID() function:

```
function getShapeID(evt){
  //get event target object
  var target=evt.getTarget();
  while (target && !target.getAttribute('id'))
        target = target.getParentNode();

 //retrieve events related information from SVG DOM
  var shapeIDName=target.getAttribute("id");
```

```
   var modelType=target.getAttribute("elementType");

   //append events related information to URL with name-value pairs
    var src="/PounamuSVGApp/controllerservlet?action=getShapeID&shapeID="
                  +shapeIDName+"& elementType ="+modelType;

    //connect to server with the obtained URL
     getURL(src,getShapeIDCallback);
}

//**getShapeID callback function**//
function getShapeIDCallback(data){
    //retrieves the next JSP page the application should go from the received data content
   if(data.content){
      var ref="/PounamuSVGApp"+data.content;
     window.location.href =ref;
   }
}
```

Having understood the approach of propagating SVG events to the web server, we now explain how these events are related to the application-specific diagram editing commands. It is reasonable to imagine that an editing command (such as move entity, add entity, add association, resize entity etc.) is composed of a series of SVG events. For example, to move an entity shape in a PounamuView diagram, a shape click event is initiated first so that the target shape is identified and highlighted through the execution of the *getShapeID* handler; and then by triggering a canvas click event, the preferred new location for the target is specified as a result of executing the *locateCanvasPostion* handler. For other editing scenarios, there are similar sequences, although each editing command can be decomposed into a different series of SVG events.

6.2.3 Implementation of Multiple Editing

Detailed knowledge of what multiple editing is and why it is wanted can be obtained from Section 4.3.3.2. There you can also find reasons why multiple editing can be done with the SVG version of Pounamu/Thin, but not with the GIF version. To summarize, multiple editing is an alternative editing mode where users can issue and buffer multiple edit commands on their own copy of SVG diagram, and the list of buffered edits is sent to and committed on the Pounamu server later. This edit buffering facility brings two main advantages to our thin-client diagramming tool. First, system performance is improved because the number of communication trips involved to complete certain tasks is reduced. Second, a set of edits by each user can be made transactionally, i.e., without knowledge of another set of user-buttered edits. Although useful, multiple editing can only be implemented in this SVG version prototype (not the GIF one). The key reason is that the web component-stored SVG XML data is manipulable while GIF binary data is not.

We now briefly discuss how multiple editing is implemented, with discussion focusing on five steps involved in performing a multiple editing task on a PounamuView diagram.

Step 1: An SVG image of the target PounamuView diagram should be generated and cached on the web server. Additionally, this SVG image needs to be displayed on a web browser for viewing and interaction. Detailed information about how to generate and display an SVG-based diagram can be seen in Section 6.1.

Step 2: In multiple editing, edit actions (such as Add/Move/Resize Shapes, Set Properties, etc.) are initiated against the cached SVG diagram. Thus, a facility for manipulating the cached SVG needs to be provided. The *ManipulateSVGDoc()* method of the *ApplicationSession* class serves such a purpose. In this method, a series of SVG events (e.g. *GetShapeID*, *locateCanvasPostion*, and *locateShapeHandle*) required for the currently activated edit action is set up accordingly. Below is a code sample for this function:

```
public class ApplicationSession implements Serializable, HttpSessionBindingListener {

   /*
    *A method used to parse the cached SVG image String and manipulate the parsed
   *SVG DOM tree
   */
   public void manipulateSVGDoc(){
      try{
         //Parse the cached SVG image String into a DOM tree
          ByteArrayInputStream instream = new ByteArrayInputStream(svgDocString.getBytes());
          ByteArrayOutputStream outstream=new ByteArrayOutputStream();
          DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
          DocumentBuilder docBuilder = factory.newDocumentBuilder();
          Document doc = docBuilder.parse(instream);    doc.normalize();
          //set up the selected Pounamu shape/connector
          setSelectedPounamuIcon(doc);
          //set up the required SVG event
          setEventHandler(doc);
          //Serialize SVG DOM tree back to the cached string to be sent via the output Servlet
          XMLSerializer serializer = new XMLSerializer(outstream, new OutputFormat(doc,"UTF-8",false));
          serializer.serialize(doc);     setSVGDoc(outstream.toString());
        this.SVGCachedManipulation=false;
      }catch (SAXException sxe) {
          x.printStackTrace();
      }catch (ParserConfigurationException pce) {
          pce.printStackTrace();
      }catch(IOException ioe){
           ioe.printStackTrace();
      }catch (FactoryConfigurationError e) {
          System.out.println("Could not locate a factory class");
      }
   }

  /*A method used to set the selected Pounamu Icon in the cached SVG image*/
   protected void setSelectedPounamuIcon(Document doc){
        //function body omitted
   }

  /*A method used to set the current SVG event in the cached SVG image*/
    protected void setEventHandler(Document doc){
        //function body omitted
  }
  /*Rest of the object methods*/
}
```

Step 3: The edit-action related information is collected and stored into an *ActionData* object of a specific sub type. In order to show what edit action has been made on the cached SVG, *drawSketch()* of this *ActionData* object is called to overlay edit sketches on the cached diagram. A code sample for *drawSketch()* of the "add shape" *ActionData* is displayed as an example:

```
public class AddEntityActionData implements ActionData {

   /*
   *A method used to overlay sketches representing this edit ActionData onto the cached *SVG image
   */
   public void drawSketch (Document rootDoc){
          //draw a dotted-outline rectangle to represent the will-be-added entity shape
           int rectWidth=160;  int rectHeight=240;
           SVGShapesWriter writer=new SVGShapesWriter(rootDoc,appObject);
           Element newgroup=writer.drawGroup(null,shapeX,shapeY,writer.getRoot());
           String style="fill:none; stroke:darkgreen; stroke-width:2;stroke-dasharray:5,1";
           Element rect=writer.drawRect(0,0,rectWidth,rectHeight,style,newgroup);
           Font stringFont=new Font("Arial",Font.PLAIN,11);
           writer.setFont(stringFont);
           FontMetrics fm=writer.getFontMetrics();
           String shapeLabel=this.modelCompName+" created";
           int h = fm.getHeight();   int w = fm.stringWidth(shapeLabel);
           int stringX=rectWidth/2-w/2;  int stringY=rectHeight/2;
           writer.drawString(shapeLabel,stringX,stringY,newgroup);
           sketchElement=(Element)((Node)newgroup.cloneNode(true));
   }
   /*Rest of the code*/
}
```

Step 4: The *ActionData* set up in Step 3 is inserted into the buffered action list contained in the user's session object.

Step 5: After all the preferred edits are made and buffered by repeating Step2~Step4, users can choose to submit, modify or clear this list of buffered edits. Upon submitting, all the buffered edits are sent to the Pounamu server and committed there. Upon modifying, sketches for those unwanted edits are removed by calling the *removeSketch()* function of the corresponding edit *ActionData*. To show what the *removeSketch()* function looks like, a code sample for *removeSketch()* of the "add shape" *ActionData* is presented below:

```
public class AddEntityActionData implements ActionData {

  /*
   *A method used to remove sketches representing this edit ActionData object from the *cached SVG
   *image due to cancellation
  */
  public void removeSketch (Document rootDoc){
         SVGShapesWriter writer=new SVGShapesWriter(rootDoc,this.appObject);
         StringBuffer sketchStringBuffer=new StringBuffer(10000);
         writer.printElememtStringRepresentation(sketchStringBuffer,sketchElement);
         String sketchString=sketchStringBuffer.toString();
         NodeList nl=rootDoc.getElementsByTagName("g");
         int j=0;
         while(nl.item(j)!=null){
               Node n=nl.item(j);
               if(n.getNodeType()==Node.ELEMENT_NODE){
                      StringBuffer tempStringBuffer=new StringBuffer(10000);
                     writer.printElememtStringRepresentation(tempStringBuffer,(Element)n);
```

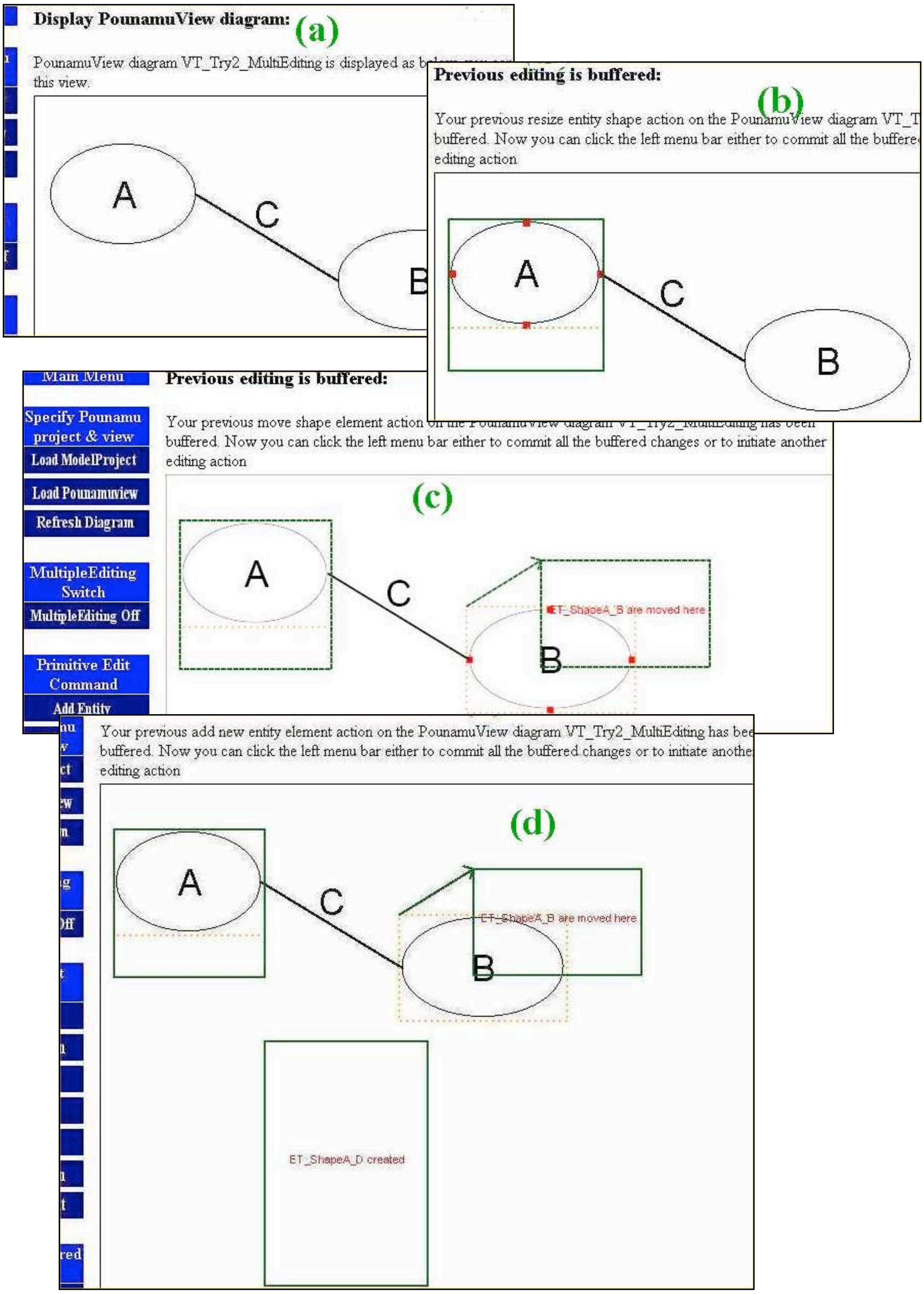
```
                    String tempString=tempStringBuffer.toString();
                    if(tempString.equals(sketchString)){
                       rootDoc.getDocumentElement().removeChild(n);
                       break;
                    }
                 }
                  j++;
            }
      }
      /*Rest of the code*/
}
```

6.3 Screen Dumps of the SVG-based thin-client user interface

The overall purpose of this section is to reveal the basic user interfaces of this SVG version thin-client tool, as well as the user's interaction with it. Also, as we have done for the previous GIF version prototyping system, screen dumps of the system running are employed to describe how the main editing functionalities can be performed via this SVG version thin-client user interface. For comparison, the same example, an OOA class diagram modelled for the Video Store system, is chosen as the test bed for these editing operations. The detailed editing scenarios are presented in the following subsections, with one for each specific editing action. However, before describing the individual behaviour of editing operations, it is necessary to restate the inherent difference between the two versions of thin-client Pounamu user interface (GIF and SVG). While only single editing is allowed in the former, multiple editing concept is available in the latter. Below we use a simple example to demonstrate how users can use this multiple editing facility.

Multiple Editing Example

Again, we emphasize that this example is specifically designed to demonstrate how to perform multiple editing via the SVG version thin-client user interface. Multiple editing can be switched off by clicking the left menu button if you prefer to do things in the other way (i.e. single editing). Figure 6-8(a) shows a test diagram which is composed of shapes A, B and connector C. Three editing tasks need to be executed on this diagram, including Resizing A, Moving B and Adding D (another entity shape). Since in the multiple editing mode, any edit that users make is buffered first, you only see sketches drawn for each edit displayed in three following figures such as 6-8(b), (c), (d). Now you can make a choice between committing and modifying these buffered edits. When choosing the former, you are provided with an updated diagram as shown in Figure 6-9(a) with all three edits committed. Otherwise, if you choose the latter, a web page (shown in 6-9(b)) with two frames appears. The left frame of this page displays a list of buffered edits, while the right frame displays the view diagram with edit sketches overlaid. You can make deletions to unwanted edits via the left frame, and the modification is reflected in the right frame. For example, Figure 6-9(c) describes the system's response to the deletion of the "Adding D" edit made before.



Display PounamuView diagram:

PounamuView diagram VT_Try2_MultiEditing is displayed as b
this view.

Previous editing is buffered:

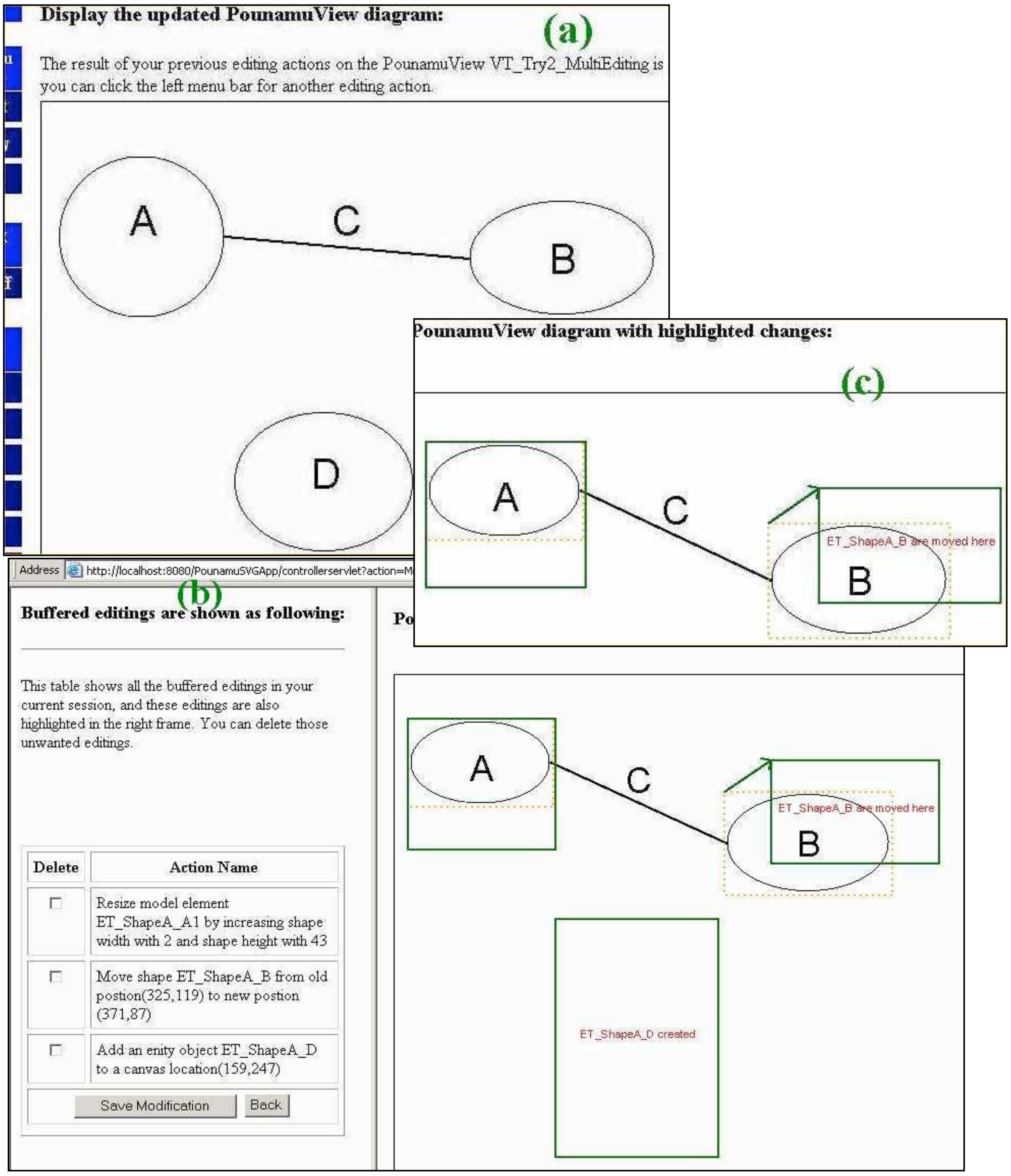
Your previous resize entity shape action on the PounamuView diagram VT_T
buffered. Now you can click the left menu bar either to commit all the buffere
editing action

Previous editing is buffered:

Your previous move shape element action
buffered. Now you can click the left menu bar either to commit all the buffered changes or to initiate another editing action

Your previous add new entity element action on the PounamuView diagram VT_Try2_MultiEditing has bee
buffered. Now you can click the left menu bar either to commit all the buffered changes or to initiate anothe
editing action

Figure 6-8 Intermediate result of buffering three edits on a PounamuView Diagram


(a)
(b)
(c)
A
B
C
D
ET_ShapeA_B are moved here
ET_ShapeA_D created
Address http://localhost:8080/PounamuSVGApp/controllerservlet?action=M
Save Modification
Back

Display the updated PounamuView diagram:

The result of your previous editing actions on the PounamuView VT_Try2_MultiEditing is you can click the left menu bar for another editing action.

PounamuView diagram with highlighted changes:

Buffered editings are shown as following:

This table shows all the buffered editings in your current session, and these editings are also highlighted in the right frame. You can delete those unwanted editings.

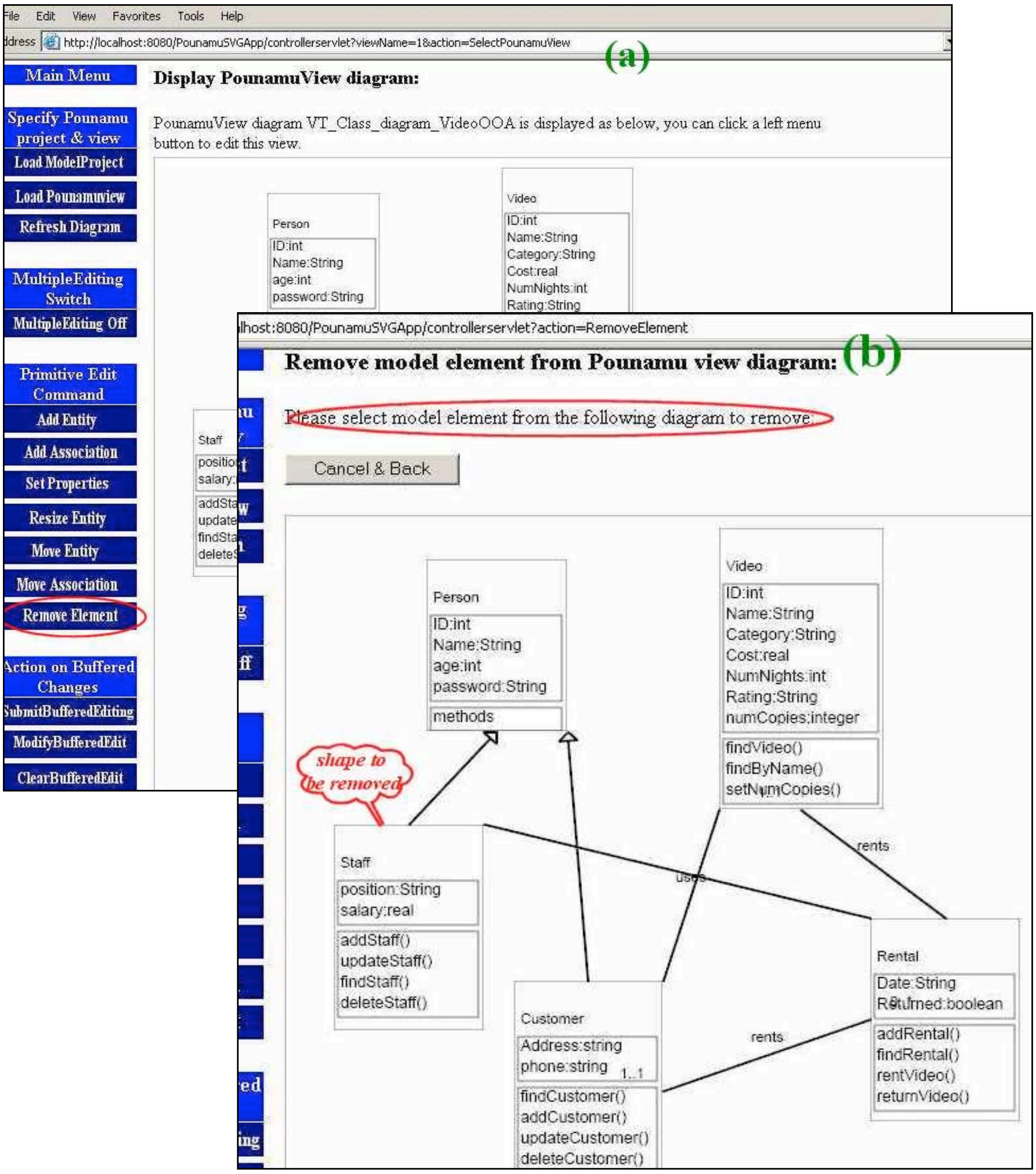
Delete	Action Name
☐	Resize model element ET_ShapeA_A1 by increasing shape width with 2 and shape height with 43
☐	Move shape ET_ShapeA_B from old postion(325,119) to new postion (371,87)
☐	Add an enity object ET_ShapeA_D to a canvas location(159,247)

Figure 6-9 Examples of Committing (a) and Modifying (b)(c) Three Buffered Edits

6.3.1 Example of Removing an Entity Shape

To demonstrate the functionalities supported in the SVG version thin-client diagramming tool, we first remove an entity shape and an association connector from the original video store OOA class diagram (shown in Figure 5-2(d)), and then add them back later. So in this and the following subsection, we describe the two removal actions first.

Figure 6-10 demonstrates how to remove an entity shape from a PounamuView diagram. First, the user needs to click the “Remove Element” left button (a). A page (b) appears prompting the user to select the intended shape for removal (b), and the target is highlighted with red dash-dotted contour sketches shown in (c). The next action that needs to be taken is either “Buffer Change” or “Cancel Change”. With buffering, the removing editing is saved for later execution (d), while with cancelling, the diagram is recovered to its original appearance (e).

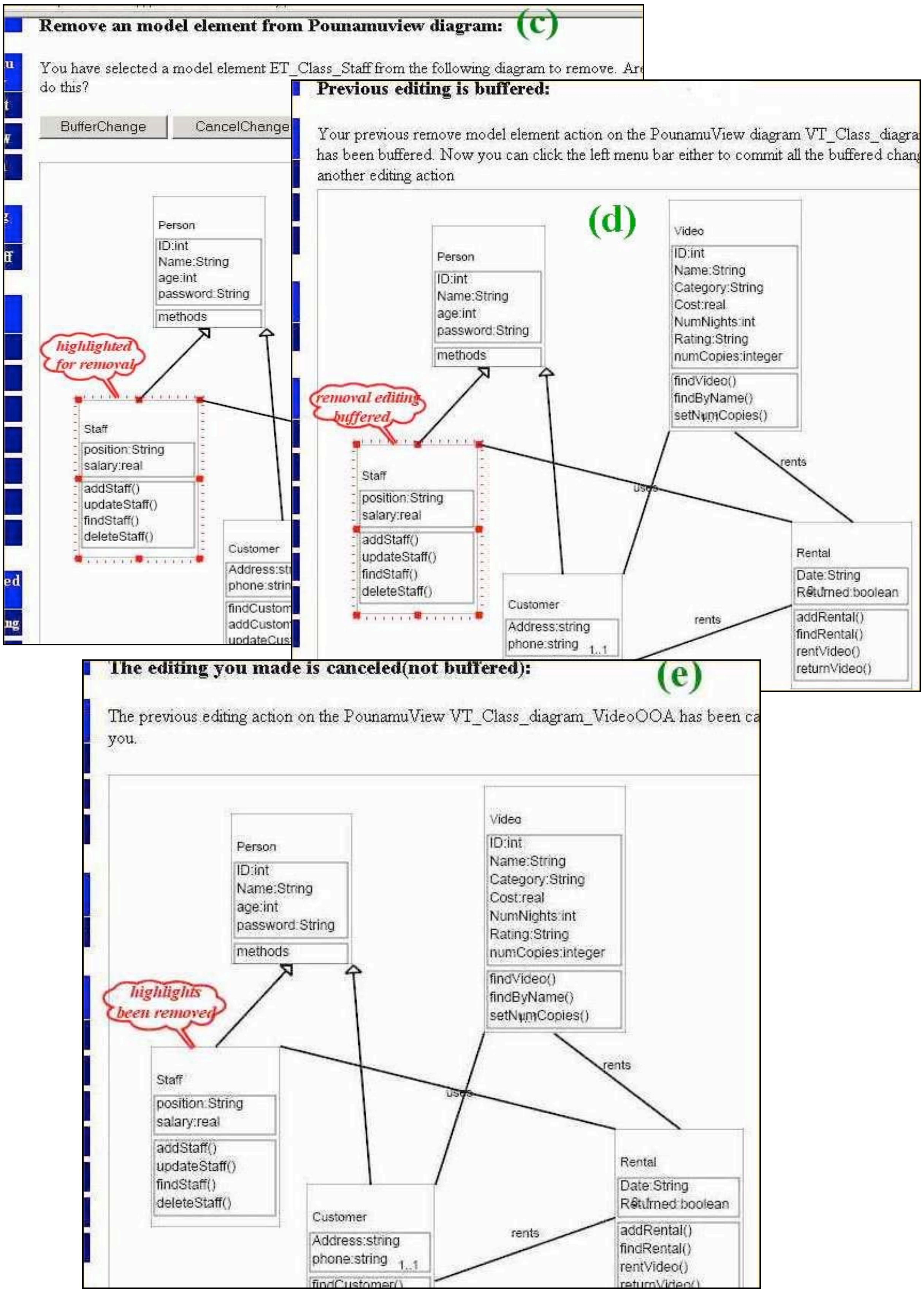


Display PounamuView diagram:

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

Remove model element from Pounamu view diagram:

Please select model element from the following diagram to remove



Remove an model element from Pounamuview diagram:

You have selected a model element ET_Class_Staff from the following diagram to remove. Ar
do this?

Previous editing is buffered:

Your previous remove model element action on the PounamuView diagram VT_Class_diagra
has been buffered. Now you can click the left menu bar either to commit all the buffered chan
another editing action

The editing you made is canceled(not buffered):

The previous editing action on the PounamuView VT_Class_diagram_VideoOOA has been ca
you.

Figure 6-10 SVG version "Remove Entity Shape" Running Examples

6.3.2 Example of Removing an Association Connector

This editing operation is similar to the previous one. The only exception is that an association connector is selected for removal, rather than an entity shape. Figure 6-11 shows procedures involved in deleting an association connector from the diagram, where (a), (b), (c) represent the results of pressing “Remove Element” menu button, clicking the target connector and buffering this connector removing action, respectively.

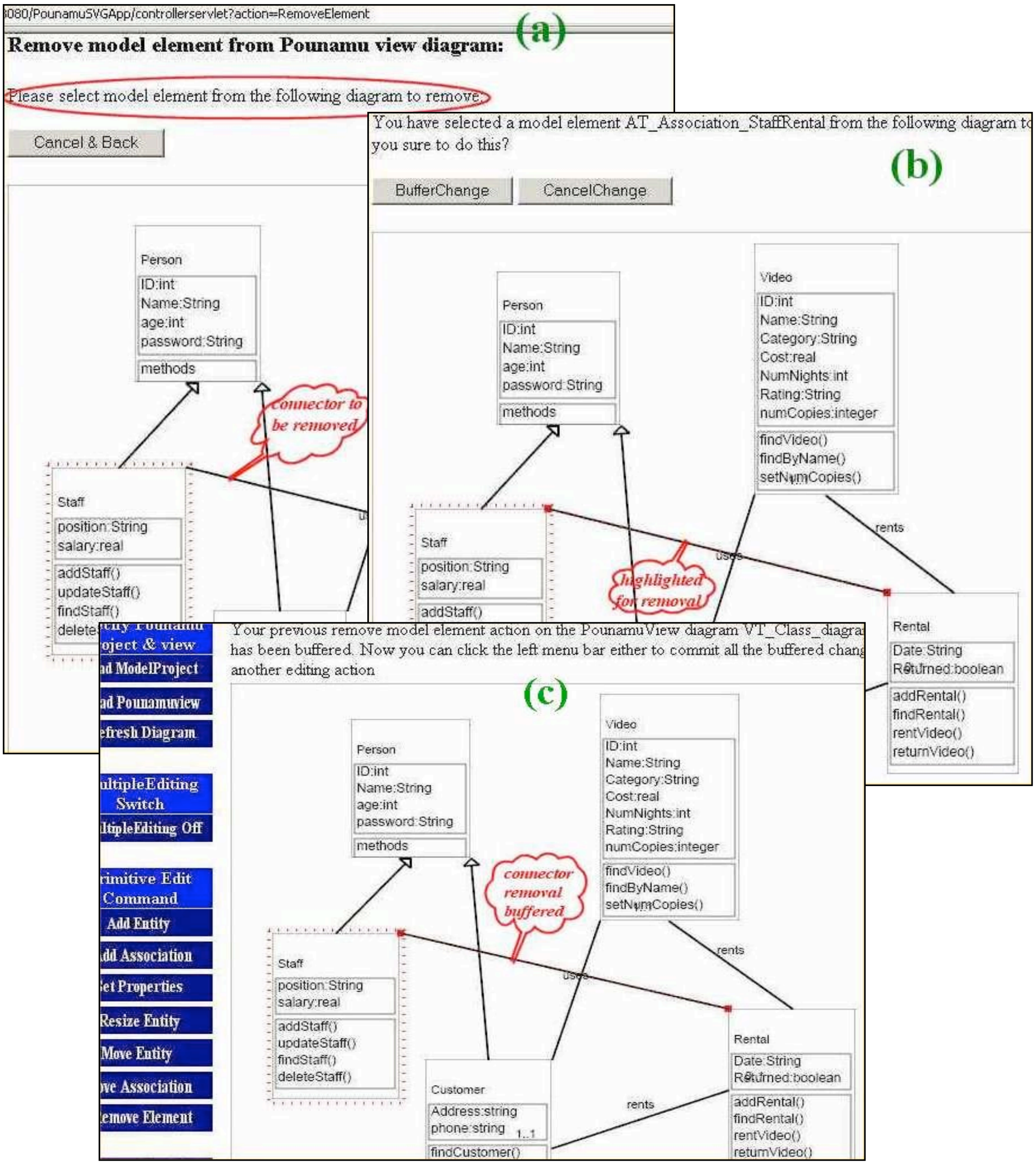
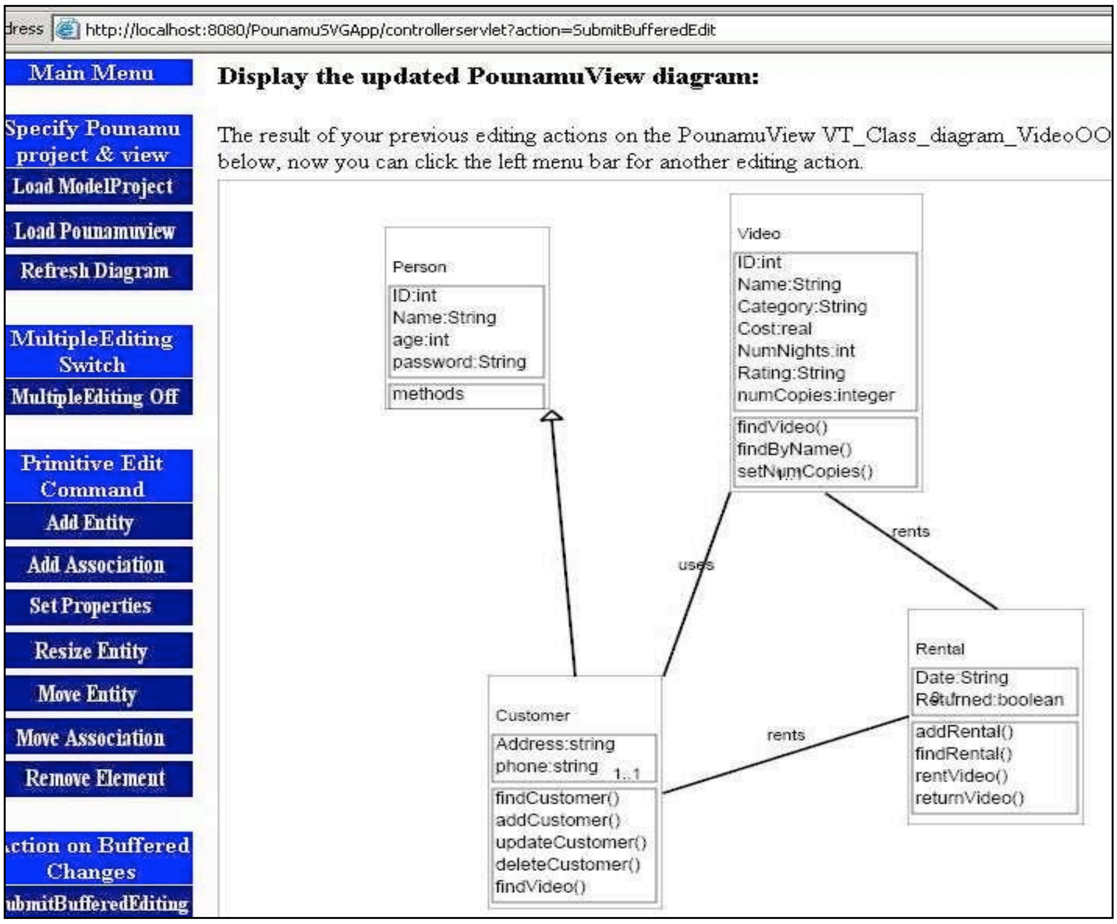


Figure 6-11 SVG version “Remove Association Connector” Running Examples

6.3.3 Example of Adding an Entity Shape

After the previous removal editings are committed, the OOA diagram now looks like Figure 6-12. In this subsection and the one following this, we demonstrate how to add back the deleted shape and connector. Figure 6-13 illustrates the steps involved in "adding an entity shape" action. In order to initiate this action, the user needs to click the "Add Entity" menu item. A new page (a) appears which prompts the user to enter shape parameters (name, type). After that, the system requires the user to specify a position for the new shape with a click on the diagram canvas (b). In response to this click, rectangle sketches appear at the designated position (c). When the user confirms to commit this action, the editing result is provided in (d).

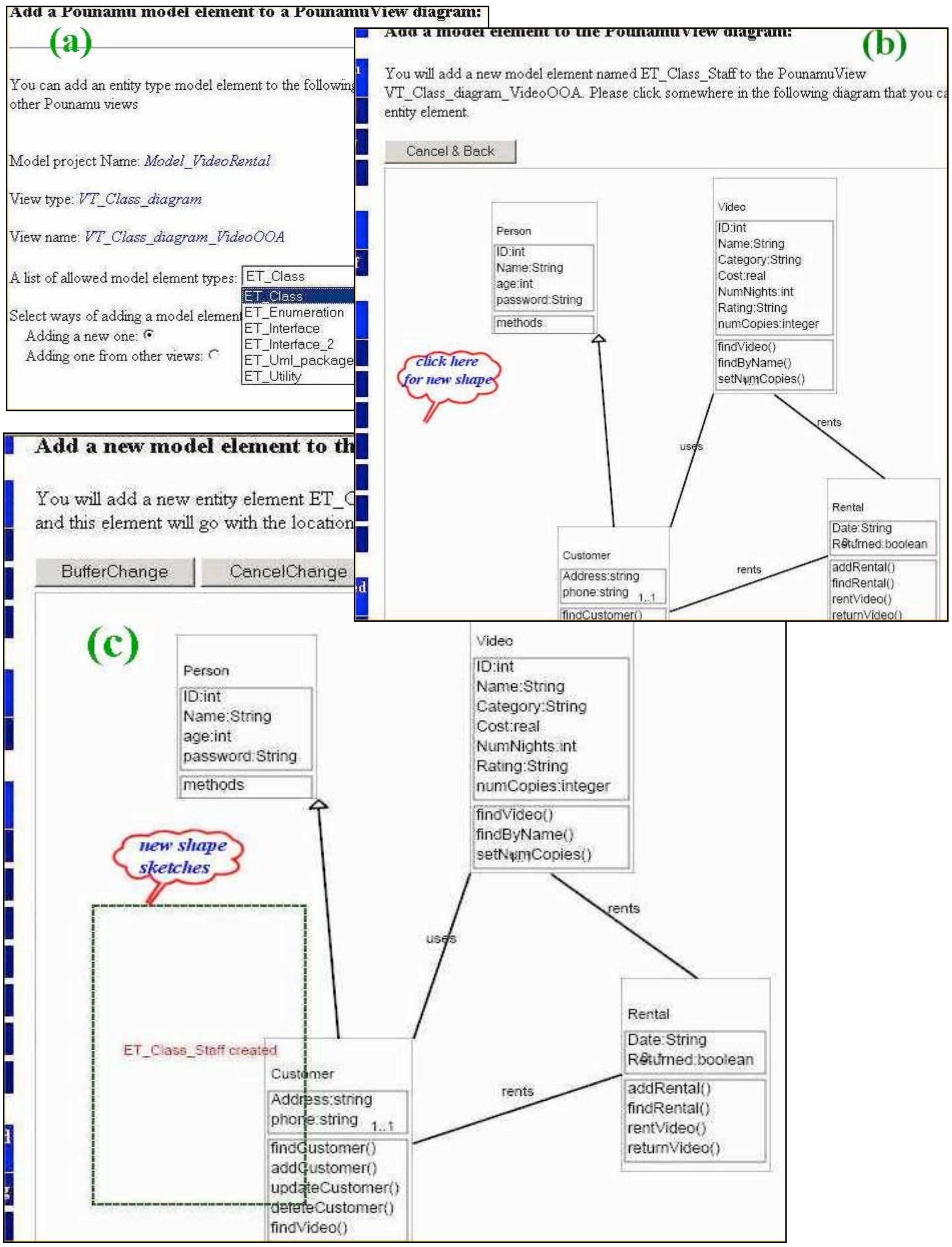


dress http://localhost:8080/PounamuSVGApp/controllerservlet?action=SubmitBufferedEdit

Display the updated PounamuView diagram:

The result of your previous editing actions on the PounamuView VT_Class_diagram_VideoOO below, now you can click the left menu bar for another editing action.

Figure 6-12 the Updated Diagram Showing Previous Removing Edits Executed



Add a Pounamu model element to a PounamuView diagram:

You can add an entity type model element to the following other Pounamu views

Model project Name: *Model_VideoRental*

View type: *VT_Class_diagram*

View name: *VT_Class_diagram_VideoOOA*

A list of allowed model element types: ET_Class

Select ways of adding a model element

Adding a new one: ◉

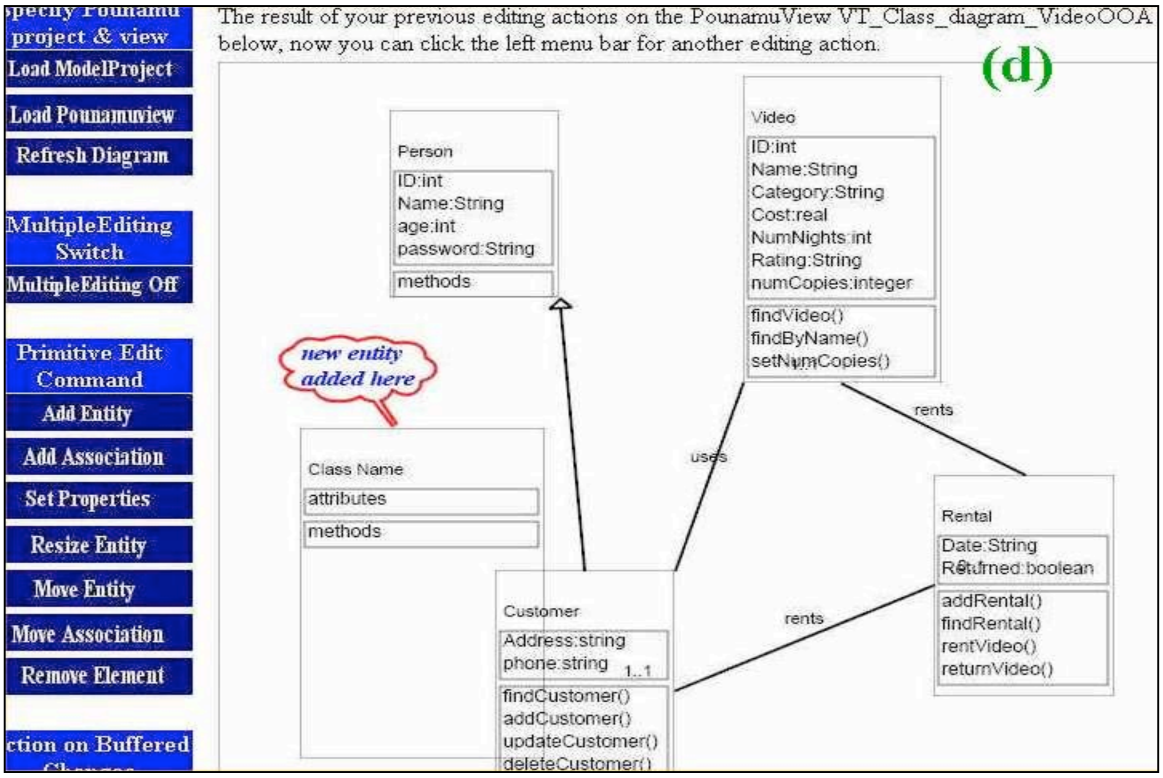
Adding one from other views: ○

Add a model element to the PounamuView diagram:

You will add a new model element named ET_Class_Staff to the PounamuView VT_Class_diagram_VideoOOA. Please click somewhere in the following diagram that you ca entity element.

Add a new model element to th

You will add a new entity element ET_C and this element will go with the location



The result of your previous editing actions on the PounamuView VT_Class_diagram_VideoOOA below, now you can click the left menu bar for another editing action.

Person
ID:int Name:String age:int password:String
methods

Video
ID:int Name:String Category:String Cost:real NumNights:int Rating:String numCopies:integer

findVideo()
findByName()
setNumCopies()

Class Name
attributes
methods

Customer
Address:string phone:string

findCustomer()
addCustomer()
updateCustomer()
deleteCustomer()

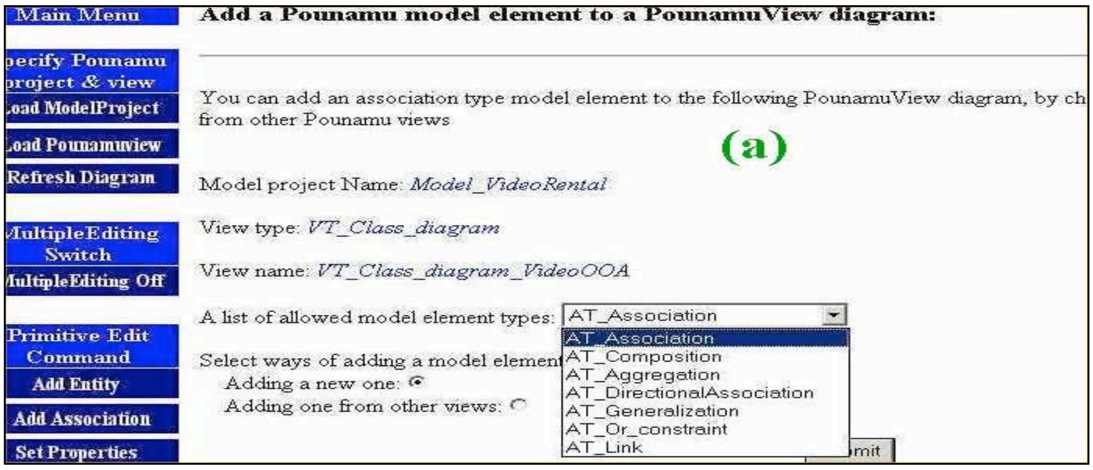
Rental
Date:String Returned:boolean

addRental()
findRental()
rentVideo()
returnVideo()

Figure 6-13 SVG Version "Add Entity Shape" Running Examples

6.3.4 Example of Adding an Association Connector

This editing action commences by clicking the "Add Association" button on the left menu bar. Similar to the above adding entity shape, a web page shown in Figure 6-14(a) appears for the user to input action parameters such as the connector name, type etc. Next, the system requires the user to specify the start and end handles for this connector (b). This can be completed by clicking the appropriate shape handles (represented in the form of red rectangles surrounding an entity shape). As a result, a dotted line connecting those two handles is drawn on the diagram to represent a simplified connector (c). Finally, as with other types of editing actions, the user can either buffer or cancel this editing depending on the situation. The result of committing this editing is shown in (d).



Add a Pounamu model element to a PounamuView diagram:

You can add an association type model element to the following PounamuView diagram, by ch from other Pounamu views

Model project Name: *Model_VideoRental*

View type: *VT_Class_diagram*

View name: *VT_Class_diagram_VideoOOA*

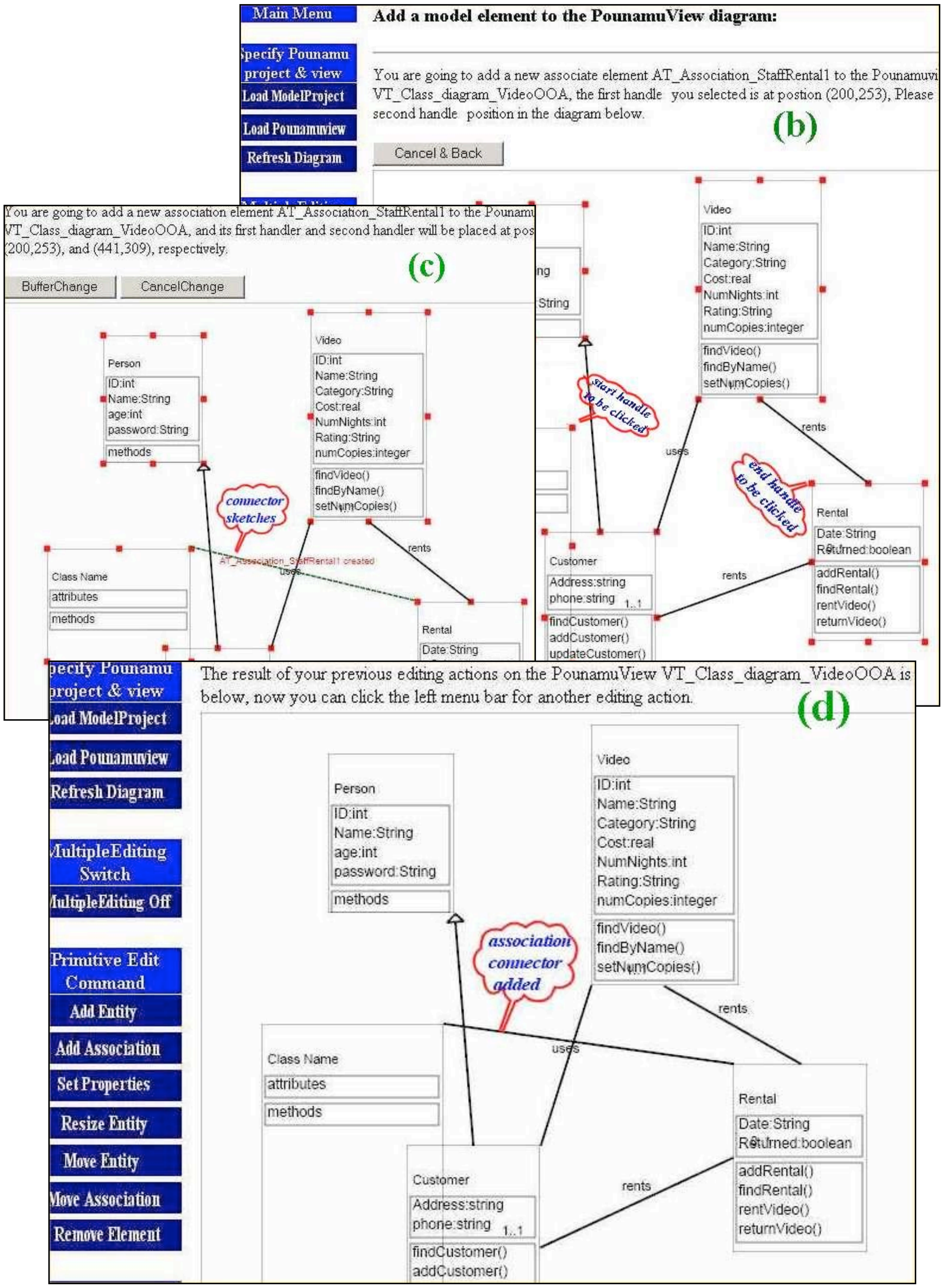
A list of allowed model element types: AT_Association

- AT_Association
- AT_Composition
- AT_Aggregation
- AT_DirectionalAssociation
- AT_Generalization
- AT_Or_constraint
- AT_Link

Select ways of adding a model element

Adding a new one: [✓]

Adding one from other views: ○


Add a model element to the PounamuView diagram:

You are going to add a new associate element AT_Association_StaffRental1 to the Pounamuvi VT_Class_diagram_VideoOOA, the first handle you selected is at postion (200,253), Please second handle position in the diagram below.

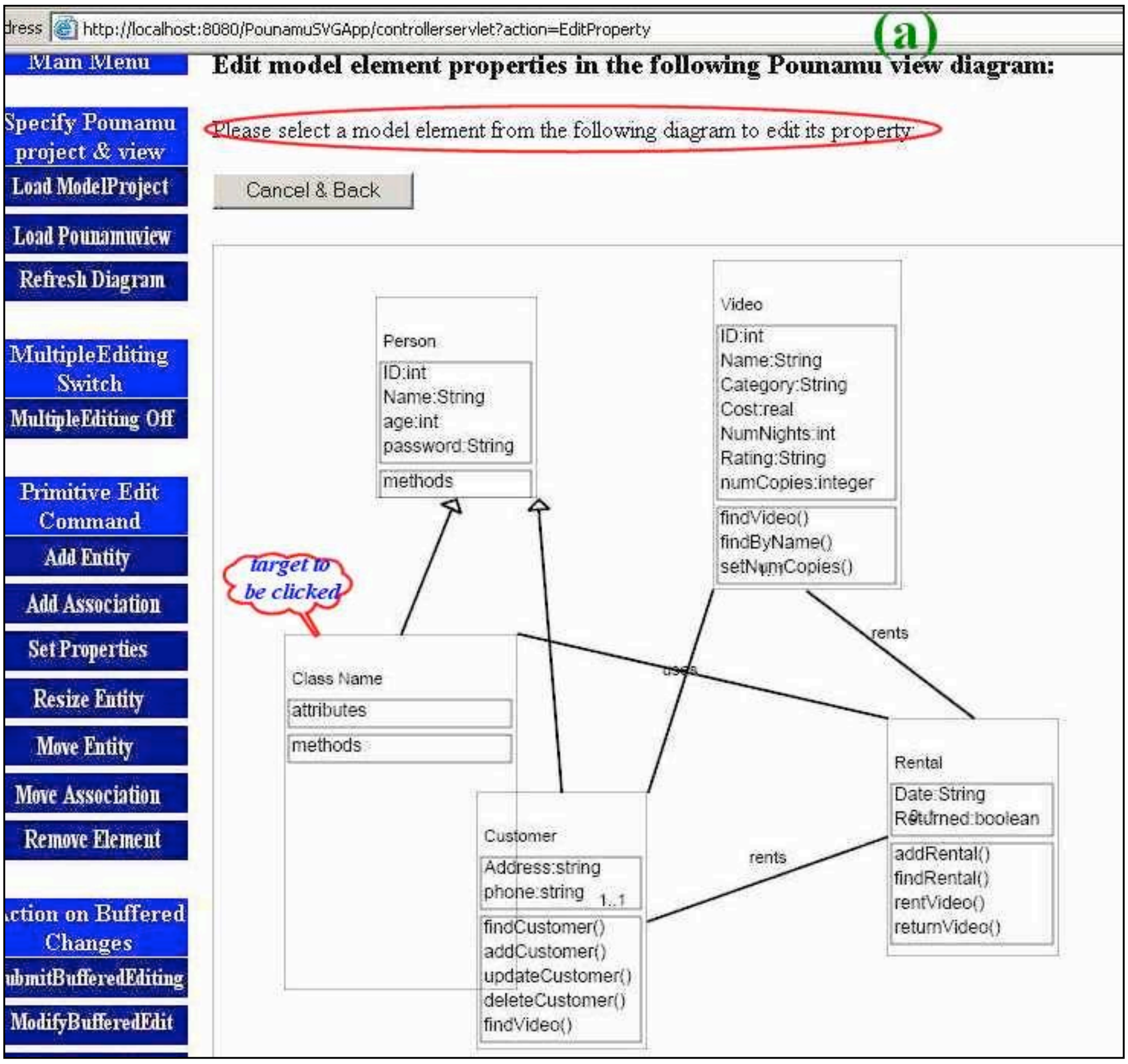
You are going to add a new association element AT_Association_StaffRental1 to the Pounamu VT_Class_diagram_VideoOOA, and its first handler and second handler will be placed at pos (200,253), and (441,309), respectively.

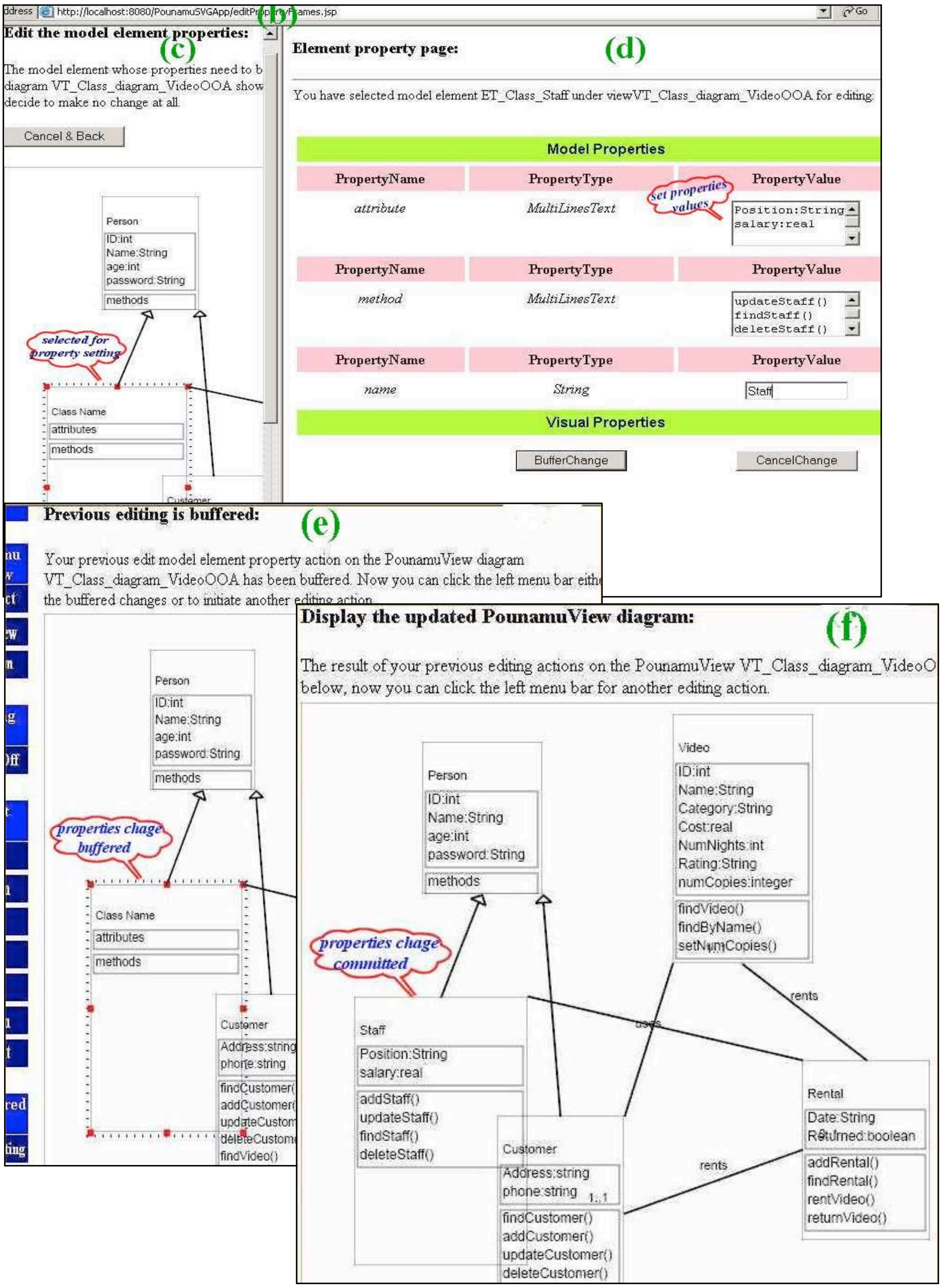
The result of your previous editing actions on the PounamuView VT_Class_diagram_VideoOOA is below, now you can click the left menu bar for another editing action.

Figure 6-14 SVG Version "Add Association Connector" Running Examples

6.3.5 Property Setting Example

This version's property editing is very similar to the GIF version. For this reason, a detailed description of this operation is skipped, and only Figures 6-15(a)~(f) are given to show execution procedures. In (a), the user needs to select the target object. The user's selection is highlighted by red rectangle handles in the left frame (c) of the web page (b), while the target's properties (name, type and value) are listed in the right frame (d). The user can enter new property values in (d). However, unlike in the GIF version of the Pounamu/Thin tool where property setting is performed immediately after new values have been set, property setting in this SVG prototype can be buffered until the user decide to submit all edits stored in the session. Due to this fact, in Figure 6-15(e) only blue dash-dotted contour sketches can be seen overlaid on the original shape to show that its properties are anticipated to change. The final execution result of this buffered property setting is shown in Figure 6-15(f).





Edit the model element properties:

The model element whose properties need to b
diagram VT_Class_diagram_VideoOOA show
decide to make no change at all.

Element property page:

You have selected model element ET_Class_Staff under viewVT_Class_diagram_VideoOOA for editing:

Model Properties		
PropertyName	PropertyType	PropertyValue
attribute	*MultiLinesText*	Position:String salary:real
PropertyName	PropertyType	PropertyValue
method	*MultiLinesText*	updateStaff() findStaff() deleteStaff()
PropertyName	PropertyType	PropertyValue
name	*String*	Staff
Visual Properties		

Previous editing is buffered:

Your previous edit model element property action on the PounamuView diagram VT_Class_diagram_VideoOOA has been buffered. Now you can click the left menu bar eith
the buffered changes or to initiate another editing action.

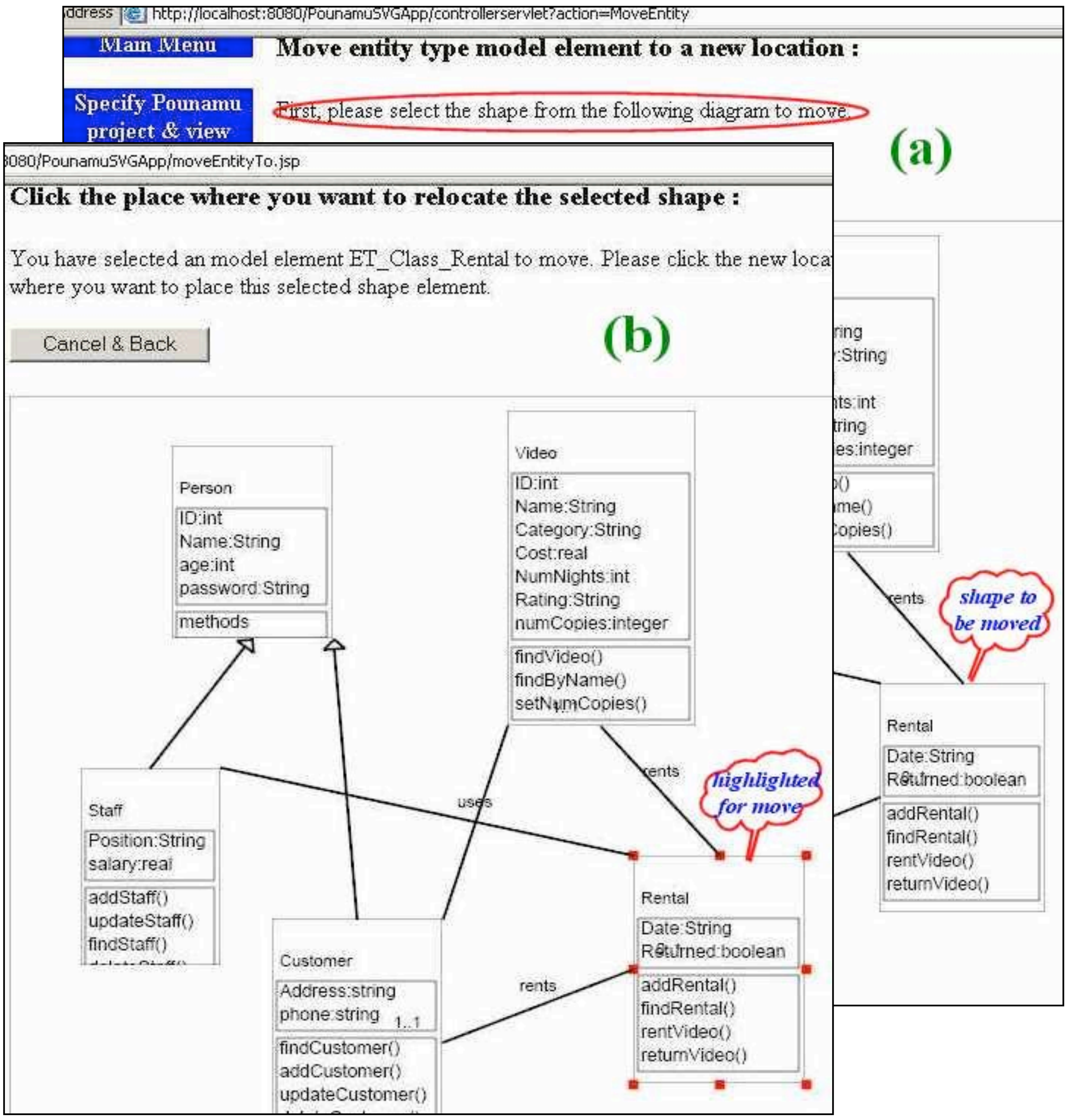
Display the updated PounamuView diagram:

The result of your previous editing actions on the PounamuView VT_Class_diagram_VideoO
below, now you can click the left menu bar for another editing action.

Figure 6-15 SVG Version "Set Properties" Running Examples

6.3.6 Example of Moving an Entity Shape

The whole process of the moving shape action is shown as in Figure 6-16. This is initiated by clicking the "Move Shape" left menu item. After that, the user needs to click the intended target shape on the page (a), and the user's selection is highlighted in its usual way (b). Then, the system reminds you to specify a new location for the to-be-moved shape. In response to the resulting canvas click event, a green dash-dotted rectangle representing a simplified moving result is drawn on the diagram (c). Again, the user is faced with the choice of buffering or cancelling this editing action. Committing this buffered moving action brings in a page looking like (d).


ddress http://localhost:8080/PounamuSVGApp/controllerservlet?action=MoveEntity

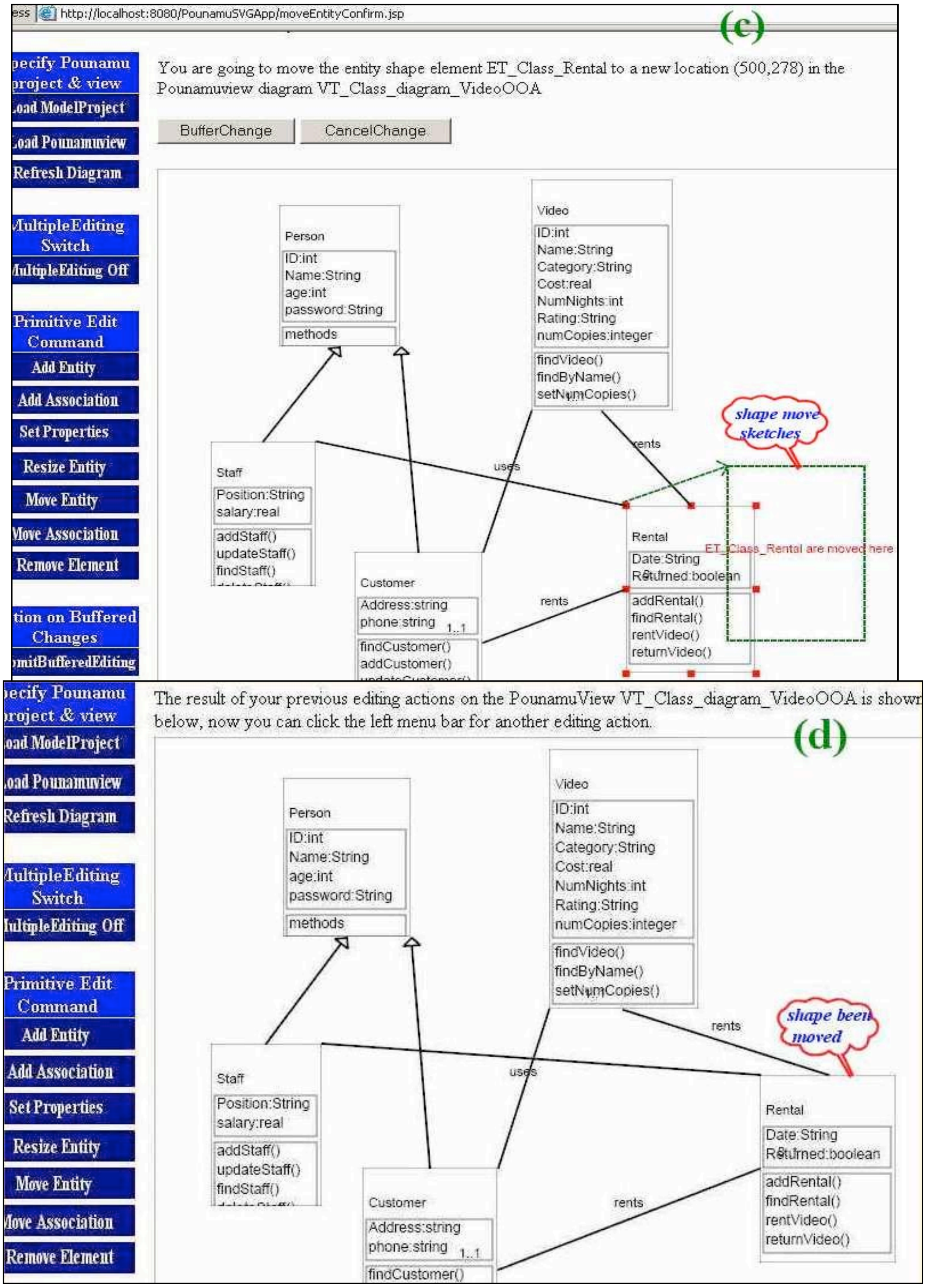
Move entity type model element to a new location :

First, please select the shape from the following diagram to move.

8080/PounamuSVGApp/moveEntityTo.jsp

Click the place where you want to relocate the selected shape :

You have selected an model element ET_Class_Rental to move. Please click the new loca
where you want to place this selected shape element.



http://localhost:8080/PounamuSVGApp/moveEntityConfirm.jsp

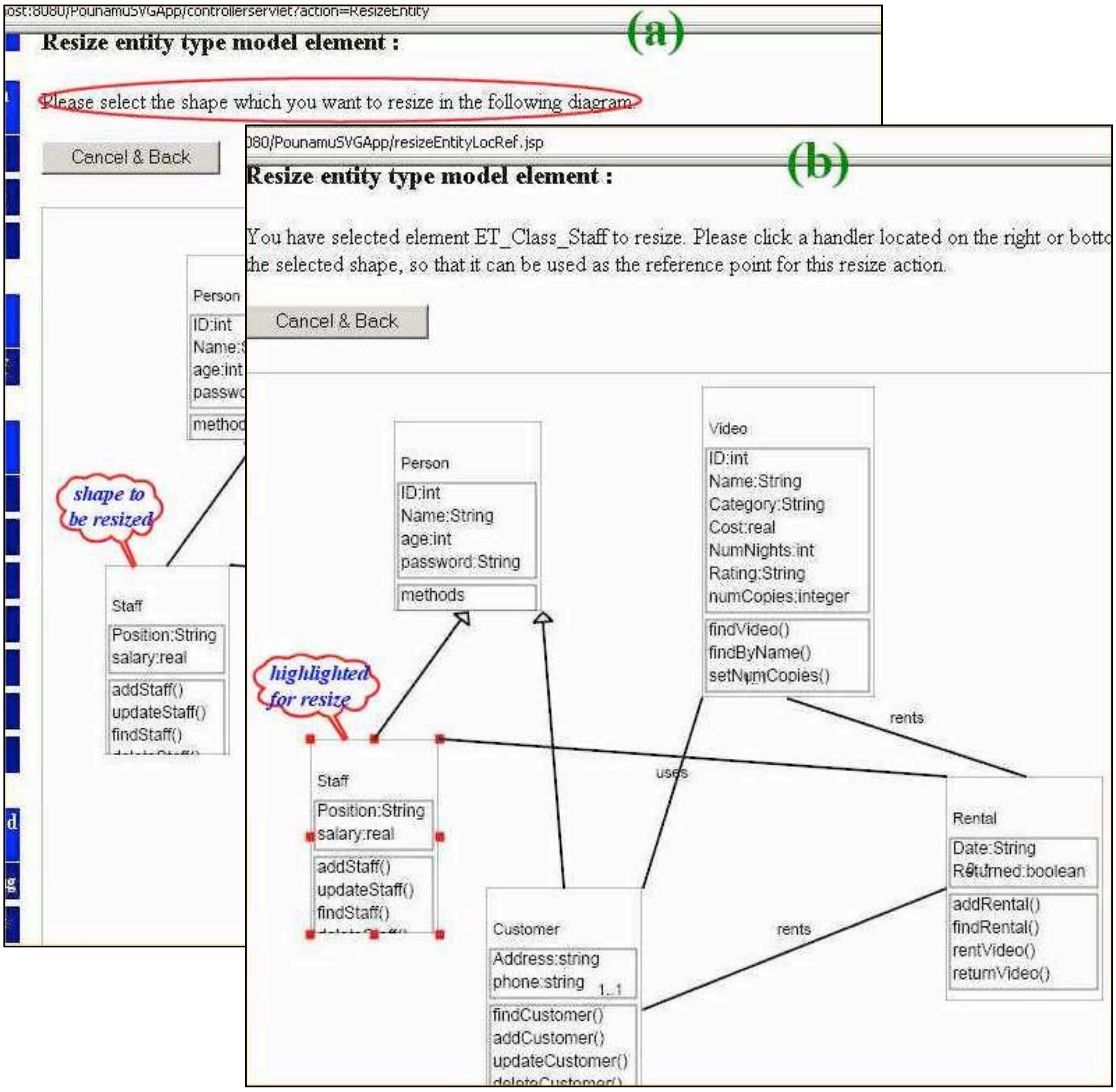
You are going to move the entity shape element ET_Class_Rental to a new location (500,278) in the Pounamuview diagram VT_Class_diagram_VideoOOA

The result of your previous editing actions on the PounamuView VT_Class_diagram_VideoOOA is shown below, now you can click the left menu bar for another editing action.

Figure 6-16 SVG version "Move Entity Shape" Running Examples

6.3.7 Example of Resizing an Entity Shape

The last editing action we need to mention is shape resizing. Its execution procedures are illustrated in Figure 6-17. This action also starts with clicking the corresponding left menu item. Then, a web page with a hint message “please click the shape that needs to be resized” appears (a). After that, the user clicks the target shape on the diagram, and this results in a new page with the selection highlighted (b). Next, the user needs to specify a reference point by clicking a red handle(c). Following is another click on the diagram, and this second clicked point is regarded as the reference point’s destination after resizing. After completing the above steps, the system calculates the resizing proportion according to the position values provided and displays a page (d) with new sketches representing the simplified resizing action. Finally, the user can submit this editing and obtain the result as shown in (e).


ost:8080/PounamuSVGApp/controllerservlet?action=ResizeEntity

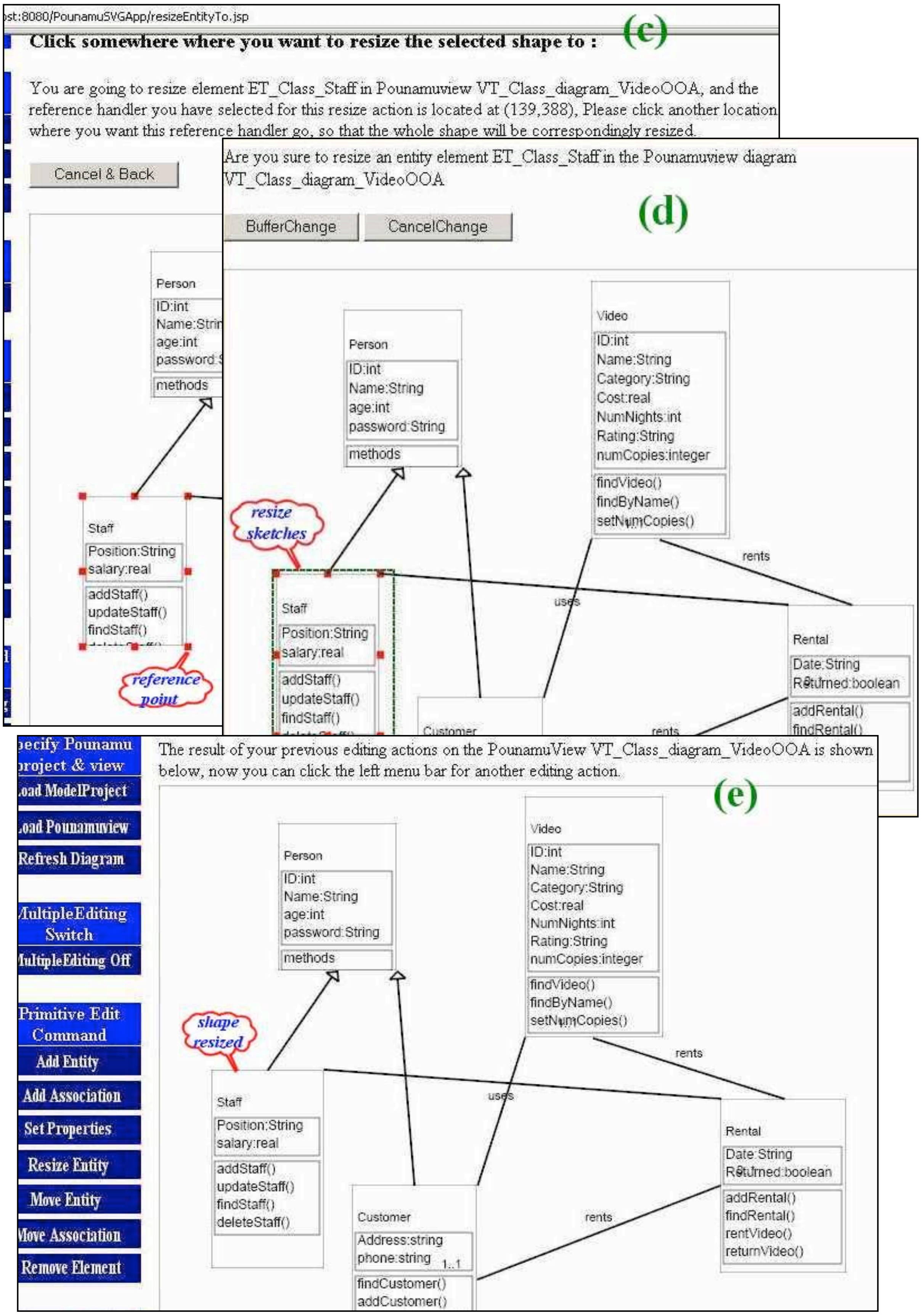
Resize entity type model element :

Please select the shape which you want to resize in the following diagram.

080/PounamuSVGApp/resizeEntityLocRef.jsp

Resize entity type model element :

You have selected element ET_Class_Staff to resize. Please click a handler located on the right or botto the selected shape, so that it can be used as the reference point for this resize action.


st:8080/PounamuSVGApp/resizeEntityTo.jsp

Click somewhere where you want to resize the selected shape to :

You are going to resize element ET_Class_Staff in Pounamuview VT_Class_diagram_VideoOOA, and the reference handler you have selected for this resize action is located at (139,388), Please click another location where you want this reference handler go, so that the whole shape will be correspondingly resized.

Are you sure to resize an entity element ET_Class_Staff in the Pounamuview diagram VT_Class_diagram_VideoOOA

The result of your previous editing actions on the PounamuView VT_Class_diagram_VideoOOA is shown below, now you can click the left menu bar for another editing action.

Figure 6-17 SVG version “Resize Entity Shape” Running Examples

6.4 Implementation Experiences

Before commencing this project, we had little idea of what SVG was and how it works. All that we knew was that SVG is an XML based web graphics format and very rich in interactivity and animation. After this project, we have mixed feelings about this newly emerged graphics format. On one side, we are very impressed with its interactivity features, especially when comparing with our previous working experience with GIF. On the other hand, we think SVG still needs quite a lot of improvement. For space reason, we cannot cover all the strengths and weaknesses that have been found, and only those practices that are closely involved in building this SVG version thin-client Pounamu tool are addressed.

Implement an optional buffering facility

A distinct feature that our SVG thin-client tool has is to allow a sequence of edits to be buffered on a user's own cached diagram before being sent to the Pounamu application server and actioned on the shared diagram. We believe this optional buffering facility is helpful to improve system performance and efficiency. The reason why such a facility is implementable in the SVG version tool owes to the fact that the web component-stored SVG XML data is manipulable. By comparison, this wasn't possible to do with the GIF-based version as it uses the Pounamu application server to generate diagram images. An additional difference between SVG- and GIF-based thin-client tools is the appearance of diagrams on the web browser. The SVG diagram, rendered by the browser plug-in, appears more polished and detailed than the other one.

Construct an SVG image in Java rather than using XSLT

XSLT (Extensible Stylesheet languages Transformations) is a declarative scripting language, which is designed for transforming the structure of an XML document. It can be used to read all kinds of XML input and to create all kinds of XML and non-XML output.

In our system, the initial idea of constructing the SVG version of a PounamuView diagram via XSLT transformation came from two facts:

- Pounamu is an XML-based CASE tool;
- SVG is XML in its nature;

However, after some investigation, we found the above idea is not as easy to implement as we thought it to be. The key issue is that a *PounamuIcon* (shape/connector) can be very complex and contain several layers of sub shapes. For example, Figure 6-18(a) shows an example of *PounamuIcon* that contains three layers of sub shapes. The first layer is *PounamuPanel 1A*. The second layer is composed of a *PounamuPanel 2B*, a *PounamuMultipleLinesInput 2C* and *PounamuOneLineInput 2D*. The third layer contains another *PounamuOneLineInput 3E* within the second layer's *PounamuPanel 2B*. When writing an SVG element for such an icon, we have to know the arrangement of all the sub shapes appropriately (because of SVG's vector format), including their X and Y position values. In a Pounamu XML file, such

3E

2B

2C

2D

1A

information is only given indirectly through Java Layout Manager Constraints (Figure 6-18(b)). This means some programming is needed to extract these X,Y values. But XSLT, which is not a fully programming language, is very limited to perform this sort of task. That is why we decided to write a special java class to generate an SVG image of the specified PounamuView diagram.

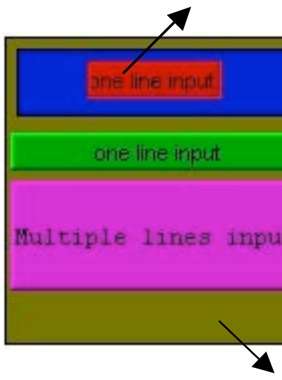


Figure 6-18(a) A PounamuIcon Example

```
<?xml version="1.0"?>
<!DOCTYPE pounamushape SYSTEM "icon.dtd">
<pounamushape>
   <source>S_DD</source>
   <type>pounamu.visualcomp.PounamuPanel</type>
    ....
   <property>
      <propertyname>layoutManager</propertyname>
      <propertytype>LayoutParameters</propertytype>
      <propertyflag>visual</propertyflag>
      <propertypath>this</propertypath>
      <propertyvalue>
         <layouttype>VerticalFlowLayout</layouttype>
         <vgap>5</vgap>
      </propertyvalue>
   </property>
   <subshape>
      <type>pounamu.visualcomp.PounamuPanel</type>
      ....
     <property>
         <propertyname>layoutManager</propertyname>
         <propertytype>LayoutParameters</propertytype>
         <propertyflag>visual</propertyflag>
         <propertypath>this_comp0</propertypath>
         <propertyvalue>
            <layouttype>FlowLayout</layouttype>
            <vgap>5</vgap>
            <hgap>5</hgap>
            <alignment>1</alignment>
         </propertyvalue>
      </property>
      <subshape>
         <type>
           pounamu.visualcomp.PounamuOneLineInput
         </type>
         .....
      </subshape>
   </subshape>

   <subshape>
      <type>pounamu.visualcomp.PounamuOneLineInput
      </type>
      ....
   </subshape>

   <subshape>
      <type>pounamu.visualcomp.PounamuMultiLinesInput
      </type>
      ....
   </subshape>
</pounamushape>
```

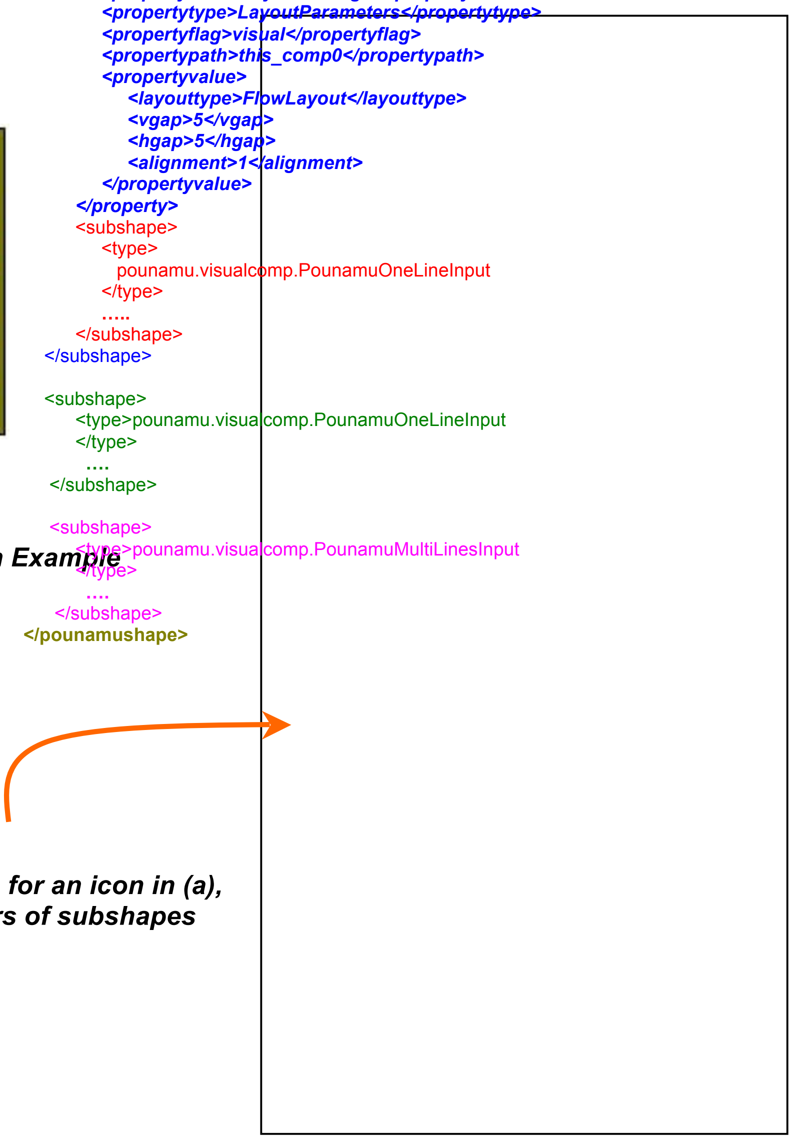


Figure 6-18(b) A Pounamu XML file for an icon in (a), Font colors are matched with colors of subshapes

Problems with Different SVG implementations

SVG, being a newly emerged technology, is still in the development stage, and this status has been reinforced by the fact that different SVG viewers have implemented only part of the complete SVG specification. In our system, Adobe SVG Viewer6.0 has been used since it is the most comprehensive implementation so far. However, SVG Viewer6.0 is still in its experimental stage and subject to changes, and there will be some time before it can be improved to a full release version.

Another noticeable SVG implementation-specific feature involved in this project is the server-connection facility. You may remember the function *getURL()* was applied to realize the communication between the client and the server. In fact, this method is not part of any standard and limited to Adobes SVG and Batik SVG viewers which have implemented it as an extension. Future work includes research on SVG implementation-independent server connection facilities, so that our application can run in any standard environment implementing SVG.

6.5 Summary

This chapter describes implementation details of our SVG version of a thin-client Pounamu tool, including how to construct on-the-fly SVG images for the specified PounamuView diagram and how to capture/interpret SVG events as application dependent editing commands, etc. In addition, as SVG is a new web graphics format which some people may not be familiar with, we also gave a brief introduction of what SVG is and its current implementation status. Finally, we presented some running examples to show user interaction with this prototyping system.

Chapter 7: Evaluation

In addition to reviewing and comparing our prototyping work against other thin-client diagramming tools, we have carried out two evaluations of our prototype Pounamu/Thin thin-client diagramming system. This chapter presents the evaluation results. These identify strengths and weaknesses of our approach, as well as other areas that need further improvement. Firstly, we describe a cognitive dimensions [80] evaluation of the thick-client Pounamu-implemented UML diagramming tool, the GIF-version thin-client UML tool, and the SVG-version thin-client tool with edit buffering. This compares their relative characteristics along a variety of usability dimensions. Then, we summarize the findings of a small evaluation survey where several users performed a series of tasks and gave impressions about their experiences using these three versions of UML diagramming tools. Finally, we give our own reflective analysis of this thin-client diagramming prototype system, with an aim to point out a number of ways to improve its usability.

7.1 Cognitive Dimensions Evaluation

The cognitive dimensions (CDs) framework was developed to provide a broad-brushed approach to evaluate notations and interactive systems, rather than the detailed evaluations offered by many usability evaluation techniques. The overall purpose of this framework is to provide a clear basis for discussion and evaluation by pointing out the cognitive artifacts of the system being designed. In order to achieve this aim, CDs provides a vocabulary for analyzing the interactions between the structure of information, environments in which the information is managed, and the type of actions that users want to perform.

The vocabulary recommended by CDs is a set of “dimensions” that can be used to describe different, sometimes competing, characteristics of visual languages and tools. These dimensions are neither design guidelines nor a cognitive model of the user, instead they are discussion tools meant to emphasize areas which are of concern to designers when building their notations and systems. The CDs framework currently suggests six general categories of user activities, and 13 dimensions. In the following paragraphs, a brief review is given to those cognitive dimensions that are of our interest. We will discuss where and how our systems fit in these dimensions, and some important findings are described as below.

Viscosity: Resistance to Change

Viscosity is a measure of the amount of work needed to achieve a local change, i.e., how much work is involved to implement a small logic or structural change in a local model or structure. For visual tools, high viscosity generally can reduce the effectiveness and result in frustration.

We feel that the GIF-version thin-client UML tool is more "viscous" than its thick-client counter parts. This is especially true with moving and resizing functionalities. As has been demonstrated previously, multiple interactions are required to perform these sorts of actions via our thin-client prototype. This characteristic is usually cited as a negative feature of thin-client diagramming approaches. In comparison, only a simple mouse-controlled drag-and-drop like operation is needed to perform similar moving and resizing tasks in the original thick-client Pounamu modeller.

Hidden Dependencies:

A hidden dependency occurs when two entities are dependant on each other but the in-between relationship is not fully visible to a user. With hidden dependencies it is very difficult for a user to understand the whole information structure. Therefore, changes to the information may either seldom be made because of the user's uncertainties or cause some serious, unpredictable problems as the result of modification.

In our Pounamu UML diagramming tools, multiple views of UML model elements are available in both thick-client and thin-client versions, and dependencies between visual elements and model elements are implemented via shape and connector names. However, the way of viewing these dependencies is very different in these two versions of tools. Pop-up menus on thick-client shapes allow access to the relevant information, including other views in which they have appeared, while thin-client tools requires users to look up similar information via buttons and non-contextual option lists. Thus, it is apparent that underlying dependencies are more easily detectable in the thick-client diagramming tool than in thin-client versions.

Consistency:

Consistency simply signifies how consistently the tool/language represents concepts and allows the user to perform certain tasks. Consistent notations or information structure tends to minimize user knowledge required to use a system by letting users generalize from existing experience of the system or other systems. Generally consistency can be assessed by asking questions like: when the user knows parts of the system, how much of the rest can be successfully guessed?

We feel that all three UML tools discussed are consistent in that they use the same visual representation and editing operations, so that the user requires little extra effort to study the thin-client version if he/she is already familiar with the thick-client one, and vice versa. Actually, you can only notice the following two significant differences between these tools:

- The GIF- and SVG-based thin-clients diagramming tools employed a button-based, page organized approach to manage editing and viewing operations. Although this approach normally takes up more screen space, it provides users with a simpler, less complex interface than the large thick-client tool pop-up and pull-down menus.

- Since SVG and thick-client diagrams are rendered with scalar graphics rather than binary data, they look more refined and print better than GIF ones.

Closeness of mapping:

Closeness of mapping indicates how closely the system's concepts for objects and actions match the user's intuitions. The motivation of this dimension is to minimize the extra knowledge required to use the system, so simplifying all task action mappings.

In our case, GIF- and SVG-version thin-client diagramming tools take a web page-based approach to view and interact with diagrams. Each user interaction (a mouse click) usually causes a POST to the web server and then the displayed diagram is replaced with the latest version. In comparison, the same diagramming viewing and interaction functionality are achieved in a more natural way in the thick-client tool, i.e., the drag-and-drop and pop-up menu/selection-based approach is taken. Again, with support for collaboration, the thin-client tools always require additional user-directed refresh of pages to show other's work, whereas the thick-client Pounamu UML tool automatically displays one diagrammer's work on collaborators' screens with push-based view update and redisplay. Following this line of thought, our opinion is that the thin-client's page-based approach supports less direct manipulation activity and provides a less natural diagramming collaboration environment, but it fits a user's mental model of web-page post and display.

Hard mental operations:

Hard mental operations are processes that require high demand on Cognitive Resources and are very difficult to complete. Generally speaking, hard operations can cause user fatigue and mistakes. There are two defining features of constructs and objects that generate "hard mental operations." First, ask yourself whether the process gets incomprehensible when two or three of these constructs are combined together. Second, ask whether there is another way that makes the work easier to accomplish.

We attempted to reduce the need for hard mental operations in the design of our thin-client tools and we feel this goal has been partially fulfilled. Actually, the learning curve of the thin-client tools is less in many respects to the thick-client tool, as it presents interaction options in more explicit ways, such as always-visible buttons and frame-based property editing. However, some editing actions such as moving and resizing require non-intuitive sequences of operations. In comparison, the thick-client counterpart realized the above named editing actions via the spontaneous drag-and-drop technique.

Visibility and Juxtaposability:

The visibility dimension simply represents whether required information is easily accessible or whether it can readily be made available. The visibility dimension is different from hidden dependencies. While hidden dependencies concern whether the relationships between two entities are apparent, visibility measures the amount of work involved in making a certain item available. Juxtaposability is an important part of visibility, and it describes the ability to view two pieces of an information structure at any particular time for comparison tasks. This is helpful to reduce the user's memory load, and thus avoid possible errors.

In the thick-client UML modeller, multiple views are allowed to be displayed side-by-side or accessed via both a tree and tabbed panes structure, and visual and model properties of model elements are simply available for change via the right-side property panel. The work currently underway to provide a zoomable Pounamu user interface will make our thick-client tool even more attractive. In comparison, although it is possible to display different views in the thin-client tools, a more complex structure has to be used, i.e., multiple web browsers, each showing a specific view, need to be opened and positioned on the host PC desktop. A simplified list of views can be provided for the user to select from. A potential restriction to this multi-web browsers approach is the size of a monitor. So, some work needs to be done to address the visibility problem of thin-client tools. Possible solutions include providing a scalable diagram and applying a standard focus+context approach.

Progressive Evaluation:

Progressive evaluation means that users are able to test and evaluate their problem-solving progress at frequent intervals and obtain feedback on how well the work has been done so far. Experiments show that the usability of a system that provides progressive evaluation is significantly improved and users' anxiety is also reduced when using this sort of system.

Looking at the Pounamu-based thick-client UML tool and the two other thin-client diagramming UML tools involved in this evaluation study, we feel that the SVG-based thin-client tool with buffered editing supports a higher level of progressive evaluation. Actually, in the SVG-based diagram-editing environment, users can buffer a series of editings on their own copy of the diagram. Before sending these editings to the server and letting them actioned on the shared copy of diagram, users are allowed to observe these buffered editings and modify those unwanted ones. In the long run, this buffered editing mechanism is quite useful to improve the system performance because the number of wasted editings, as well as the involved undo and redo, is reduced. By comparison, the GIF version thin-client tool does not allow users to see in advance what the diagram would look like as the result of a series of editings. When performing collaborative work, the thick client tool seems better in the aspect of progressive evaluation. The reason that we say so is that the push-based view update and redisplay approach taken in the thick-client Pounamu UML modeller allows other's work to be shown immediately and automatically while the thin-client tools require user-directed refresh of pages to do the same thing.

7.2 User Survey

To assist us in evaluating our work, we carried out a user survey where participants were asked to complete a list of tasks and answer a number of close- and open-ended questions.

7.2.1 Survey Procedure

Nine persons were invited to participate in the usability survey. Each participant was required to fill out a pre-test questionnaire (see Appendix B), which includes questions like: Is he/she a software engineering practitioner? Has he/she used CASE or Meta-CASE tools before? Does he/she have experience of using our Pounamu tool? By analyzing answers to these questions, we found all participants had used CASE or Meta-CASE tools to some extent in their previous projects. Six of them had working experience with Pounamu.

The survey was done in the computer science department offices at the University of Auckland. The Tomcat server to which our GIF- and SVG-based thin-client web components had been deployed was on the same host as the Pounamu application server. The PCs that the participants used to perform the tests were connected via a LAN.

We designed two tasks for participants to perform (see Appendix C). These tasks covered both of our thin-client prototypes and their facilities. In task 1, evaluators were required to model the same class diagram using the thick-client Pounamu-implemented UML diagramming tool, the GIF-based thin-client UML tool and the SVG-based thin-client tool with edit buffering turned off. Task 2 was simply designed to test whether the buffering facility enabled in the SVG-based thin-client tool is beneficial to users of diagramming tools. The general purpose of these tasks was to compare users' experience with using three alternative tools and obtain a range of feedback on the thin-client Pounamu UML diagramming tools that we have developed.

Participants were given a brief tutorial before they were immersed into doing the specified tasks. The contents of the tutorial covered: a brief description of the Pounamu meta-CASE tool, and the Pounamu-based GIF and the SVG version thin-client tools; instructions of how to use the above three tools; demonstration of use of the three tools; explanation of the main goal of this evaluation and a description of tasks.

After finishing the tasks, the participants were asked to fill out a post-evaluation questionnaire (see Appendix B) that contained a set of open and closed questions. Closed questions were mainly designed to test prototype characteristics such as ease of learning, ease of use, system efficiency and response time, while open questions were really a loose guide and aimed at getting participants thinking the issues we are interested in. For your reference, some examples of closed and open questions have been listed in Table 7-1.

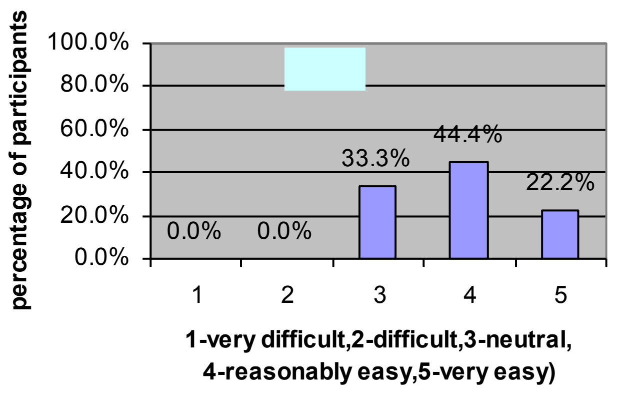
(a) (b)

Table 7-1 Examples of Survey Questions

Question Category:	**Question Examples:**
Closed Questions:	a) Is it easy to use our GIF- or SVG-based Pounamu/Thin tool for editing a diagram? b) When performing editing tasks via our thin-client prototypes, how do you feel system response time? c) Do you agree that our GIF- or SVG-based Pounamu/Thin prototype contains all the facilities you need to effectively perform tasks?
Open Questions:	a) How easy to use our Pounamu-based thin-client diagramming tools vs. its thick-client counterpart? Do our thin-client tools provide all the core functionalities required to perform tasks? b) Which one of two thin-client diagramming tools do you prefer to use? c) How do you feel edits buffering facility supported in the SVG version thin-client tool?

7.2.2 Survey Results

Due to the relatively small number of users surveyed (9 persons), we can't draw any statistical conclusions from the tests. However, some useful information can still be obtained from the tests. For example, Figure 7-1~Figure 7-3 illustrate user responses to most closed questions of the post-evaluation questionnaire. After analyzing these figures, we found both of our thin-client prototypes are reasonably easy to use as they provide low-complexity user interface and functionality. Users generally feel that they can perform the tasks effectively with the facilities currently supported in these prototypes. In addition, system response time of both tools was found to be acceptable.



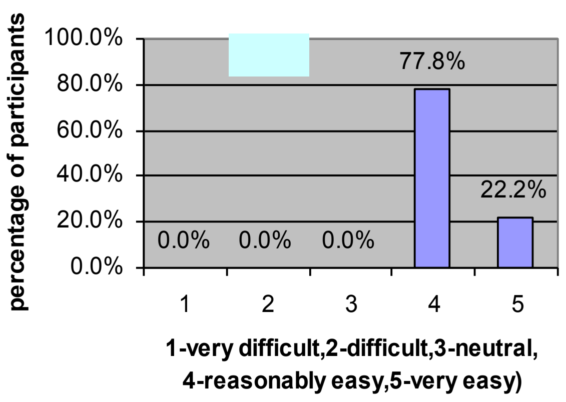
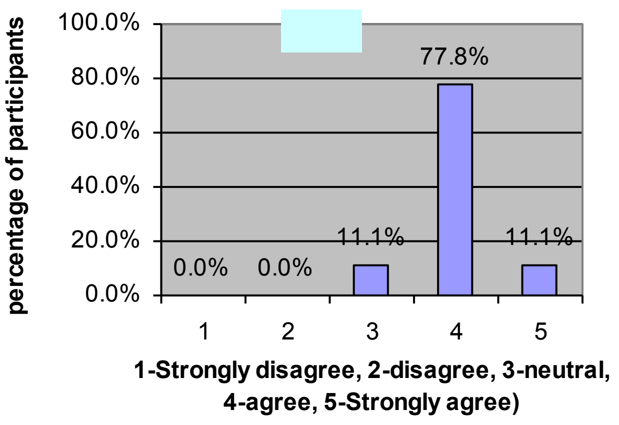


Figure 7-1 System Easy-to-Use Test Results,
(a) and (b) are for GIF and SVG-based thin-client user interface prototypes, respectively

(a)(a) (b)(b)



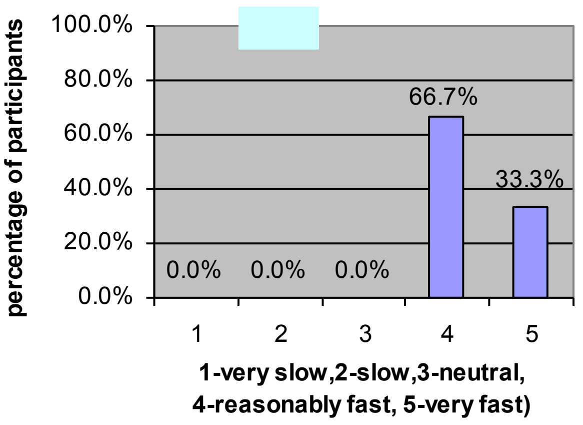
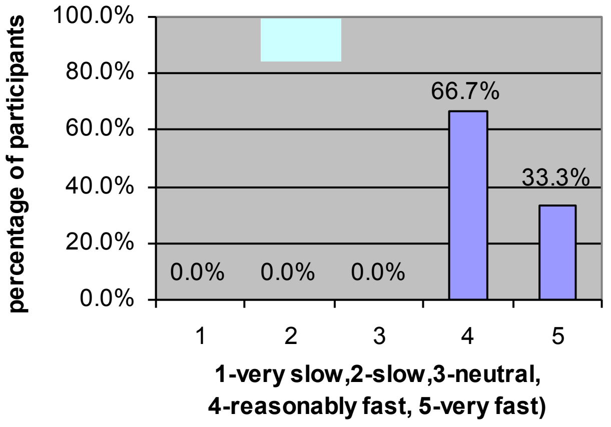


Figure 7-2 System Efficiency Test Results
(a) and (b) are for GIF and SVG-based thin-client user interface prototypes, respectively



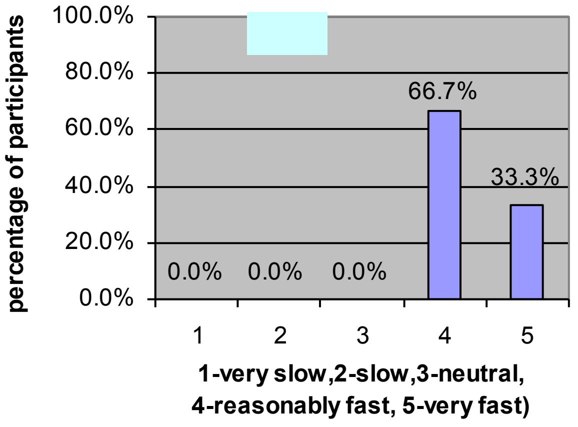


Figure 7-3 System Response Time Test Results,
(a) and (b) are for GIF and SVG-based thin-client user interface prototypes, respectively

In addition to the above positive responses to each individual prototype, we also obtained some comparative results from participants' replies to other questions of the post-evaluation questionnaire. Here, we present some important observations below.

Most users perceived that our thin-client tools provided the same functionalities as the thick-client version but used a combination of page-based and frame-based web browser metaphors rather than overlaid windows as in the thick-client tool. In addition, they felt user interaction supported in the thin-client tools was acceptable, with creating and modifying diagram components completed very easily. However, they also admit as they are used to using pop-up menu and drag-and-drop techniques to interact with diagrams, they need a little time to adapt our thin-client way of diagramming because it is a little less natural to perform some special actions (e.g. resizing and moving) via a multiple of clickings. Despite this deficiency, most participants still think the learning curve for inexperienced users of the Pounamu tool was found to be less for users of the GIF-based thin-client tool.

Although both SVG-based and GIF-based diagrams are easily readable through a web browser, feedback shows that users preferred building and rendering the diagram on the SVG tool rather than the GIF one. There are several reasons for this preference. First, users generally felt that individual diagram components in SVG diagrams are easier to be pinpointed for selection than the equivalent in GIF diagrams. This is especially true for those Pounamu connectors with thin width. GIF versions of such components always appear with “jaggies” that cause them difficult to be located. From our observation, users tend to get frustrated when they missed highlighting the right connector even after the second try. The second reason is that an SVG rendered diagram looks more polished than the GIF one because of its vector-based graphics nature. However, the most influential reason is that most users think tooltips and dotted outlines provide useful signs to help them to understand the progress made toward accomplishing a certain editing task. For example, they commented “being able to see the editing result before committing make them feel that a built-in, though limited, ‘undo’ functionality exists in the SVG version thin-client diagramming tool”.

The edits buffering facility in the SVG version thin-client tool received mixed support. Some users like this facility and feel it is useful when performing a collection of related editings. For example, in order to get a better layout of a class diagram, a set of moving operations can be performed with buffered editing on, and then you can observe and readjust the relative positions of shapes before submitting these operations. However, despite this positive opinion, users pointed out that there is some confusion when practicing multiple editings. For example, suppose there are two editings (moving followed by resizing) to be performed on an entity shape in the SVG thin-client tool with the buffered facility turned on. After moving, the dotted outlines are overlaid on the original diagram showing the intended new position of the shape. We found users are confused with which one (the dotted outlines or the original shape) should be clicked for the following resizing action, Although after our explanation, users can make the right choice, we still feel that a better solution has to be sought to get rid of this confusion.

To summarize, both our thick- and thin-client UML diagramming tools provide suitable support to diagramming. User preferences lean to using one or the other more than any other consideration. Most users in our evaluation of the same Pounamu UML diagramming tool felt that they would be happy to use any of the three approaches, the Java Swing-based thick-client tool, the GIF-based thin-client version or the SVG-based version. Novice Pounamu users suggested that the learning curve for the thin-client tools was less than the thick-client one and that it was an advantage that they needed no installation and configuration to use. The buffering of edits in the SVG version received mixed reaction. This may be a case of further experience with it being needed and it can be turned off if not wanted. Response time of the thin-client tools was found to be acceptable despite their both using full diagram and page refresh after every editing operation. Client-side scripting of some operations e.g. highlighting diagram elements in the SVG plug-in, was suggested to improve some aspects of response-time by one user. Currently changing a tool specification can only be done with the thick-client tool designer, which is seen as a disadvantage by some users if thin-client editing of the resultant tool was desired.

7.2.3 Suggestions on Improvement

Besides the good and bad experiences summarized as above, we also received various useful suggestions on how to improve the usability of our SVG-based and GIF-based thin-client diagramming tools. Here only those important ones have been picked and listed in the following paragraphs. The main aim is to give some indication for avenues of future work with our approach.

The main stream of suggestions for improving system usability involved reducing the number of user interactions in performing certain tasks such as resizing and interaction. We doubt that there will be major breakthroughs on this aspect with regard to our GIF-based thin-client tool. However, there does exist a way to work around this problem in the SVG version tool, i.e., implementing the drag-and-drop like moving and resizing operations. The approach that can be taken is to write and register client-side ECMAScripts as handlers to mouse events specified in the SVG standard.

Although it is unavoidable to have multiple interactions in performing moving/resizing actions in the GIF-version thin-client tool, at least some redundant clicks could be eliminated. For example, users suggest the last confirmation step currently in the GIF-version tool could be removed, and undo/redo facilities instead should be provided to roll back and repeat the last operation. From our perspective, this suggestion is quite reasonable as users can’t preview the result of their editing in the displayed diagram, and text messages displayed on the top of web page reveal only trivial editing information. Most users feel that these messages appear meaningless in the context of our visual programming tool. Moreover, as diagrams become large in size, they don’t want to be bothered to scroll up to check messages displayed on the top. So in their real work, they tend to ignore these hint messages once they get used to performing tasks with our tools.

One of our experienced software tool users also suggested that designing a chain of relevant editings is helpful to achieve better usability. For example, in the common practice, creating a new entity is always followed by setting its properties, and this is then followed by the resizing operation, so an effective way of doing work could be to drop into the property sheet editor page after an adding operation, then into the resizing page after that.

Besides the suggestions mentioned above, users also wrote down a variety of comments aiming to assist us in improving the user interfaces of both thin-client diagramming tools. Examples include:

- *A larger canvas area in the web page to display a view diagram.* As a diagram with visual components can easily become large in size, the area of the web page left to display the diagram should be a little bit bigger than its current size.
- *Pop-up window option for property setting page.* As can be seen from Chapter 5, frame-based pages are used to set up properties of shapes and connectors. While the left frame displays the diagram with the highlighted shape, the right frame can be used enter new values. The problem with this user interface is as the two frames can become larger, users always need to readjust both frames' size to either observe the highlighted component or enter new property values. A better alternative is to display an HTML form-based property setting page in a separate window.
- *Rearrangement and resizing left side menu buttons.* Since there are quite a number of menu buttons, they currently can't fit inside one screen. Suggestions are to make them smaller or rearrange them for a better layout. A possible solution would be to use a dynamic pull-down menu.

7.3 General Comments

Reflecting back upon our work we have reason to be happy with our Pounamu-based thin-client diagramming tools. In general, we feel the approach we have taken has great potential and offers several unique benefits over other thick- and thin-client software engineering tools.

From users' feedback we feel some features that our thin-client tools provided (including ease of learning and use) are the major benefits. Compared with the thick-client counterpart, most users think our SVG- and GIF-based thin-client tools support all the essential functionality through less sophisticated web-based user interfaces. There is no doubt that positive responses like these are the best reward to us. However, critical opinions are important as well because they give us insight into potential future work. In fact, back in the design stage, we realized there might be arguments over our tools because we had to make some design compromises. For example, 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 was desired, but for other users who prefer simple lightweight tools this is definitely an advantage.

Until now, the major issue raised by the survey is multiple interactions are required to perform editing operations such as moving and resizing. As mentioned earlier, there is not too much that can be done to change such practices in the GIF version thin-client tool because of the limitation on its binary data format. However, in the SVG-based tool, realizing resizing/moving as drag-and-drop like operations is not a difficult task due to SVG images' interactivity characteristics. The only issue that bothers us is how fast the diagram can be repainted when the mouse is moved. As we already know, shapes or connectors in the Pounamu Meta-CASE tool can be very complex and there are many computations going on behind the scenes. So we can't help wondering whether client-side JavaScripts can handle this amount of computation-intensive code. To clarify this, we have prototyped an implementation of the drag-and-drop like resizing/moving as an optional editing mode in the SVG thin-client tool. According to users' personal preferences and available resources, they can now choose one among three alternative editing modes, including a standard mode (where both buffering and script are turned off), multiple editing, and script driven. Compared with the other two, script-driven editing requires a little more client-side computing capability, but rich interactions similar to those that have been experienced in the thick-client diagramming tool can be achieved.

A bonus benefit with our thin-client diagramming tools is that group collaboration at different locations can be carried out fairly easily. Team members can edit a shared copy of diagram as long as their computers have a web browser (for SVG, a extra plug-in). Although thread-safe coding principles have always been adhered to in our development process, we aren't 100% sure that our prototype is a bug-free system in the group collaboration environment. This is especially true with our SVG version thin-client tool with buffering facility on, as a local copy of buffered editings has to be successfully merged with other people's work. The result could be that some of the editings are executed as expected, while others just end up failing to pass server-side validation. In order to ensure the quality of our system, a large number of testing scenarios have to be explored and performed.

Last we want to mention the usability issues derived from the prototype nature of our tools. As we have concentrated on realizing the useful functionality that our tools should provide, this category of issues has not received its due attention. However, we are confident that some of them can be easily fixed and our tools could evolve into sophisticated versions had we the time to implement them.

7.4 Summary

In this chapter, we carried out two evaluations of our thin-client Pounamu diagramming tools. In the first evaluation, we analyzed usability issues by applying cognitive dimensions framework to our prototypes. The second evaluation is a user survey via a questionnaire of experiences using our GIF- and SVG-based thin-client diagramming tools and their thick-client counterpart. Through analysis of these evaluations, we identified the strengths and weaknesses of our prototype system while with a view to potential future work.

Chapter 8 Conclusions and Future Work

This is the concluding chapter of the thesis. It summarizes the work done in this project and outlines possible future work.

8.1 Conclusions

In this project, we studied some related work on web-based software engineering tools, especially on web-based diagramming tools. The advantages and disadvantages of these existing tools have been investigated. Based on this investigation, we presented the proposal of providing a thin-client extension to our original Pounamu Meta-CASE tool. Then, we have gone through activities involved in the system development life cycle, including system requirements determination, object-oriented analysis and design, prototype implementation and testing. Finally, we carried out a cognitive dimensions evaluation and a user survey to explore existing usability issues. Our achievements are:

Identified system requirements: Our system should provide an HTML-based thin-client user interface (UI) to provide end users with an alternative way of accessing the Pounamu Meta-CASE tool. A key requirement for this thin-client UI is to allow users to directly interact with diagrams embedded in a web page. Since a Pounamu diagram is often composed of two elemental graphical components such as entity shape and association connector, all the core editing commands actioned on these components should be available via HTML buttons or links etc. These commands include Add/Association, Remove Entity/Association, Move Entity/Association, Set Entity/Association Properties, and Resize Entity etc. In general, our system should comply with the conventional approach for design of web based UIs, i.e., using the POST/GET page display metaphor to view or edit diagrams.

Designed system: As our objective is to provide a thin-client diagramming extension to the original Pounamu Meta-CASE tool, it is natural to use the web-based N-tier software architecture as the backbone of our system. The clients of our system are HTML pages downloadable from web browsers. Diagrams can be rendered in many commonly used web image formats such as GIF, SVG, VRML and so on. The web components, running on the web server and accessed by web browsers, are responsible for dealing with user interactions with diagrams. Their main functionalities include collecting editing command-related information and sending it to the Pounamu server for execution. A remote interface component was designed and plugged into the Pounamu application server to support remote diagramming functionalities. It is mainly used to receive editing messages from the web components and then convert them into Pounamu recognizable command objects.

Implemented system prototypes: We implemented two prototypes, namely GIF- and SVG-based Pounamu/Thin tools. For both prototypes, we used Java Servlets, JSP and JavaBeans to implement the web components, RMI to implement the remote interface plug-in. The fact that a PounamuView diagram is saved in XML format and editing command objects executed on a view can be easily converted to and from an XML format supports our decision to package user interactions with diagrams into XML-based command messages. These are sent across RMI to the Pounamu application server and committed on the shared diagram. To generate PounamuView diagrams as GIF and SVG images and display them on Web browsers, we have provided GIF and SVG renderers. But, these two are quite different. The former simply requests that the Pounamu server returns a GIF image of a view each time a view is changed, and then it embeds the GIF image inside a web page along with editing command buttons, property sheet labels and text fields. The SVG renderer instead requests an XML-encoded version of the view from the Pounamu server. It then traverses the XML and generates an SVG encoding of the Pounamu shape and connector objects in the view XML, along with HTML for view and property editing as per the GIF-based renderer. We implemented the SVG translator in Java rather than using XSLT as its performance is much faster and requires less complex algorithms. The SVG is rendered by the browser plug-in, resulting in more polished, detailed diagram appearance than the GIF-based version. Another difference between the two prototypes is that the optional edit bufferring facility is enabled in the SVG version tool. This facility uses the web component-stored SVG XML data, meaning buffered edits do not need to be sent to the Pounamu application server and are hidden from other users until committed. This wasn't possible to do with the GIF-based version as it relies on the Pounamu server to generate diagram images.

Carried out system evaluation: We have performed two evaluations of our Pounamu/Thin diagramming prototypes, in addition to reviewing and comparing our prototyping work against other thin-client diagramming tools. In the first evaluation, Green and Petre's cognitive dimensions framework [81] has been applied to assess the relative characteristics of the thick-client Pounamu-implemented UML diagramming tool, the GIF-version Pounamu/Thin UML tool, and the SVG-version Pounamu/Thin UML tool. In the second evaluation, users were interviewed via a questionnaire of their experiences of using the above three tools. Evaluation results show that both the thick-client and thin-client UML diagramming tools provide appropriate support for diagramming and most participants felt they would be happy to use any of these tools. Although most users felt that it is a little less natural to use thin-client tools, as some special actions require a multiple of clicks (e.g. resizing and moving), they still suggested the learning curve for the thin-client tools is less than the thick-client one. Some features seen as advantages of SVG-and GIF-version thin-client tools include: they need no installation and configuration to use; and group collaboration at different locations can be carried out fairly easily. Inability to modify a tool specification is a significant disadvantage seen by some users who desire thin-client editing of the resultant tool. Currently changing a tool specification can only be done with the thick-client tool designer. The optional edit buffering facility enabled in the SVG version received a mix of positive and negative reactions. Response time of both thin-client tools was found to be reasonably fast despite them using full diagram and page refresh after each editing operation. Client-side scripting of some operations e.g.

highlighting/moving/resizing diagram elements in the SVG version, was suggested to be effective for improving system usability.

Reflecting back upon the prototypes that have been completed, we feel that they offer some benefits over the existing diagramming tools. When compared with those thick-client ones, our tools provide advantages such as no installation, less complex and easy-to-use user interface, and supporting collaborative work in itself. Then compared with existing thin-client diagramming tools, our tools are better on the following respects:

- ♦ First, the GIF-based version can be used on any modern browser because it needs no augmentation technologies such as JAVA Applet or browser plug-ins. Actually, it did not even contain any client-side JavaScript which may not run appropriately on some cross-platform browsers. Although the SVG-based version requires a SVG Viewer plug-in, many newer browsers have this installed since SVG is a standard recommended by W3 organization. More importantly, upsides brought by this SVG version tool seem far greater than the above downside. For example, with its rich interactivity users are able to perform most editing commands in the same way as they are used to with thick-client tools, including moving and resizing shapes/connectors in a diagram via drag and drop-like operations. With its optional buffer facility, the efficiency and performance of our thin-client diagramming tool gains significant improvement.
- ♦ Second, unlike those thin-client tools developed specifically for editing a specific type of diagram (e.g. UML class diagrams or sequence diagrams), our tools can be used to edit a versatile range of diagrams as long as notations and semantics for drawing diagrams has been specified and stored in the server-side XML files.

8.2 Future Work

Our thin-client visual diagramming tool is still a prototype. Only after considerable refinement can it become mature and sophisticated. Here we wish to highlight a number of possible extensions aiming at improving the functionality and usability of our prototype system.

Based on the evaluation results, we would like to make the following additions and changes to our system:

(1) Removing the editing confirmation step currently coded in the GIF Pounamu/Thin tool. By doing so, we can reduce the number of interactions involved in performing any diagram editing task. This in turn would improve the efficiency of our tool.

(2) Adding undo and redo functionalities to our GIF Pounamu/Thin. These additional functionalities allows users to roll back or repeat the last editing actioned on a diagram, so that users can enjoy more flexibility in using this diagramming tool.

(3) Chaining the related editing activities. In both of our SVG- and GIF-based Pounamu/Thin tools, an editing action is always initiated by clicking the left menu button. This is a little less natural for those users who are familiar with the thick-client way of doing things, in which pop-up menu/selection-based approach is taken to view and interact with diagrams. One way to ease this inconvenience is to reduce the number of button clicks required to perform certain tasks with our Pounamu/Thin. This can be accomplished by exploring and serializing some related editing scenarios. For example, users of the Pounamu tool tend to add a shape first, then set its properties, finally resize it. In our thin-client tool, we can simulate a similar editing sequence by dropping into the property sheet page after adding an entity, then bringing in the resizing action after property setting. By doing this, we save some button clicks that users need to make in editing a diagram.

(4) Rebuilding a movable menu bar. Currently, our GIF- and SVG-based thin-client diagramming tools contain a static menu bar on the left of every page. As a diagram can easily become large, users have to be bothered with scrolling down and up to find a right button. For this reason, we suggest to build a dynamically movable menu bar that can always be located in the current view port.

(5) Client-side scripting of some operations (e.g. moving/resizing diagram elements via drag-and-drop) currently implemented in our SVG version tool only considers Pounamu shapes with less complex appearance. For those shapes that are constructed with more complex java layouts and subshapes, resizing algorithms to be implemented in ECMAScript have to be carefully researched.

(6) Although thread-safe operations have been ensured in our prototype systems, quite a strict locking policy has been adopted in our implementation. For example, we enforce a sequence execution of all the remote editing commands from multiple users no matter whether the objects that these commands are applied to refer to the same or different diagrams. Future work include research on more moderate concurrency mechanisms.

In the current system, we used RMI to implement the remote editing interface to be plugged into the Pounamu application server. Although RMI is simple to use, it may not work very well for distributed applications running on different computers with different operating systems, programming languages and networks. For example, in our evaluation, one participant received an “RMI connection error” when trying to access our tool on his Macintosh computer. So, to support better accessibility and portability, our remote interface needs to be extended and implemented using other platform-independent, language neutral technologies such as CORBA or SOAP. Actually, it is very easy to adopt SOAP in our system because an XML-based message transmission approach has already been taken to build our system.

Another extension we have considered is to enable Scripting in our thin-client diagramming tool. Currently, the GIF version tool doesn’t contain any type of client-side scripts. This design decision came from two initial considerations. First, we want to make full of use of server-side resources. Second, we are interested to see how far this thin-client diagramming tool can go without the use of technologies such as JavaScript, browser plug-ins, Java Applet. Although the resultant tool has been rated reasonably easy to use, we still feel that a small amount of client-side scripts will be of great help to improve system performance and usability. For example, with these scripts, we can easily convey some complex visual properties which currently can't be represented with the limited web user interface resources. The number of round trips to the web server will also be reduced as some trivial computations could be handled by the scripts. In general, the combination use of client-side scripts and dynamical HTML will offer a more flexible and rich web-based user interface for users to view and edit diagrams.

As far as other improvements are concerned, the multi-user collaboration functionality supported inherently by our Pounamu/Thin tools should be refined. Currently users in a typical thin-client collaboration scenario have to purposely refresh the pages to show other's work, and this seems a little unusual compared with the push-based view update and redisplay approach taken by the thick-client tools. A solution to this deficiency would be the use of an HTML auto-refresh facility, i.e., automatically refreshing web pages every a few minutes to show what others have done. In addition, some proper use of highlighting would make diagramming users aware of elements that currently are worked on by others, and therefore some contradictory editing scenarios can be avoided. Potential future work include research on highlighting techniques commonly used in Groupware and find out ways to adapt them in our tools.

Reference:

1. Bentley, R., Horstmann, T., Sikkel, K., and Trevor, J. Supporting collaborative information sharing with the World-Wide Web: The BSCW Shared Workspace system. In Proceedings of the 4th International WWW Conference, Boston, MA, December 1995.
2. Chalk P, Webworlds-Web-based modeling environments for learning software engineering, Computer Science Education, vol.10, no.1, 2000, pp.39-56.
3. Dossick, S.E. and Kaiser, G.E. Distributed Software Development with CHIME, In Proceedings of the 1999 ICSE Workshop on Software Engineering over the Internet, http://sern.cpsc.ucalgary.ca/~maurer/ICSE99WS/ ICSE99WS.html.
4. Ebert, J., Siittenbach, R., Uhe, I.: Meta-CASE in practice: A case for KOGGE. In Olive, A., Pastor, J.A., eds.: Proceedings 9th International Conference on Advanced Information Systems Engineering (CAiSE'97). LNCS 1250, Barcelona, Spain, Springer-Verlag (1997) 203-216.
5. Ferguson, R.I., Parrington, N.F., Dunne, P. Hardy, C., Archibald, J.M. and Thompson, J.B. MetaMOOSE - an Object-Oriented Framework for the construction of CASE tools, Information and Software Technology, vol. 42, no. 2, January 2000.
6. 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.
7. Graham T.C.N., Stewart, H.D., Kopaee, A.R., Ryman, A.G., Rasouli, R. A World-Wide-Web architecture for collaborative software design, In Proceedings of the Ninth International Workshop on Software Technology and Engineering Practice (STEP '99), IEEE CS Press, 1999, pp.22-29.
8. Green, T.R. Cognitive dimensions of notations, People and Computers V, Sutcliffe, A. and Macaulay, L. Eds, Cambridge University Press, 1989.
9. Grundy, J.C., Mugridge, W.B. and Hosking, J.G. Constructing component-based software engineering environments: issues and experiences. Information and Software Technology 42, 2, January 2000, pp. 117-128.
10. Iivari, J., Why are CASE tools not used?, CACM, vol. 39, no. 10, 1996.
11. Kelly, S., Lyytinen, K., and Rossi, M., Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, In Proceedings of CAiSE'96, Lecture Notes in Computer Science 1080, Springer-Verlag, Heraklion, Crete, Greece, May 1996, pp. 1-21.
12. 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.
13. Lyu, M. and Schoenwaelder, J. Web-CASRE: A Web-Based Tool for Software Reliability Measurement, Proceedings of International Symposium on Software Reliability Engineering, Paderborn, Germany, November, 1998, IEEE CS Press.

14. 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.
15. Maurer, F. and Holz, H. Integrating Process Support and Knowledge Management for Virtual Software Development Teams, Annals of Software Engineering, vol. 14, no. 1-4, 2002, pp. 145-168.
16. Maurer, F., Dellen, B., Bendeck, F, Goldmann, S., Holz, H., Kötting, B., Schaaf, M. Merging project planning and web-enabled dynamic workflow for software development, IEEE Internet Computing: Special Issue on Internet-Based Workflow, May/June 2000.
17. Quatrani, T. and Booch, G. Visual Modelling with Rationa Rose™ 2000 and UML, Addison-Wesley.
18. Robbins, J., Hilbert, D.M. and Redmiles, D.F. Extending design environments to software architecture design, Automated Software Engineering 5 (July 1998), 261-390.
19. Stoeckle, H., Grundy, J.C. and Hosking, J.G. Approaches to Supporting Software Visual Notation Exchange, In Proceedings of the 2003 IEEE Conference on Human-Centric Computing, Auckland, New Zealand, October 2003, IEEE CS Press.
20. Sun, J., Dong, J.S., Liu, J. and Wang, H., An XML/XSL Approach to Visualize and Animate TCOZ. In Proceedings of the 8th Asia-Pacific Software Engineering Conference, Macau SAR, China, December 2001, IEEE Press, pp. 453-460.
21. Biddle, R., Noble, J. and Tempero, E. Lightweight Web-based Tools for Usage-Centered and Object-Oriented Design, Technical report, Victoria University of Wellington, New Zealand, 2002.
22. Grundy, J.C. and Hosking, J.G. Software Tools, Wiley Encyclopaedia of Software Engineering, 2nd Edition, Wiley InterScience, December 2001. *PDF*
23. **"**ABC TO METACASE TECHNOLOGY"**,** available from MetaCase Consulting, Jyväskylä, Finland, http://www.metacase.com/papers/ABC_to_metaCASE.pdf
24. Furguson, R.I., Parrington, N.F., Dunne, P., Archibald, J.M. and Thompson, J.B. MetaMOOSE – an Object-oriented, Framework for the Construction of CASE Tools, *Proceedings of CoSET'99*, Los Angeles, 17-18 May 1999, University of South Australia, pp. 19-32.
25. Simply Objects^{TM}, available from http://www.adaptive-arts.com/prod_downloads.htm
26. Reps, T. and Teitelbaum, T., Language Processing in Program Editors, *Compute*r, 20 (11), 1987, pp. 29-40.
27. Ratcliffe, M., Wang, C., Gautier, R.J., and Whittle, B.R. Dora - a structure oriented environment generator, *IEE Software Engineering Journa*l, **7** (3), 1992, pp. 184-190.
28. Ewan Tempero, James Noble, Robert Biddle, "Delegation Diagrams: Visual Support for the Development of Object-Oriented Designs", Technical Report, University of Wellington, March 2003.
29. Stephen Asbury, Scott R. Weiner. Developing Java Enterprise Applications, 2th Edition, New York: J. Wiley, c2001
30. "Simple Object Access Protocol (SOAP) V1.2", A technical recommendation standard of the W3C, 2003, available from http://www.w3.org/TR/SOAP/
31. Fowler, M., Scott, K. UML Distilled. Addison-Wesley, 1996.

32. Sommerville, I. Software Engineering, 5th Edition, Addison-Wesley, 1996.
33. Bruegge, B., Dutoit, Allen H. Object-Oriented Software Engineering: Conquering Complex and Changing Systems, Upper Saddle River, NJ: Prentice Hall, 2000.
34. Duignan, M., Biddle, R., & Tempero, E. (2003), Evaluating Scalable Vector Graphics for use in Software Visualization, in proceedings of The Australasian Symposium on Information Visualization, Adelaide, 2003. Conferences in Research and Practice in Information Technology, Vol 24. Tim Pattison and Bruce Thomas, Eds.
35. Terry Quatrani, "Designing the system architecture", Chapter 11 in Visual Modeling with Rational Rose and UML.
36. On-line Tutorials from the Computer Science department of UOA, New Zealand, available on http://www.cs.auckland.ac.nz/compsci335s2t/archive/2002/tutorials/part1/
37. Hanna, Phil., JSP: the complete reference, Osborne/McGraw-Hill, 2001.
38. Govind Seshadri, "Understanding JavaServer Pages Model 2 architecture: Exploring the MVC design pattern", available from http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html
39. Scalable Vector Graphics (SVG) v1.2, A technical recommendation standard of the W3C, Nov. 2003.
40. Jakarta Tomcat, available from http://Jakarta.apache.org/tomcat
41. Sun JavaServer Pages, available from http://java.sun.com/products/jsp/
42. Sun Java Remote Method Invocation (RMI), available from http://java.sun.com/products/jdk/rmi/
43. CORBA website, OMG, "Corba Basics", available from http://www.omg.org/gettingstarted/corbafaq.htm
44. David Reilly, "Java RMI & CORBA: A comparison of two competing technologies", Nov. 1998, available from http://www.javacoffeebreak.com/articles/rmi_corba/
45. Norman Walsh, "What is XML?", Oct. 1998, available from http://www.xml.com/pub/a/98/10/guide1.html#AEN58
46. J2EE JavaServer Pages Overview, http://java.sun.com/products/jsp/overview.html
47. Marty Hall, Core SERVLETS and JAVASERVER PAGES^{TM} , Sun Microsystems Press and Prentice Hall, May 2000, available from http://pdf.coreservlets.com/
48. World Wide Web (W3C), "Overview of SGML Resources", http://www.w3.org/XML/
49. "XML introduction", available from http://www.softwareag.com/thexmlacademy/denmark/XML_intro.htm
50. T. Houihane, O. Eshahawi, "The Advantages & Disadvantages of XML", Dec. 1999, http://ntrg.cs.tcd.ie/mepeirce/Dce/99/xml/2_Merits_Demerits_&_how_it_works.htm
51. John Wurtzel, "GIF vs. JPEG", July 1997, available from http://hotwired.lycos.com/webmonkey/geektalk/97/30/index3a.html?tw=design
52. "The GIF Image Format", available from http://users.abac.com/dmfair/webcontest/Weisberg/FileFormatsPages/gif.html
53. Marty Hall, Session Tracking, 1999, available from

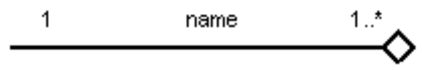
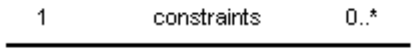
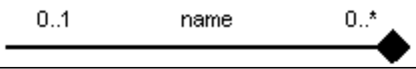
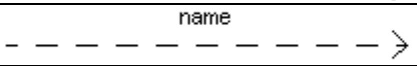
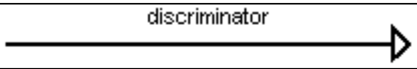
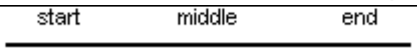
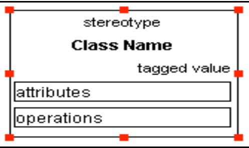
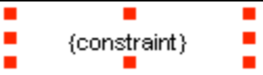
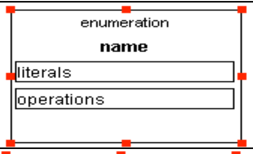
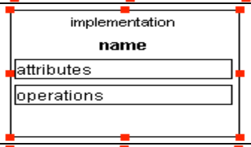
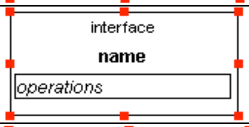
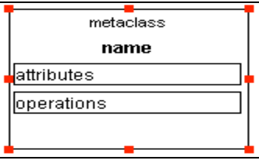
http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/Servlet-Tutorial-Session-Tracking.html

54. RMI specification, Sun Microsystems Inc., available from http://java.sun.com/j2se/1.4.1/docs/guide/rmi/spec/rmiTOC.html
55. John Zukowski, "Delivering Dynamic Images from JavaServer Pages (JSP) Technology", August 21, 2001, http://java.sun.com/developer/JDCTechTips/2001/tt0821.html#tip2
56. "Servlet How To - How to use the JASHist plotting widget in a Java Servlet", July 26 2000, http://www-sldnt.slac.stanford.edu/jas/Documentation/howto/servlet/default.shtml
57. Matti Kotsalainen, "How to I create dynamic GIFs for my JSP?", http://wwwmath.uni-muenster.de/math/inst/num/Vorlesungen/akt_web/Literatur/JSP%20FAQ.htm
58. "Client/Server-side image maps", available from http://curry.edschool.virginia.edu/go/WebTools/Imagemap/CSIM_SSIM.html
59. The ACME laboratories homepage, http://www.acme.com
60. "Appendix: Base 64 Encoding", http://www.freesoft.org/CIE/RFC/2065/56.htm
61. Vincent Hardy, "Scalable Vector Graphics (SVG): An Executive Summary", http://wwws.sun.com/software/xml/developers/svg/
62. "SVG overview", http://www.usbyte.com/SVG/svg_overview.htm
63. Chengyuan Peng, "SCALABLE VECTOR GRAPHICS (SVG)", 2000, available from http://mia.ece.uic.edu/~papers/WWW/MultimediaStandards/svg.pdf
64. IBM, "What is SVG -Vector vs. rasterized graphics", available from https://www6.software.ibm.com/developerworks/education/x-svg/x-svg-2-1.html
65. "ECMAScript Language Specification", http://www.el-mundo.es/internet/ecmascript.html
66. Adobe Systems. SVG zone. http://www.adobe.com/svg/basics/intro.htm
67. The Apache Software Foundation, Batik svg toolkit, http://xml.apache.org/batik
68. KDE. KSVG. http://svg.kde.org/, 2002
69. Mozilla organization. Mozilla svg project. http://www.mozilla.org/projects/svg
70. Corel, Corel smart graphics, http://www.corel.com/smartgraphics
71. Adobe Systems, Adobe SVG Viewer 3.0, http://www.adobe.com/svg/viewer/install/main.html
72. Adobe Systems , Adobe SVG Viewer 6.0, http://www.adobe.com/svg/viewer/install/beta.html
73. "Current Support for SVG Adobe SVG Viewer 3.0", http://www.adobe.com/svg/indepth/pdfs/CurrentSupport.pdf
74. Antoine Quint, SVG Tips and Tricks: Adobe's SVG Viewer, July 03, 2002, http://www.xml.com/pub/a/2002/07/03/adobesvg.html
75. Antoine Quint, "SVG: Where Are We Now?", Nov. 21 2001 http://www.xml.com/lpt/a/2001/11/21/svgtools.html
76. Formatting Objects Processor(FOP), available from http://xml.apache.org/fop/
77. ILOG JViews Component Suite, available from http://www.ilog.com/products/jviews/
78. Get URL syntax page, http://www.protocol7.com/svg-wiki/ow.asp?GetUrl
79. T.R.G. Green, M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework", Journal of Visual Languages and Computing 1996(7), pg 131-174, 1996.

Appendix A: Tool_UML Notations

Tool_UML is a UML tool specified using the Pounamu Meta-CASE tool (by Karen Liu in her 780 project). This is a simplified tool since only a subset of UML notations are defined for drawing use case diagrams, class diagrams, sequence diagrams, and collaboration diagrams. Detailed information about these notations, including meta-Model types and visual representations, are presented in the following tables.

Table A-1 Class Diagram (19 elements)

UML Notation	Meta-Model Element Type	Visual representation
Aggregation	AT_Aggregation	
Association	AT_Association	
Composition	AT_Composition	
Dependency	AT_Dependency	
Generalization	AT_Generalization	
Link	AT_Link	
Class	ET_Class	 stereotype Class Name tagged value attributes operations
Constraint	ET_Constraint	
Enumeration	ET_Enumeration	
Implementation	ET_Implementation_class	
Interface	ET_Interface	
Metaclass	ET_Metaclass	

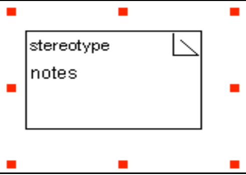
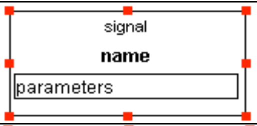
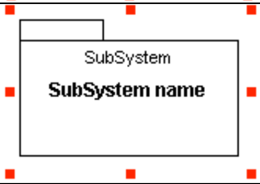
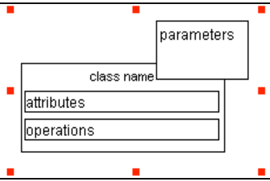
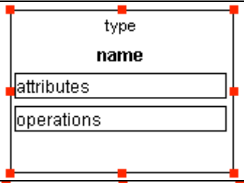
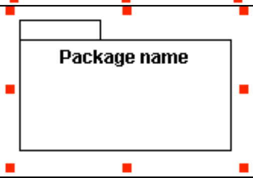
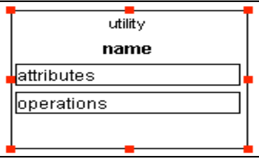
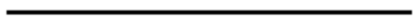
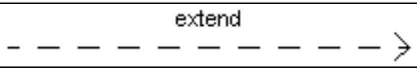
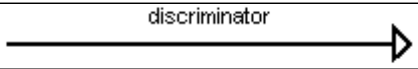
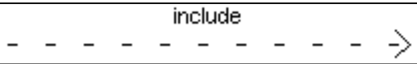
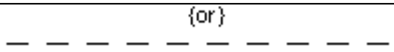
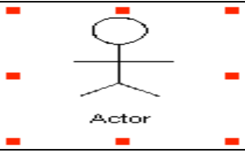
Note	ET_Note	
Signal	ET_Signal	
Subsystem	ET_Subsystem	
Template class	ET_Template_class	
Type	ET_Type	
Package	ET_Uml_package	
Utility	ET_Utility	

Table A-2 Use Case Diagram (12 elements)

UML Notation	**Meta-Model Element Type**	**Visual representation**
Use_case_communication	AT_Communication	
Extend	AT_Extend	
Gneralization	AT_Generalization	
Include	AT_Include	
Or_Constraint	AT_Or_constraint	
Actor	ET_Actor	

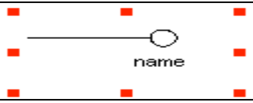
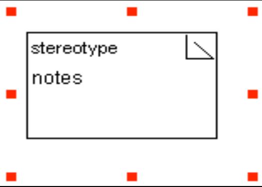
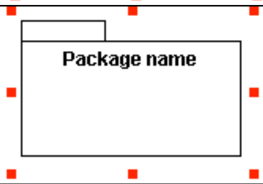
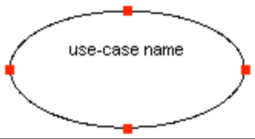
Interface	ET_Interface	
Note	ET_Note	
Package	ET_Uml_package	
Use Case	ET_Use_Case	

Table A-3 Sequence Diagram (9 Elements)

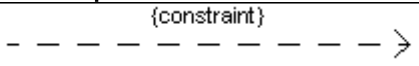
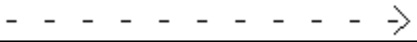
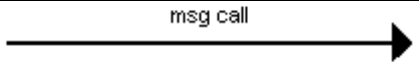
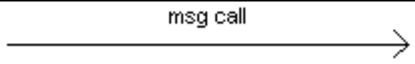
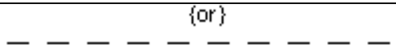
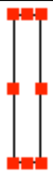
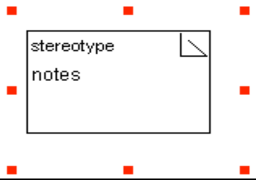
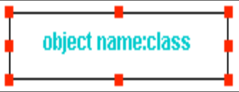
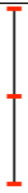
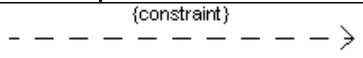
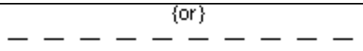
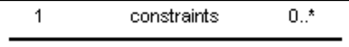
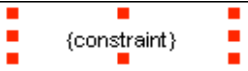
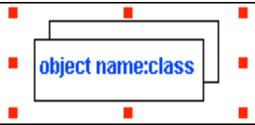
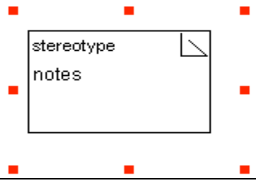
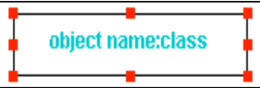
UML Notation	Meta-Model Element Type	Visual representation
Constraint	AT_Constraint	
Seq_return	AT_Msg_return	
Seq_msg_call	AT_Msgcall	
Seq_msg_call_async	AT_Msgcall_async	
Or_Constraint	AT_Or_constraint	
Activiation	ET_Activation	
Note	ET_Note	
Object	ET_Object	
Lifeline	ET_Object_lifeline	 Exported location and size for user adjustment.

Table A-4 Collaboration Diagram (7 Elements)

UML Notation	**Meta-Model Element Type**	**Visual representation**
Constraint	AT_Constraint	
Or_Constraint	AT_Or_Constraint	
Association	AT_UML_Collaboration_association_role	
Constraint	ET_Constraint	
Multi-object	ET_Multiobject	
Note	ET_Note	
Object	ET_Object	

Appendix B: Questionnaire

We would appreciate it if you can take the time to answer the following questions. The purpose of this questionnaire is to help us to gain feedback about using our GIF- and SVG-based thin-client Pounamu tool as compared against its thick-client version. These questions are designed to focus on the areas we are interested in, however, any comments you have when you do evaluations are welcome.

All the information you provide is confidential. Your name is not stored with this questionnaire, and the information you provide will not be used for any other purpose.

Pre-test Questionnaire

1. **Personal information**

Gender	☐ Male	☐ Female
Education	Bachelor Î High School Î	PhD Î Master Î
Major	☐ Computer Science	☐ Engineering
	☐ Business & Economics	☐ Other ________________

Job Title____

2. **Do you have any experience with any software engineering CASE or Meta-CASE tool? If yes, please specify what kinds of tools and their strengths & weaknesses?**

No ☐

Yes ☐ ____

3. **Do you have any experience of using Pounamu meta-CASE tool? If yes, specify how do you feel about it?**

No ☐

Yes ☐ ____

Post-test Questionnaire

1. **From your perception, how easy to use the GIF-based Pounamu/thin tool vs. its thick-client version?**

____

____

2. **Are GIF- and SVG-version of PounamuView diagrams readable in a web browser?**

____

____

3. **In general, is it easy to use the GIF version of the Pounamu/thin tool to edit a Pounamu view diagram?**

   Very easy Very difficult

   1 î 2 î 3 î 4 î 5 î 6 î 7 î

4. **For each of the following main editing functionalities, please specify the features that you feel easy/hard to use in the GIF version of the thin-client tool?**

Add association-type connector:	Easy to use:
	Difficult to use:
Move entity-type shape:	Easy to use:
	Difficult to use:
Move association-type connector:	Easy to use:
	Difficult to use:
Delete entity-type shape:	Easy to use:
	Difficult to use:
Delete association-type connector:	Easy to use:
	Difficult to use:
Resize entity-type shape:	Easy to use:
	Difficult to use:
Set entity or association properties:	Easy to use:
	Difficult to use:

5. **In general, is it easy to use the SVG version of the Pounamu/thin tool to edit a Pounamu view diagram?**

   Very easy Very difficult

   1 î 2 î 3 î 4 î 5 î 6 î 7 î

6. **How do you feel edit buffering functionality provided in the SVG version of the Pounamu/thin tool? Is it useful or just confuses you?**

7. **From your perceptions, which one of two versions of the thin-client user interfaces is more usable? Please state some reasons why you say so?**

GIF version î SVG version î

Reasons:

8. **How do you feel the system response time when performing your edit tasks via the two versions of thin-client user interfaces, are they fast enough?**

GIF version:

SVG version:

9. **Write down your suggestions on how to improve our GIF- and SVG- based Pounamu thin-client user interfaces.**

GIF version:

SVG version:

Appendix C: Evaluation Task List

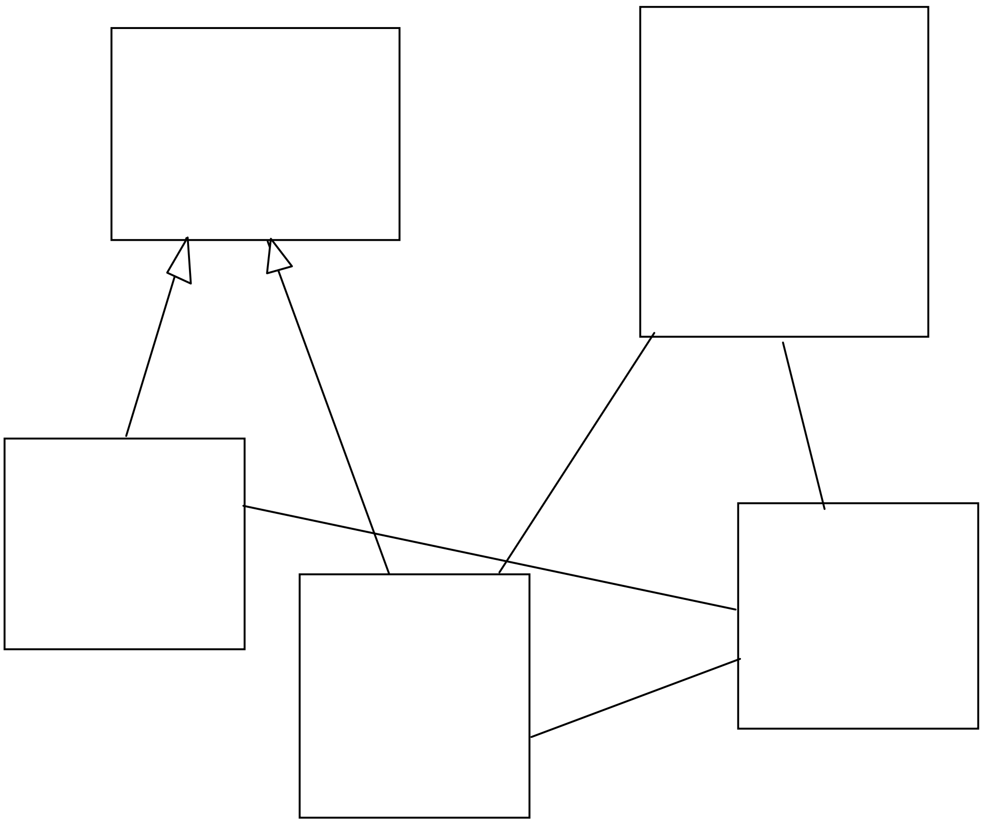
The tasks are designed to see how different people would feel about using the thick client user interface of Pounamu and its thin-client counterparts. For this purpose, the following tasks need be done multiple times, and the details are stated as below:

(1) Task1 is done three times, that is, do this task via Pounamu/thick and the GIF- and SVG-based Pounamu/Thin, respectively. In addition, please switch buffering facility off when you perform it via the SVG Pounamu/Thin.

(2) Task2 is done via the SVG version of the thin-client tool with buffering facility switched on.

Task1:

In this task, you are asked to model and document OOA class diagram of the Video Store system. To make your life easier, a modeling project named Model_VideoRental that takes Tool_UML as its underlying tool has been set up for you in Pounamu, and a view with type VT_Class_diagram and name VT_Class_diagram_ConceptualModel has also been created for you. Of course, currently this PounamuView is just a blank canvas, what you need do is to add model elements to this view so that the final version of this PounamuView looks similar to the following diagram.



Task2:

In this task, you are asked to evaluate multiple-editing functionality enabled in the SVG-based Pounamu/thin tool. Thus you need to switch edit buffering facility on by clicking the left menu item before performing any edit command on the PounamuView diagram specified previously.