INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

Cross-Platform User Interfaces

by
Dorin Sandu

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

December 1999

© Copyright
1999, Dorin Sandu

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-48444-0

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

Cross-Platform User Interfaces

submitted by

Dorin Sandu

A handwritten signature in black ink, appearing to read 'Dorin Sandu', written over a horizontal line.

(Director, School of Computer Science)

A handwritten signature in black ink, appearing to read 'Dwight Denno', written over a horizontal line.

(Thesis Supervisor)

Carleton University
December 1999

Abstract

Cross-platform user interfaces are time-consuming to develop and even more time-consuming to maintain. To solve this problem we introduce the Interface Markup Language, a language we developed to represent platform-independent user interfaces, along with a system of software patterns that provide a consistent way for user interfaces to interact with application models. To support our approach, we have implemented translators that convert VisualWorks Smalltalk user interfaces to the Interface Markup Language, and others that convert from this format to operating system resources, HTML forms, Java Swing and AWT frameworks.

Acknowledgements

I would like to thank my supervisor, Dr. Dwight Deugo, for his guidance, motivation and support. His work ethic and energy has been an inspiration to me. Also, I would like to thank Darrell Ferguson, Mike Valenta, and the rest of the COOP gang for putting up with me during the final stretch of this project. Their encouragement, understanding, and taste in music leave a lot to be desired but they have an excellent sense of humor.

Contents

Abstract	iii
Acknowledgements	iv
1 Introduction	1
1.1 Contributions	2
1.2 Overview	3
2 Background	4
2.1 GUI Overview	4
2.1.1 Architecture	5
2.1.2 Interaction	7
2.1.3 Construction	10
2.2 GUI Portability	12
2.2.1 C++ Frameworks	14
2.2.2 Smalltalk Frameworks	15
2.2.3 Java Frameworks	17
2.3 Summary	21
3 User Interface Patterns	23
3.1 Introduction	23
3.1.1 Pattern Summary	28
3.2 View Event Handler Pattern	30
3.2.1 Context	30

3.2.2	Forces	30
3.2.3	Problem	31
3.2.4	Solution	31
3.2.5	Consequences	32
3.2.6	Example	33
3.3	Complete Update Pattern	35
3.3.1	Context	35
3.3.2	Forces	35
3.3.3	Problem	36
3.3.4	Solution	36
3.3.5	Consequences	36
3.3.6	Example	37
3.3.7	Variants	38
3.4	Multiple Updates Pattern	42
3.4.1	Context	42
3.4.2	Forces	42
3.4.3	Problem	43
3.4.4	Solution	43
3.4.5	Consequences	44
3.4.6	Example	44
3.5	User Interface Patterns Discussion	46
3.5.1	Known Uses	47
3.5.2	Related Patterns	48
3.6	The Lambda Pattern	48
3.6.1	Context	49
3.6.2	Forces	50
3.6.3	Problem	51
3.6.4	Solution	51
3.6.5	Example	53
3.6.6	Resulting Context	55
3.6.7	Rationale	57

3.6.8	Related Patterns	58
3.6.9	Known Uses	58
3.7	Summary	63
4	Methodology	64
4.1	XML Overview	65
4.1.1	XML Structure	65
4.1.2	XML Linking	68
4.1.3	XML Processing	71
4.1.4	User interfaces in XML	72
4.2	IML Overview	72
4.2.1	Widgets	74
4.2.2	Generic Widgets	75
4.2.3	Reference Widgets	76
4.2.4	Tab order	77
4.2.5	Discussion	77
4.3	IML Translators	78
4.3.1	Smalltalk-to-IML Translator	80
4.3.2	IML-to-Resource Translator	83
4.3.3	IML-to-Forms Translator	84
4.3.4	IML-to-Swing Translator	85
4.3.5	IML-to-AWT Translator	89
4.4	Summary	89
5	Related Work	90
5.1	Windows Resources	91
5.1.1	Discussion	94
5.2	User Interface Markup Language	95
5.2.1	Discussion	101
5.3	Swing Markup Language	102
5.3.1	Discussion	104
5.4	XML-based User Interface Language	104

5.4.1	Discussion	109
6	Conclusion	110
6.1	Future Work	112
A	IML Document Type Definition	113
B	IML TodoList Example	121
B.1	IML Source	122
B.2	TodoListWindow Source	123
B.3	GridLayout Source	128
B.4	GridConstraint Source	131
	Bibliography	133

List of Tables

3.1	Pattern Problems and Solutions.	30
3.2	λ – <i>Function</i> equivalents in different languages.	52

List of Figures

2.1	User interface levels.	5
2.2	VisualWorks widget hierarchy.	15
2.3	VisualAge widget hierarchy.	16
2.4	AWT component peers.	18
2.5	AWT widget hierarchy.	19
2.6	AWT menu hierarchy.	20
3.1	Pattern language dependencies.	28
3.2	ToDoList example.	33
3.3	Multiple ToDoList example.	46
3.4	MVC adaptors.	59
3.5	Selection tool internal state.	60
4.1	XML processor.	71
4.2	IML object model.	79
4.3	IML translator object model.	80
B.1	ToDoList example.	121

Chapter 1

Introduction

Myers and Rosson's survey on user interface programming [28] indicates that, on average, 48% of an application's code relates to the user interface. The average percent of time spent on the user interface is 45% during the design phase, 50% during implementation, and 37% during maintenance. The high cost for design and maintenance is attributed to the need to understand the requirements, build prototypes, and interact with potential users. However, most of the participants in the survey reported that the high cost during the implementation phase is due, in part, to the poor separation between the interface and the application, combined with poor communication between programming languages and poor portability across different windowing systems.

Solutions to these implementation problems exist:

- User interfaces can be separated from the application model by using frameworks that make a clear distinction between the model and view. Such frameworks use the classic *Model-View-Controller* [4] pattern or, more recently, the *Model-View-Presenter* [21].
- User interfaces can be made portable by employing cross-platform frameworks. Such frameworks fall in two categories: those that implement an abstraction layer on top of the platform-native user interface, and those that emulate the look and feel of the native user interface. The former include the Smalltalk for VisualAge interface framework and Java Abstract Windowing Toolkit. Smalltalk VisualWorks and Java

Swing fall into the second category.

However, using cross-platform frameworks may not be desirable. Adding an extra indirection layer results in user interfaces that support only a common subset of the functionality available on all platforms. Also, emulating the look and feel of a platform increases the application size and special care has to be taken that the user interface really looks and behaves like its native counterpart. Most important, these approaches only work within a single programming language, for example, user interfaces cannot be reused between applications implemented in different languages or operating systems.

A better solution to these problems is to define a user interface representation that can be either translated to platform-native code or directly interpreted, much in the same way the Hypertext Markup Language has been used to represent information on the World Wide Web. User interfaces represented in such format definitely promote separation from the application model and can be easily translated between different frameworks implemented in different languages and running on different operating systems.

1.1 Contributions

Our main objective is to simplify the development and maintenance of cross-platform interfaces. This thesis examines the potential of creating a platform-independent user interface format, along with methods by which user interfaces expressed in this format can interact with application models. We have made the following contributions to meet these goals:

- *Interface Markup Language.* We have developed a markup language to express user interfaces in a platform-independent format. This language is implemented using the eXtensible Markup Language [8], or XML, an existent standard for representing information on the web and structured information in general.
- *User interface patterns.* We have developed a system of patterns that can be used to link user interfaces to application models in a consistent way. The *View Event Handler* pattern is used in conjunction with the *Lambda* pattern to handle the events triggered by the interface, and *Complete Update* and *Multiple Update* patterns to update the user interface from the application model.

- *User interface translators.* We have developed a framework that uses the patterns to translate the interface markup language to different toolkits and frameworks. Specific translators convert VisualWorks Smalltalk user interfaces to the markup language, and others convert the markup to operating system resources, HTML forms, Java Swing and AWT frameworks.

1.2 Overview

The thesis is divided into three parts: (i) an overview of user interfaces, (ii) a presentation of the software patterns for interaction between user interfaces and applications along with the description of the Interface Markup Language, and (iii) a summary of similar approaches to the one presented in this thesis.

The first part comprises Chapter 2. In this chapter we briefly review user interfaces and present different frameworks that implement cross-platform user interfaces. The purpose of this chapter is two-fold: first, to acquaint the reader with the topic of user interfaces and second, to identify the user interface characteristics that need to be mapped between platforms.

Chapters 3 and 4 represent the second part of the thesis. In Chapter 3 we present a system of software patterns that provides a consistent way for the user interface to interact with the application. In Chapter 4 we briefly overview the eXtensible Markup Language and describe the design and implementation of the Interface Markup Language along with the concrete translator implementations.

The third part covers Chapters 5 and 6. In Chapter 5 we present similar approaches that use an intermediate representation to implement cross-platform user interfaces. The purpose of this chapter is to compare the approach we proposed in this thesis with other similar approaches. In Chapter 6 we summarize the thesis and outline directions for future work.

Chapter 2

Background

The main purpose of this chapter is to introduce the reader to user interfaces. In Section 2.1, we review their architecture, we describe different ways users can interact with interfaces, and present techniques for constructing user interfaces. The second purpose of this chapter is to identify the user interface characteristics that need to be mapped between platforms. We do this in Section 2.2, where we review a couple of C++, Smalltalk, and Java frameworks that implement cross-platform interfaces.

2.1 GUI Overview

Graphical user interfaces are meant to simplify the interaction between humans and computers. They provide a consistent look and feel across applications, so the user does not have to memorize different sets of interaction techniques for different applications, and generally empower the user by providing task feedback, undo/redo capabilities, help and error handling.

The rest of this section reviews the architecture of user interfaces, describes different ways the user can interact with a computer via a graphical user interface, and overviews tools and techniques that help construct user interfaces.

2.1.1 Architecture

According to Foley [13], applications implement user interfaces by using a combination of features provided in three layers, as shown in Figure 2.1:

Figure 2.1: User interface levels.

Window Management System. This layer acts as a resource manager for the display area and the hardware input devices connected to the system. It manages a list of arbitrarily shaped, most often rectangular, screen areas called windows and, at the same time, it directs the flow of information from the input devices to the appropriate application.

Toolkit. This layer implements widgets such as menus, buttons, scroll bars, and text input fields. These are basic windows provided by the window management system paired with special input and output handling routines used to edit a particular data structure. The widgets serve as the building blocks for graphical user interfaces and are used to edit aspects of application models.

User Interface Management System. While toolkits define the syntax of user interfaces, user interface management systems define their semantics [13]. User interface management systems focus on sequence control, they provide the means to define valid user input sequences and sometimes provide high-end features such as undo/redo managers and user profiles.

A generic window management system is further subdivided into a window system and window manager. The window system is the one that actually manages the windows

whereas the window manager uses the functionality provided by the window system and the graphics package (part of the window system or a separate library) to implement the look and feel of the actual windows.

It is up to the window system to create and destroy windows. When a client makes a request to create a window, the system returns a resource identifier for some memory it allocates to store window position, size, and color information. The caller uses this identifier to open (the window is made visible so the user can interact with it), reposition, resize, modify the foreground/background colors, and paint the window. The window system also decides how certain screen or window parts are redrawn; it either caches the window's look at creation time and redraws it when refresh is needed or it notifies the client to repaint the window. Furthermore, the window system clips windows against each other and performs coordinate transformation between windows, or between windows and the global display.

Window systems usually keep track of windows in either a flat or hierarchical structure. The flat structure is a simple list that contains all the created windows in the system. In a hierarchical structure, the windows can be arbitrarily nested, thus resembling the tree structure promoted by the *Composite* [14] pattern. In such systems, the window system associates parent and children information with each window; windows with no parent are called top or main windows. Hierarchical structures relieve the clients from having to maintain explicit window relationships. Such systems also propagate some actions performed on a window to its children, for example, when a main window is destroyed, its children are destroyed too.

Normally the user has control over the size, position, and depth order of the top windows. The window manager captures user input in specially designated areas of the main windows, usually the title bar and the frame, and makes the appropriate calls to the window system. Sometimes, the window manager maintains the spatial relationship between windows with constraints [26, 27]; two common window layouts are tiled and cascaded [22].

In some window management systems, there is a clear separation between the window manager and the window system [13]. In such systems, the window system is sometimes called a server and the window manager a client. One such system is XWindows in which the window management system supports a true client-server architecture; for example, the window manager appears to the window system just like another client program. By

contrast, in systems such as Windows and Macintosh, there is a closer relationship between the manager and the system than between a client and a server. The windowing services are provided by some portion of the operating system, and therefore are dependent on the architecture and the available resources of the machine.

A clear distinction between the manager and the window system has both advantages and disadvantages. True client-server architectures allow the window manager to be replaced, thus changing the look and feel. Also, such systems can be distributed across networks (to allow some expensive computation to be performed on a powerful computer and have the user control it through a graphical user interface on a terminal). On the other hand, these benefits are also the main disadvantages. Distribution across a network can introduce communication delays and making the window manager replaceable has the side-effect that applications with different look and feel can run on the same system. This defeats one purpose of a graphical user interface, which is trying to provide a consistent look and feel in order to minimize the command memorization required to interact with the computer.

2.1.2 Interaction

User interface input should be designed to be independent of the many hardware devices connected to the system. Consequently, many devices can be used to perform similar actions, for example, a user can scroll information by either pressing the arrow buttons on the keyboard or by using the mouse to manipulate the scrollbars.

Devices are classified according to the type of input, rather than the hardware designation, such as mouse or keyboard. The most common inputs can be classified as locator, stroke, string, valuator, choice, and pick [16, 13]:

Locator devices are used by a program to read a single coordinate position. This can be achieved with physical devices such as mice, keyboards, and digitizers. Usually the screen cursor is moved to the desired location after which an action, such as a button press, is performed to capture that coordinate position.

Stroke devices are used to read a stream of coordinates. They are equivalent to multiple calls to a locator device. As an example, consider the continuous motion of a mouse:

as the cursor is moved across the screen, a stream of coordinate positions is generated and read by the program.

String devices are used to input text. The keyboard is the most frequently used device to enter text into a computer but, in the case of hand-held computers, the pen, or stylus, is used as well. With a pen, the individual characters are drawn on the screen and converted to text by a pattern recognition program.

Valuator devices are used to input numeric values. Such devices include control dials and sliders, or the keyboard numeric keypad as an alternative. Valuator devices are usually implemented in software as scrollbar and dial widgets.

Choice devices are used to input a selection. Such devices are implemented both in hardware (the keyboard function keys or separate button devices that allow only a single button to be pressed at a time) and software (menus, toolbars).

Pick devices are used to select an object on which further action needs to be taken. Usually, the user moves the screen cursor on top of the desired object and performs an action, for example a button press, that will trigger a search to locate the object under cursor. If the search is too expensive to be performed, alternative pick methods exist: a stroke device can be used to highlight objects sequentially, or use the keyboard to type in the name of the object to be selected.

An application can acquire input in three different modes [16]:

Request Mode In request mode, it is the application that asks the device for input, and any other processing is suspended until the required values are received. In this operation mode, either the application waits for input, or the device waits for requests.

An application running in request mode has the disadvantage that it has to wait for the input device to release control in order to proceed. Also, during the request time, proper feedback to the user cannot be provided. The only advantage of the request input mode is that it is fast. This is because the input device is guaranteed not to be interrupted until it finishes sending the input.

Sample Mode In sample mode, both the application and hardware device operate simultaneously and, when the program needs information, it queries the device for the current values.

An application running in this mode may overlook important data. For example, while processing input, significant user actions may never be registered. Also, the sequence of events is not guaranteed to be accurate. However, by using this mode, an application can provide dynamic feedback to the user by continuously sampling the hardware devices, interpreting the sampled data, and displaying the results.

Event Mode In event mode, the input is initiated by the hardware device. The program and input device operate concurrently, and data is delivered directly to the application.

Event mode guarantees that all data is gathered and is in proper order. This is done via a queue: input is placed by all hardware devices into the program queue where is picked up by the program, which runs in an infinite loop continuously checking the queue for information. The queue can be optimized to filter the input. For example, patterns in the input can generate new data (two single click events that occur within a short period of time can be interpreted as a double click event) or continuous identical data can be discarded (multiple scroll events can be interpreted as a single scroll event of a bigger amount).

Open, close, and resize events are generated by the corresponding functions in the window system. Additionally, most window systems generate window enter/leave events when the display cursor enters/leaves the window. Usually an event consists of the event type, window identifier, a time-stamp and any event-specific information that is required (for example, a mouse click event will contain the x and y coordinate of the screen cursor).

Events must be routed by the window system to the appropriate application queue. This is done in two ways: either move-to-type or click-to-type. In the first case, the event goes to the application whose window contains the cursor – the user has to position the screen cursor on top of the window that should receive input. In the second case, the event goes to a designated window, usually the top most one – the user has to click the window to bring it on top of all the other windows.

When an event occurs, the windowing system must decide which window should get the event. While this is easy for flat window systems, hierarchical systems have problems deciding which window in the hierarchy should receive the event. In XWindows, it is the lowest window in the tree containing the screen cursor that receives the event; if that window rejects the event, then it is promoted up the tree until some other widget processes it. On the other hand, in MacOS, the event is passed to the highest window which will forward it to one or more of its children if necessary. This technique is much more flexible because messages passed this way can be filtered.

A program can make use of several input devices at once, where each device can be operating in one of the three modes specified above. Many devices can be used in sample and event mode at the same time but only one device is allowed to work in request mode. Today's user interface systems use event mode the most.

2.1.3 Construction

In this section we investigate how user interfaces can be constructed. There are basically four ways, by programming, by using editors, by using higher-level languages, or by example.

The first method is by programming, where the programmer calls routines in the toolkit to create the hierarchy of widgets. This can be done by using either the low-level interface provided by the toolkit, or higher-level application frameworks:

Application Programming Interfaces. These are simple routine libraries, usually built in ROM(the Macintosh toolbox) or directly in the operating system(the Windows API), that provide very low level widget access, so low that the programmer must provide the event-handling loop. These routines do not provide any application structure.

Application Frameworks. Application frameworks actually hide the event loop from the programmer, and can be either procedural or object-oriented. The framework reads the events and calls out various procedures which the application has registered with the library. This way, the framework takes over the burden of managing a complex

event-driven system and also provides an application structure. Sometimes, frameworks are not so flexible because they can only be modified in places that they have been designed to be modified. Examples are Microsoft's Foundation Classes and Borland's Object Windows Library.

Most frequently, rather than writing a program, user interfaces are built using visual editors. Such editors allow the user to instantiate a set of widgets defined in one or more toolkits, set the values of their properties, and arrange them visually inside a container (usually a window or dialog frame). This approach is a bit more restrictive than directly programming the user interface because user interface editors do not allow to specify any dynamic behavior, such as modifying the interface in response to some user action. Such features still have to be programmed by hand.

User interface editors output either an intermediate format (which later can be translated to code), code, or compiled code (called resources on both Windows and Macintosh, these can be linked in with the executable to produce the final version of the application). The user interfaces created by such tools are better because designs can be prototyped before the application code is written. Tool-created user interfaces are also easier to modify, and the generated code is more reliable and consistent than its hand-coded counterpart.

Another way to build user interfaces is to derive them from either the application model or some other specifications. Automatic generation reduces the cost and time of development and promotes experimentation [29]. The generated user interface is more reliable than the hand-coded equivalent, and also more consistent in providing help and error reporting. De Baar [12] explains that application building consists of designing/building both a data model and a user interface that is responsible to present that model to the user. His paper couples application and user interface design by automatically generating the user interface from the data model. This is done by using an inference engine that contains rules for selecting and positioning widgets. This approach is interesting because the same model can be mapped to different user interfaces by simply substituting the rule set in the inference engine. A similar approach is used by Janssen [17] who automatically generates user interfaces from application models by using an expert system to generate the static layout of the user interface and petri nets to specify dynamic behavior.

User interfaces can also be built by using higher level programming languages. For

example, in Micky [18], user interfaces are defined as record declarations in an extended version of the Pascal language. The record data types are later used to determine the type of the widget used.

Other systems, such as LEMMING, described by Olsen [20] allow the creation of widgets by example. The tool distinguishes between the presentation and control part of widgets and provides mechanisms for users to change the presentation part and automatically learns the mapping between the presentation and control parts.

2.2 GUI Portability

The most obvious solution to port graphical user interfaces to different platforms is to partition the application into non user interface and user interface components, where the latter are implemented using the platform-native toolkit. Then only the user interface components need to be rewritten for each new target platform. This method places the burden on the programmer to divide the code into application model and views and, in order to avoid this, special frameworks have to be used. Such frameworks use either a layered or emulated approach.

Frameworks based on the layered approach use a *Bridge* [14] pattern to decouple the user interface abstraction from its implementation, so that the implementation can vary based on the platform that the application is running on. Such frameworks use the platform-native toolkit and therefore maintain a consistent look and feel with the rest of the applications running on the same platform. However, most of the time, these frameworks only provide the intersection of the features of all supported platforms.

Frameworks based on the emulated approach provide proprietary toolkits that implement the native look and feel as closely as possible. The main disadvantage is that the code needed to duplicate the native toolkit adds a lot to the size of the framework. Also, most of the frameworks using this approach do not provide an accurate look and feel, and usually programs using such frameworks stand out from the rest of the applications. One advantage is that such frameworks are usually faster than their layered counterpart, and allow for applications to be developed and tested on a single platform.

The next sections provide an overview of existing frameworks that implement cross-platform user interfaces. The frameworks are categorized based on the implementation language along with the following criteria:

Structure. The widgets are assembled in a hierarchical format which resembles the tree structure promoted by the *Composite* [14] pattern. The window system usually associates parent and children information with each widget, and this information is used to traverse the widget tree and sometimes propagate events through the inheritance path.

Properties. Widgets are data-centered, they can be configured by changing their properties values without implementing any new code. Properties can be changed either internally by the application, or externally, by the user through interaction with the interface.

Layout. The widgets keep track of their position and size relative to their parents. When the position or size of the parent changes, the widgets modify their position and size according to a layout algorithm. The simplest layout is absolute positioning, where the widget never changes position or size. Others include layout managers (Java), grids (VisualWorks), struts and springs [5].

Updating. When properties are changed, widgets provide default behavior: first the attributes values are updated, and then the display is updated to reflect the new state of the widget. Additionally, the application may be notified of the changes, if it previously registered an interest in the widget (via callbacks, events, etc.).

2.2.1 C++ Frameworks

Amulet

Amulet [27] is a user interface development environment for C++ that is portable across XWindows, Microsoft Windows, and the Macintosh. It achieves such portability by providing different platform look-and-feel policies that can be assigned at run-time. Amulet simplifies the creation of graphical, direct-manipulation user interfaces by using a prototype-instance object model to implement constraints, undo/redo facilities, animation, gesture recognition, and a full set of widgets.

The prototype-instance object system is similar to the C++ object model. Widget properties are stored in slots, where each slot plays the role of an instance variable for the prototype. An important difference between C++ and Amulet objects is that Amulet allows the dynamic creation of slots. An Amulet program can add and remove slots from an object as needed. In a C++ object, the value of the class's data can be modified at runtime, but changing the amount of data in an object requires recompiling the source code.

When Amulet prototypes are instantiated, an inheritance link is established between the prototype and the instance. Inheritance allows instances to use slots in their prototypes without setting those slots in the instances themselves. For example, if an Amulet program sets the fill style of a rectangle to be gray, and then creates an instance of that rectangle, then the instance will also have a gray fill style.

Layout is performed with constraints. With constraints, the value of one slot always depends on the value of one or more other slots; in the case of layouts, arbitrary constraints can be set between the slots representing the position and size of the widgets. An important point about constraints is that they are always automatically maintained by the system. They are evaluated once when they are first created, and then they are re-evaluated when any of their dependencies change. If several objects depend on the top of a certain rectangle, then all the objects will change position whenever the rectangle is moved.

There are two ways to interact with widgets. The first is to define a constraint that depends on the value of the widget, and the second is to define a method to be executed by the widget's command object whenever the user activates the widget. Both ways can be used to keep the application and its user interface synchronized.

2.2.2 Smalltalk Frameworks

VisualWorks

VisualWorks Smalltalk achieves user interface portability by emulating the platform look and feel. Different looks and feels are implemented by concrete subclasses of the `LookPolicy` and `FeelPolicy` abstract classes.

Figure 2.2: VisualWorks widget hierarchy.

Structurally, user interfaces in VisualWorks are composed of `VisualComponent` objects (part of the hierarchy is shown in Figure 2.2). A window contains `SimpleComponent` (static components like labels and group boxes) and `View` objects (like buttons and scrollbars that actually have a model and controller) which can be nested inside `CompositePart` objects. This model is similar to the *Composite* [14] pattern where simple components and views play the role of leaf objects, and composite parts play the role of composite objects.

VisualWorks parts can be positioned within other parts by specifying a `Rectangle`. To specify relative positions however, it is necessary to use either a `LayoutOrigin` or `LayoutFrame`. The former is used when you are only concerned with the position of the widget, and the latter when you are concerned with both the position and the size of the widget relative to its container. The `LayoutOrigin` contains a `leftFraction` and `topFraction`, with possible values between 0 and 1, that specify a relative point inside

the parent container. Additionally, a `leftOffset` and `topOffset` are used to compute the absolute position. The `LayoutFrame` has similar attributes for the origin, with the additional ones for the corner `rightFraction`, `bottomFraction`, `rightOffset`, and `bottomOffset`.

Updating is done using the *Model-View-Controller* [4] paradigm. Propagating changes used to be done via the dependency mechanism employed by all `Object` subclasses but, recently, `VisualWorks` has introduced a new event mechanism. According to the new event mechanism, the models, views, and controllers generate events that the application can subscribe to.

VisualAge

The IBM Smalltalk user interface framework provides a set of *common widgets* to implement a consistent look and feel across platforms. The interface for these widgets follows the Motif guidelines and each widget is either mapped to a native widget if the widget is available, or is emulated if the widget is not present on the native platform.

Figure 2.3: VisualAge widget hierarchy.

VisualAge Smalltalk widgets are divided into `CwBasicWidget` and `CwExtendedWidget` types, where the latter are extensions of the basic widgets to support visual programming. The basic widgets are further classified into `CwPrimitive`, `CwComposite`, and `CwShell` widgets, as shown in Figure 2.3. Shell widgets are usually top windows and dialogs, whereas primitive and composite widgets serve as user interface building blocks.

The difference between primitive and composite widgets is that a composite widget manages one or more other primitive or composite widgets. Therefore, a VisualAge user interface resembles a tree internally. The root of the tree is usually a shell widget, the internal nodes are made of composite widgets, and the leaves are primitive widgets.

Common and extended widgets can be configured via attributes to look and feel differently. VisualAge calls these attributes *resources* and associates an *access designator* with every resource to indicate when the resource can be accessed. A resource designator of <c> indicates that the attribute can be set when the widget is created, a <s> indicates that the attribute can be accessed at anytime, and a <g> indicates that the attribute can be retrieved only after the widget is created.

Composite widgets are responsible for laying out the widgets they manage according to rules that the user interface developer specifies. Every widget has a set of rules that indicate how it should resize when the parent widget changes size. The rules can be attached to the top, bottom, left, right, or the four corners of each widget and must specify an anchoring point. The anchoring point can be a fixed position inside the parent container, can be one of the top, bottom, left, right, or the four corners of either the parent container or any other sibling widget.

User interface updating in VisualAge Smalltalk is based on the familiar *Model-View-Controller* [4] paradigm combined with event and callback handlers. Events are used to handle low-level interactions with the mouse and keyboard, whereas callbacks are used to handle high-level interactions between the application and the widgets.

2.2.3 Java Frameworks

The AWT Framework

The Abstract Windowing Toolkit framework provides widgets with a platform independent programming interface but with a platform dependent look and feel. To achieve this kind of flexibility, AWT implements platform independent *components* that in turn make use of platform dependent *peers*. For example, in the case of the `Button` widget, AWT implements a peer for every platform that Java runs on, in this case a `ButtonPeer` for

Windows, MacOS, and Motif, as shown in Figure 2.4. It is up to the peers to call the appropriate platform functions to create, manage, and destroy the actual widgets and deal with user input.

Figure 2.4: AWT component peers.

The components hierarchy is shown in Figure 2.5:

An AWT user interface is structured as component tree, where the root node is an instance of either a Window or Dialog class. The inner and leaf nodes of the tree can be instances of any Component subclasses, although Window and Frame instances are not recommended. This way, Button, Canvas, Checkbox, Choice, Label, List, Scrollbar, and TextComponent can be considered *primitive components* and Window, Dialog, Panel, and ScrollPane *composite components* because they usually manage other primitive or composite components.

All components have attributes that can be modified to alter the look and feel of the components without having to write any extra code to implement the required behavior. Subclasses of Component have common attributes for background and foreground colors, fonts, visibility, input blocking, and focus. Each component, in turn, provides a series of specialized attributes, for example, the Checkbox component has an attribute that indicates whether it has been checked or not, and List and TextComponent have attributes to deal with the selection.

The reason that menu components subclass MenuComponent and not Component is because they support a smaller set of attributes compared to regular components. As such, menus cannot change foreground, background colors, and fonts. This is because the native window managers use *desktop themes* to control certain attributes to provide a consistent

Figure 2.5: AWT widget hierarchy.

look and feel of all applications on the desktop.

Since containers keep track of other components, they must provide protocol to manage the size and position of their children. Any `Container` object will lay out its children according to the algorithm specified by its layout manager. By default, `Panel` objects have a `FlowLayout` manager, and `Window` objects have a `BorderLayout` manager. These layout managers can be replaced by `null` in order to have full control over the positioning of the components.

The simplest of all layout managers are `FlowLayout` and `GridLayout`. Flow layouts lay out components from left to right, starting new rows when the combined width of the managed components is bigger than the width of the container. Grid layouts lay out the components on a grid where each component will have the same size.

One of the more complex layout managers is the `GridBagLayout`. This layout will lay out components on a grid of cells, where each component can span more than one cell. It is possible that the rows in the grid have different heights, and that the columns have

Figure 2.6: AWT menu hierarchy.

different widths.

Two special purpose layout managers are `BorderLayout` and `CardLayout`. The border layout uses five areas to hold components: north, south, east, west, and center where all the extra space is placed in the center area. The card layout arranges components to have the same size and position.

Updating the user interface is done by handling events. In the case of AWT, these events have to be handled transparently for every platform Java runs on. This is done with the help of the *peer* objects introduced above. For every component, the corresponding peer object receives a platform specific event, converts it to a higher level Java event and passes it to the component. If the component does not handle it, then it is given to its parent, and so on. If no component in the hierarchy handles the event, it is finally returned to the peer which may perform some default action or ignore it as well. One exception to this rule are the mouse events which are first processed by the peer and then given to the component.

AWT events are based on the generic Java event model. According to this model, events are generated by *event sources* and one or more *event listeners* are notified about the events. Objects have to register as listeners to the events generated by other objects. Such objects are also called *event handlers* because they usually provide a method that gets executed every time an event occurs.

The AWT framework can be extended with proprietary components by subclassing any of the classes in the Component hierarchy. For completely new components, AWT

recommends that `Canvas` be subclassed, and the appropriate methods from `Component` overwritten to handle drawing the component and handling of events.

The Swing Framework

The Swing framework is built as an extension of AWT. The structure of a Swing user interface is therefore the same as if it was built with AWT components. Swing components inherit all the attributes of their AWT counterparts and some add more. For example, Swing menu components do support background, foreground colors, and fonts. Swing containers lay out components by using the AWT layout managers, and updating is done the same as in AWT.

However, the Swing framework is different from AWT. First, the Swing components are implemented with absolutely no native code so, for every Swing component, there is no corresponding *peer* for every platform that Java runs on. Second, because Swing contains no native code it is not restricted to the intersection of the features present on all platforms. Swing can provide a lot more functionality that works in a consistent way across platforms. Third, Swing allows the user to specify the look and feel of the user interface at runtime. Fourth, Swing components use models to hold the state so that the user can substitute the default model with some other model at runtime.

2.3 Summary

In this chapter, we introduced the reader to user interfaces, a mechanism designed to simplify the interaction between humans and computers. User interfaces are built on top of applications in order to display the application state and modify this state in response to user actions. User interface functionality is implemented in layers, the lowest level being the window system which enables graphical elements to be displayed on the screen, followed by the toolkit which implements a set of commonly used widgets, and finally by the window manager which defines how windows are placed on the screen and how their borders look and act. Additionally, we presented different ways of building user interfaces, by programming, by using interface builders, higher-level languages, or by example.

The most important contribution of this chapter towards the goals of the thesis is that we identified the user interface characteristics that need to be mapped between platforms. These features include the structure, properties, layout, and updating mechanism. The structure of a user interface describes the parent-child relationship between widgets, the properties specify the internal state of widgets, the layout describes how the widgets are positioned on the screen in relation with each other, and the updating mechanism describes how the widgets interact with the application. Finally, we analyzed different cross-platform C++, Smalltalk, and Java frameworks to see how these features get mapped across platforms.

Chapter 3

User Interface Patterns

Application user interfaces should be easy to build, modify and maintain. An obvious way to meet this goal is to have all developers use the same approach. Patterns and pattern languages are excellent vehicles for expressing common design and architectural approaches. The *Observer* [14] pattern, the *Model-View-Controller* [4] pattern and the pattern language for developing form style windows [3] provide a partial set of patterns for building GUIs. After building GUIs for years, we found that more details were required for properly handling the event notification from the visual components and for updating them. We have incorporated these decisions into new patterns without changing the ones mentioned above, forming an ever-growing pattern language for GUI development.

In this chapter we present a system of user interface patterns in Sections 3.1 through 3.5 and one support pattern in Section 3.6. These patterns are used in Chapter 4 by the translators to link the user interface to the application model.

3.1 Introduction

A graphical user interface (GUI) is composed of three entities: a visual container object such as a window or form, other visual components such as buttons and selection lists, and objects modeling the domain. The function of the container object is to act as the master of its visual components, coordinating their assembly and the activity between the user, the visual components and the models. Some visual components, such as a button, enable

an application's user to initiate actions on the models in addition to displaying information about them. Except for a window, containers usually coordinate the display and actions of a single model object rather than those of an entire application. In this paper, we collectively call both windows and forms views.

An important goal to keep in mind when constructing a view is that it is easy to build, modify and maintain. If a developer has a unique implementation style, only he understands his views. It may be easy for him to build, modify and maintain his view. However, we require an approach enabling anyone, not just the original developer, to work on a view without spending weeks understanding the previous implementation.

An obvious way to meet this goal is to use an architecture that standardizes the design. Patterns and pattern languages are excellent vehicles for expressing common approaches to designs and architectures. The *Model-View-Controller* [4] pattern, the *Observer* [14] pattern and the pattern language for developing form style windows [3] provide a partial set of patterns for building views.

Bradac and Fletcher's [3] pattern language addresses how to use forms to manage window construction and how to reuse them while minimizing the number of discrete views required. The language consists of five patterns: *Subform*, *Alternative Subforms*, *Subform Selection*, *Subform Match*, and *Subform Mismatch*. These patterns present a systematic approach to the construction and decomposition of an application's views by factoring visual components into a number of reusable visual components called subforms - which we also refer to as views - and then using them to construct multiple application windows.

The *Model-View-Controller* [4], or *MVC*, pattern divides an application into three parts: a model, a view (window, form, or some other visual container) and a controller. The model contains the information and functionality required by the application. The view displays the information in the model to the user, and the controller handles the user input. This architectural pattern describes the relationships between the model, view and controller and provides a change-propagation mechanism to maintain the consistency between the models and the views.

The *Observer* pattern, used by the *MVC* pattern, manages the notification and updating between dependent objects called observers and subjects. The pattern describes how to notify observer objects when a subject changes state, without forcing the subject to know

the specific classes of the observers.

We find that there are design decisions not dealt with in the previously mentioned patterns and pattern language, which are required to make views maintainable. We incorporated these decisions into new patterns without changing the above-mentioned patterns, forming an ever-growing pattern language for GUI development.

Our patterns are for applications in the following global contexts:

- *You are developing a new window subclass.* A window is a top-level visual container class that contains a title, a border, a menu bar, and other visual components, e.g., buttons, selection lists and forms (otherwise know as panels in Java and subcanvases in VisualWorks Smalltalk). Your window is responsible for the layout of other visual components. It is also responsible for mapping messages generated by the visual components, normally initiated from user actions in the user interface, to the appropriate actions in the models. This task ensures the window and its visual components accurately reflect the state of the models, and that it can handle its own events such as window closing.
- *You are developing a new form or subform subclass.* A form or a subform is just a simple container class. It provides space in which to layout other visual component, including other forms but not a window. A form is similar to a window in behavior, but it does not have the responsibilities associated with maintaining or manipulating a title or menu bar. Your new form is to be used eventually as a component of other windows or forms, or both. Therefore, your windows can contain forms, and your forms can contain forms.
- *The visual components contained in your new form or window have a low degree of dependencies between one another.* The visual components depend on the models and must reflect their state. However, when you update a visual component, it does not force the update of another and another, and so on. Rather, a change in a visual component, requires a change in the models, which in turn requires a single update in the visual components, but that's where the process ends. For example, you are not building a spreadsheet view.

- *The visual components use the Observer pattern to notify the views of changes.* Visual components could invoke specific methods in the views but this would increase the coupling between them. Since many windowing systems already support this form of notification, such as Java, we find that more details are required to handle the event notification from the visual components and for updating the views.

It is because of the similarities between a window and a form - they both contain other forms and visual components - that we treat them the same, referring to both as views.

Within the global contexts, our patterns attempts to resolve the following global forces:

- *A view's functionality changes often.* As an application evolves from early prototypes to a shipped product, practical experience suggests that its overall views will change much more than the underlying business model. Such changes can occur as a result of changing business requirements, ports to different platforms, the desire to construct a more user-friendly version of an interface or from clients requesting a custom user interface. Examples of changes include rearranging a view's layout, shifting its components to another view and changing its behavior. All types of changes must be easy to make but not require a rethinking of the overall user interface design.
- *A significant period may elapse between the time a developer last worked on a view and the time he is making changes to it.* As a developer works on different parts of an application, weeks or even months can go by before he revisits a view that he developed. It should not take a long time for the developer to remember and understand his view's implementation.
- *The original developer of a view is not always the same person that maintains or extends it.* Developers often change groups or companies during the time required to develop a single application. Therefore, the original developer may be unavailable or unwilling to discuss their previous design with a developer assigned to support the old view. As with the original developer, it should be easy for the new one to understand the previous design and implementation and for the same reasons. However, the easier a user interface's design is to understand the easier it is to change it. New developers often feel the need to make changes to suit their styles. These changes come at a time when the user interface should be stabilizing, not changing.

- *Tightly coupled view components are difficult to merge or split.* It should be possible to merge or split views with minimal effort. Ideally, when making changes, it is desirable to refactor the interface rather than throw it away and implement it from scratch. The cost of refactoring is decreased significantly if parts or entire views can be merged or split, i.e., one should be able to add and remove visual components with minimum disturbance to the existing ones or user interface. However, the initial refactoring of views increases development times and costs.
- *Visual containers that have many constraints between their visual components, such as tree views and grids, are difficult and time consuming to update.* Updating one component at a time is simple to implement and understand. However, due to the constraints, cyclical dependencies, and changes to other components, a component may be updated several times before a full update completes. This approach is inefficient and produces a flickering effect on the window.
- *A large percentage of time is required for user interface development.* One often hears of view construction requiring 80% of the development effort. Therefore, it is important to try to minimize the cost required to develop and maintain views, keeping your effort level low at this part of development. However, your views are what end-users see and interact with, and you must take the time to make them visual appealing, easy to use, and ensure the user makes a strong connection between them and your application.
- *Unique view designs are more difficult to support than consistent ones.* Views implemented in a consistent way are much easier to understand and maintain because everyone involved with the task works from the same design principle. Consequently, no developer is required to 'figure out' what the other has done. However, this can make all the views too similar, constraining novelty and innovation.
- *Since a view is a user's entry point into an application, it must be user-friendly.* An application's view that is either complicated to use or slow to react will frustrate a user. Users expect intuitive views that operate in real-time. However, designing user-friendly views takes time.

3.1.1 Pattern Summary

Table 3.1 summarizes the problems addressed by the patterns in our pattern language and their solutions. In the following sections, we describe the new patterns we have added to the already existing patterns. None of them stands on their own. Rather they form dependencies with others in the language, as shown in Figure 3.1, with single-headed arrows pointing to the preceding patterns.

The *MVC* pattern represents the core of the pattern language. It provides the overall architecture for connecting models, views and controllers. The *Subform* patterns help describe how to structure the views. Our new patterns help describe how to structure views and handle user initiated event messages in a more linear manner than the views in the *MVC* pattern and in the *Subform* patterns. Section 3.2's *View Event Handler* pattern describes how a view should handle an event message such that its update process is independent from the visual component generating the event and from the models that have changed state. Section 3.3's *Complete Update* pattern describes how to structure a view's update code such that you or any future developer can quickly and easily understand the logic of the update. Section 3.4's *Multiple Updates* pattern describes how to manage the update of visual components arranged in multiple subforms. In section 3.5, we summarize.

Figure 3.1: Pattern language dependencies.

Pattern Name	Problem	Solution
<i>Model-View-Controller</i> [4]	How do you architect an interactive application?	Divide interactive application into three parts: model, view and controller.
<i>Subform</i> [3]	How do you minimize the development and maintenance efforts when designing form style windows?	Divide a form into subforms.
<i>Alternative Subform</i> [3]	How do you design a window where different sets of widgets are needed based on application state data?	Create a subform for each variation of widget sets.
<i>Subform Selection</i> [3]	How do you choose from a collection of subforms that become active/inactive based on application state data?	Have the parent form maintain and coordinate the subforms.
<i>Subform Match</i> [3]	How does a parent subform select the appropriate subform based on application state data?	Have the parent form maintain and poll the subforms for a match.
<i>Subform Mismatch</i> [3]	How does a parent subform determine when a child subform no longer applies based on child state data?	Have the subform notify parent when it no longer matches.
<i>View Event Handler</i> < new >	How should a view, such as a form, handle an event notification message from an observed visual object, such as a button?	Write a handler method for the event that only pulls information from the visual components and then calls for an update.

Pattern Name	Problem	Solution
<i>Complete Update</i> < new >	How do you structure a view's update mechanism?	Assume the entire view is out-of-date with the models and update everything.
<i>Multiple Updates</i> < new >	How does one view force the update of visual components in other views?	The view invokes the update behavior of its parent and child views.

Table 3.1: Pattern Problems and Solutions.

3.2 View Event Handler Pattern

3.2.1 Context

You have incorporated the *Observer* [14] pattern into your view's design. This may have been a deliberate decision or the result of using another pattern, such as *MVC*, that uses the *Observer* pattern as part of its structure. Your view is interested in receiving event messages from its observable visual objects when specific events occur in them. Your view needs to collaborate with one or more model objects in order to handle these notification messages and then update its visual components in response. You want your view to update its visual components in an organized, linear manner, so that components in corresponding regions of the view update together rather than an apparent random manner.

3.2.2 Forces

- Many observable visual components can generate identical event messages. If you handle each message with a different method, it is easy to add and remove visual objects; just add or remove a handler method. However, this approach creates dependencies between your handler methods and the specific visual components.
- You can use a case statement to determine which visual object sent the event message and how to process it in a single handler method. However, this is inefficient and can

make it difficult to decide how to add, remove or change the way you handle the events. Moreover, if you change the source of the event, you must always recompile the method. Replaceable, or plug-and-play, components will also not be an option.

- Setting flags when handling event messages in order to decide which observable visual objects to update ensures that you don't perform any redundant updates. However, the relationships and use of flags is complicated for you and for future developers to understand.
- If the handling of a single event message results in many changes to your view's model, the *MVC* pattern forces the model to send notification messages to its corresponding visual objects several times. This can be inefficient when the visual components need only updating once after the model has completed all changes.

3.2.3 Problem

How should a view handle event notifications from its observable visual components such that it does not create any updating dependencies between the methods and the number of updates to the visual components is minimized?

3.2.4 Solution

In your view, create and register a handler method for each event message from your visual components. Do not register your views with the models. Your view's handler method should pull information from the appropriate visual components when invoked and then invoke the corresponding services in the models. Since the models never update the visual components, make sure your view updates the corresponding visual objects with the latest states of the models after handling an event. Your handler method should not do any updating of the visual components itself. Instead, have your handler use a single update method that is responsible for updating all visual components.

3.2.5 Consequences

The advantages of the *View Event Handler* pattern are the following:

- *Predictable update.* Since only a single update is required, the view can be refreshed in a predictable, coordinate manner and redundant updates are avoided.
- *Unifies the event handling scheme.* After receiving an event message, your view takes the required information from the appropriate visual components and invokes the corresponding services in the models. These actions change the states of the models, and, consequently, the view is now in need of an update to reflect the models' latest states. All visual components, all events and all updating is handled the same way in all views.
- *Simple to add and remove or modify the handling of visual components.* To add the capability to handle a new event message from a visual component, add a new handler method. To remove the processing of an event, delete the corresponding handler method. To modify the way an event message is processed, change the corresponding handler method.
- *Easy to locate how visual components are handled.* There is always one and only one method to look at for the addition, deletion or modification to the way an event message is handled, which makes it easy for you or future developers to understand. Moreover, no special flags or case statements are required to process events.
- *Provides best maintenance results if used consistently.* To gain the maximum maintenance benefits from this pattern, all developers will need to use it consistently.

The disadvantages of the *View Event Handler* pattern are the following:

- *May make updating complex.* For very large views, a single update method may become too complex. To some extent, this can be solved having the main update method coordinate several smaller updates.

3.2.6 Example

Consider the following Java implementation of a `ToDoList` window, as depicted in Figure 3.2. The business model consists of a collection of items, where each item is a `String`. The user can add new items to the list by typing the new item in the text field and clicking the `Add` button. The user can remove items from the list by selecting an item and clicking the `Remove` button.

Figure 3.2: `ToDoList` example.

The view's models are represented by instance variables associated to the `items` collection, the `selectedItem`, and the `newItem` to be added. In addition, the view keeps track of each visual component along with their corresponding listeners:

```
public class ToDoListFrame extends JFrame {
    private JTextField newItemView;
    private JList itemsView;
    private JButton addButton;
    private JButton removeButton;

    private String newItem;
    private Vector items;
    private String selectedItem;
```

```
private DocumentListener newItemDocumentListener;  
private ListSelectionListener itemsListSelectionListener;  
private ActionListener addButtonActionListener;  
private ActionListener removeButtonActionListener;  
...  
}
```

The user can type in the text field, select an item in the list or click the Add and Remove buttons. Handler methods need to be written for each of these events. In Java, you need to add listeners to the appropriate visual components (code not shown in the interest of brevity) in order for the view to receive event notification messages from them. In this example, the `DocumentListener` for the text field sends the `ToDoListFrame` the message `changedNewItemView` every time the user types in the text field. The `changedNewItemView` handler method reads the contents of the text field into the `newItem` instance variable:

```
private void changedNewItemView() {  
    if(newItemView.getText().length() == 0) {  
        newItem = null;  
    } else {  
        newItem = newItemView.getText();  
    }  
    update();  
}
```

The `ListSelectionListener` for the list sends the `ToDoListFrame` the message `changedItemsView` when the user selects an item in the list. The handler method reads the selected item from the list into the instance variables `selectedItem` and `newItem`:

```
private void changedItemsView() {  
    selectedItem = newItem = (String)itemsView.getSelectedValue();  
    update();  
}
```

The Add and Remove buttons use an `ActionListener` to send the `ToDoListFrame` the messages `addToDoItem` and `removeToDoItem` when the user clicks on

them, respectively. The `add_TODOItem` handler method adds the new item to the collection of existing items, whereas the `remove_TODOItem` handler method removes the currently selected item from the same collection:

```
private void add_TODOItem() {
    items.add(newItem);
    selectedItem = newItem;
    update();
}
private void remove_TODOItem() {
    items.remove(selectedItem);
    selectedItem = null;
    newItem = null;
    update();
}
```

All handler methods invoke a single update method just before they complete. The view models' states are modified as a result of handling user events, after which the states are loaded back into the visual components, during updating, to force the view to reflect the changes to the states.

3.3 Complete Update Pattern

3.3.1 Context

You used the *View Event Handler* pattern to develop your view (typically a form) that contains a small number of visual components that can be updated simultaneously in a reasonable amount of time.

3.3.2 Forces

- Changes of the view's model objects may be the result of the view's action or of another object's action the view does not know. However, you want to handle both cases at once and avoid redundancy

- Developer created flags indicating which visual components require updating in a view are complicated to understand for both the immediate and future developers. Flags used in a systematic and documented way by all views makes them easier to understand and to use by developers.
- You are not always the one who will maintain your views so the view's logic must be easily understood.
- You can often improve performance by using tricky implementation techniques. However, these tricks often complicate your code by making it less easy to read and comprehend.
- You do not always know what has changed in your view's models or which visual components need updating.

3.3.3 Problem

How do you structure a view's update mechanism?

3.3.4 Solution

Assume the entire view is out-of-date and update everything. In most cases, there will be superfluous updates. However, if the view is not too complex, most computers are fast enough to do the updates without recognizable flicker. Write a specialized update method for each region of your view. Each specialized update method is responsible for updating its corresponding visual components from information in the models. Permit each specialized update method to load model information directly into the visual components or compute the contents on the fly. Write a single, main update method that invokes all of the specialized update methods.

3.3.5 Consequences

The advantages of the *Complete Update* pattern are the following:

- *Precise.* The view's main coordinating update method ensures that the view reflects the current state of the models.
- *Easy to determine how visual components are updated.* There is only one corresponding update method to add, remove or change.
- *No special flags required.* The speed of processors today allows this luxury and eliminates the need for developers to create their own special purpose flags to remember what to update.

The disadvantages of the *Complete Update* pattern are the following:

- *Infinite Loops.* This approach can cause infinite loops in some windowing systems (see 3.8.1 for solution).
- *Performance.* Complex views may take too long to update.

3.3.6 Example

Continuing with the previous example, update methods must be written for all visual components in the `ToDoList`. The update method for the text field sets the contents of text field to the string in `newItem` variable. The update method for the selection list sets the contents of the selection list to the collection of `items` and the selection to the `selectedItem` string:

```
private void updateNewItemView() {
    newItemView.setText(newItem);
}
private void updateItemsView() {
    itemsView.setListData(items);
    itemsView.setSelectedValue(selectedItem, true);
}
```

The update methods for the Add button and the Remove button only take care of enabling or disabling the buttons depending on whether their events are permitted. The Add button cannot be clicked if there is nothing typed in the text field (in other words `newItem`

is null). The Remove button cannot be clicked if there is no selection made in the list (in other words `selectedItem` is null):

```
private void updateAddButton() {
    addButton.setEnabled(newItem != null);
}
private void updateRemoveButton() {
    removeButton.setEnabled(selectedItem != null);
}
```

The coordinating update method, which all handler methods use, invokes the specialized ones ensuring all visual components reflect the current state of the models:

```
private void update() {
    updateNewItemView();
    updateItemsView();
    updateAddButton();
    updateRemoveButton();
}
```

3.3.7 Variants

Infinite Event-Handle-Update

This approach can cause infinite loops in some windowing systems. This situation happens when messages sent to one of the visual components in the update method trigger a new event message from the visual components, causing the subsequent execution of a handler method, which results in another invocation of the update method, and so on. To prevent this situation from occurring, disable the visual components from generating new event messages while your view is currently handling an event message. In our example, this involves removing the listeners from the visual components. The `disableListeners` method removes the Java listeners from the corresponding components, and `enableListeners` adds them back. The result is that we update all visual components while the listeners are disabled:

```
private void update() {
    disableListeners();
    updateNewItemView();
    updateItemsView();
    updateAddButton();
    updateRemoveButton();
    enableListeners();
}

private void disableListeners() {
    newItemView.getDocument().removeDocumentListener(newItemDocumentListener);
    itemsView.removeListSelectionListener(itemsListSelectionListener);
    addButton.removeActionListener(addButtonActionListener);
    removeButton.removeActionListener(removeButtonActionListener);
}

private void enableListeners() {
    newItemView.getDocument().addDocumentListener(newItemDocumentListener);
    itemsView.addListSelectionListener(itemsListSelectionListener);
    addButton.addActionListener(addButtonActionListener);
    removeButton.addActionListener(removeButtonActionListener);
}
```

Do not create new listeners (observers) or visual components (subjects) when disabling and enabling event messages; this act would be inefficient. Create the listeners and visual components once in the view's constructor and use private instance variables to maintain the references to them. Use these references to enable and disable the corresponding objects. Taking this approach, the methods for disabling and enabling are simple to understand and modify. However, this does open another place to modify whenever a component is added or deleted.

Stopping Recursive Updates Without Disabling/Enabling Listeners

As mentioned before, the updating of visual components by a view may generate additional event messages. Since the handling of these event messages always invokes an update which may cause more event messages, and so on, the result can be an endless loop of update message, event message, and handler responses.

Rather than disabling and enabling the objects responsible for generating the new event messages, which may be computationally expensive, another solution is to discard any new event messages while already processing one. To include this ability, add a new private instance variable to your view called `updating` and initialize it to `false` when creating your view. In our example, the class definition for the `TodoListFrame` is modified as follows:

```
public class TodoListFrame extends JFrame {
    private boolean updating = false;

    private JTextField newItemView;
    private JList itemsView;
    private JButton addButton;
    private JButton removeButton;

    private String newItem;
    private Vector items;
    private String selectedItem;

    private DocumentListener newItemDocumentListener;
    private ListSelectionListener itemsListSelectionListener;
    private ActionListener addButtonActionListener;
    private ActionListener removeButtonActionListener;
    ...
}
```

Also provide the following methods:

```
private boolean isUpdating() {
    return updating;
}
private void beginUpdating() {
    reacting = true;
}
private void endUpdating() {
    reacting = false;
}
```

Next, change all handler methods to the following format.

```
private void changedItemsView() {
    if(isUpdating()) return;
    selectedItem = newItem = (String)itemsView.getSelectedValue();
    update();
}
```

Finally, remove the `enableListener` and `disableListener` methods for the main update method and replace them by `beginUpdating` and `endUpdating`:

```
private void update() {
    beginUpdating();
    updateNewItemView();
    updateItemsView();
    updateAddButton();
    updateRemoveButton();
    endUpdating();
}
```

Now, no handler methods can execute until the current one has finished, including its update. In cases where different threads send and handle different event messages, the handler and update methods must be synchronized.

Partial Updates

As the view stabilizes, and refactoring becomes less and less frequent, one can improve the approach's performance by implementing partial updates. In this case, modify the update methods to update the visual components only when the information in them does not reflect what is in the models, rather than always. This variant may also be required to stop any noticeable flickering of the views on slower machines. For example, the update method for the text field can be written as follows:

```
private void updateNewItemView() {
    if(!newItemView.getText().equals(newItem)) {
        newItemView.setText(newItem);
    }
}
```

3.4 Multiple Updates Pattern

3.4.1 Context

You are using the *View Event Handler* pattern, the *Complete Update* pattern and one or more of the subform patterns in the pattern language for developing form style windows. In each one of your views - a subform also being a type of view, you have implemented a number of handler and update methods to keep the visual components in the view up-to-date with the corresponding models. Either your views share models, or events generated and processed in one view require other views to be updated. The immediate view is related to the others requiring updating in one of three ways:

- They may be one of its parent views.
- They may be one of its child views.
- They may be one of its sibling views.

The root type of a view is usually a window, or in Java it can also be an applet. This root view can contain additional views, and they in turn contain others. Although an update may start in any view, any or all others may need to go through their own update.

3.4.2 Forces

- It is not scaleable for every view to be informed when every other view changes in order to perform updates. One could use the *Observer* pattern with views playing both the parts of observers and subjects, allowing views to be notified when others changes. However, this would create dependencies between all views when the dependencies should be between the views and the models. In large views, the complexity of the implementation and the potential for cyclical dependencies makes this approach difficult to understand and manage.
- Views are often used as plug and play components. Therefore, they must be easy to install or remove from another view. For example, if you have an address view that

works with a Canadian address, you may need to change this to a US address view when working with a US address object. The switching between views should not demand a large coding effort.

- Views often change at runtime. In some applications, often involving notebooks or tab controls, views are swapped dynamically. This requires that views and their update mechanisms must work for all update situations, not just situations involving specific views. Therefore, updating between views can not be hard-coded, view specific.
- You are not always the one who will maintain your code so the update logic must be easily understood by anyone.
- No magic flags should be used to indicate what needs to be updated because they are complicated to understand for both the immediate and future developers.

3.4.3 Problem

Since a view's update methods are only responsible for updating their owned, or contained, visual components, how does a view update the visual components in other related views if required?

3.4.4 Solution

The first part of the solution is to insure that an updating view has a reference to its parent view and references to all of its child views. The updating view can establish these references when initialized. The second part of the solutions has the updating view send both its parent and child views the update message whenever it is requested to update. Since both the parent and child views will do the same, all views must ignore other update messages while currently processing one. This will prevent any infinite updating loops from occurring. Since it is user events generated by visual components, such as a button, that begin the updating processing, and not the models, there is a single point of control. This insures that no simultaneous updates can occur within one view.

3.4.5 Consequences

The advantages of the *Multiple Update* pattern are the following:

- *Simplicity.* From the single initiation point, update requests traverse the view's component hierarchy. Eventually, all views execute their updates and the views will reflect an accurate display of the underlying models. No magic flags are required by the approach and views can be changed, added or removed without having to make major modifications to existing views. The approach is so simple that any developer, not just the original one, can work on a view that uses it without knowing much about what coding was done in the past or in other views.
- *Uniform synchronizing of views.* Unlike the *MVC* that initiates synchronization of the visual components from the models resulting in an unordered updating of them, this pattern synchronizes updating of the visual components after all models have completed their change of state, leading to a more uniform and ordered updating of the visual components.
- *Minimizes dependencies.* Unlike the *MVC*, that would force the set up and maintenance of dependencies between all models, views and controllers, in the *Multiple Updates* pattern the dependencies are only between the parent and child views, which are usually maintained by the windowing system already.

The disadvantages of the *Multiple Update* pattern are the following:

- *Infinite Loops.* This approach will cause infinite loops if not managed correctly.
- *Performance.* Complex views may take too long to update.

3.4.6 Example

Consider the following Java implementation of a multiple `ToDoList` window, as depicted in Figure 3.3. For this example, the corresponding buttons, list and text field are assembled in a `ToDoListPanel` (Java's version of a subform). Therefore, the example consists of a single window with two child panels. The window and panel views share common models:

a collection of items (each item a String), a `selectedItem`, and the `newItem` to be added. The functionality of this window is the same as in the original `ToDoList`. However, what happens in one of the panels will now also change what is displayed in the other. For example, if an item is removed from one panel's list, it will also be removed from the other.

We leave out the full implementation of the window and panels in the interest of brevity and focus on how updating occurs. In short, the handler and update methods for each panel are almost identical to the examples given before. However, in this situation, we must handle the fact that multiple panels may require updating. Therefore, an update method invokes the update methods of its parent window or view and any of its child views, which in this case is only the `MultipleToDoListWindow`. We also wrap the update method in a guard like we did for handling infinite update loops to ensure that any new requests to update will not be done until the current one completes. The following update method is for the `ToDoListPanel`:

```
public void update() {
    if(isUpdating()) return;
    beginUpdating();
    updateNewItemView();
    updateItemsView();
    updateAddButton();
    updateRemoveButton();
    parent.update();
    endUpdating();
}
```

The `MultipleToDoListWindow`'s update method follows the same convention, but needs only to invoke the update of its subforms since the window has no parent:

```
public void update() {
    beginUpdating();
    panel1.update();
    panel2.update();
    endUpdating();
}
```

Figure 3.3: Multiple ToDoList example.

3.5 User Interface Patterns Discussion

Our new patterns work in two phases. The handle phase changes the states of a view's models in response to end-user events, and the update phase updates the visual components to reflect the models' new states. Since the update phase immediately follows the handle phase, the view always reflects the latest changes.

Our patterns allow the views to be modified easily. In order to add a new component, it is necessary to write an update method for it, and as many handler methods as required. In a similar fashion, to remove a component, it is necessary to remove the corresponding update method and the all of the handler methods. Therefore, every component is defined by one update and many handler methods, which is consistent for all the components. Since

the update and handler methods access only specific visual components, these components can be added or removed without having to modify the update and handler methods for the other components.

Our patterns form expectations in developers' minds. Any view contains handler and update methods and an easily identifiable interaction between the views and the models. If there is a problem in the handling or updating in response to a user event, the developer knows where to look.

If you decide to use these patterns consistently, you can summarize them with the following step-by-step instruction:

- Determine each view's state variables and associated objects.
- Write a handler method for each event a user can generate from interacting with the view's visual components. Allow a handler method to modify only the view's model (state) objects and retrieve information from the immediate visual components but prohibit it from directly modify any other visual components. Force the last statement in each handler method to invoke the view's main update method.
- Write a specialized update method for each visual component. Each method is responsible for updating the corresponding visual component from the view's state information. The method is permitted to directly load state information into the visual components or can compute the components' contents on the fly.
- Write a single, main update method that invokes all the specialized update methods.

3.5.1 Known Uses

- Carleton University has taught similar approaches to user interface development for many years in its first year Design and Implementation of Computer Applications course.
- Employees of The Object People, Inc. have successfully used variants of these patterns during large consulting projects.

- The composition editors of VisualAge for Java and VisualAge for Smalltalk use a portion of the *View Event Handler* pattern, creating individual handler methods for each of the visual connections. However, updating is not managed by one central update method.
- Many systems use the approach of having the view inform its parent of any changes [19]. However, they also often localize all code to process the notification in the parent, which is usually not reusable.

The patterns have been used with Digitalk's Smalltalk/V (now SmalltalkExpress) and VisualSmalltalk, ObjectShare's VisualWorks Smalltalk, IBM's VisualAge Smalltalk, and recently, with Sun's Java AWT and Swing frameworks.

3.5.2 Related Patterns

The *Model-View-Controller* [4] and the *Observer* [14] patterns rely on an update mechanism similar to the patterns presented in this thesis. However, the *MVC* pattern does not specify when information can be retrieved or placed in the visual components. Without this specification, one often requires either several user defined flags or spaghetti code to get the updating sequence correct. Our patterns make the assumption that the user interface is completely out-of-date after handling a request. This may seem a bit extreme. However, with today's computer speeds, this simplification does not cause any performance degradation and results in a much easier and extensible design than one that must figure out what and when to update.

The updating mechanism in the *Multiple Updates* pattern is a bi-directional application of the *Chain of Responsibility* [4] pattern.

3.6 The Lambda Pattern

The need often arises to write simple, possibly one-shot behaviors that do not seem to belong in the interface of any class. The behavior can be assigned a name and encapsulated in a method in a specific class, but the given name would convey little information beyond

what can be inferred by directly looking at the code. In such cases it is better to express the behavior as λ – *Functions*, unnamed behaviors expressed as blocks in Smalltalk and inner classes in Java.

3.6.1 Context

Although the need for simple, one-shot behaviors can arise in other types of programming (such as functional and imperative procedural programming) this pattern applies to object-oriented programming and is particularly useful for use with languages like Smalltalk and Java. Use the lambda pattern under one or more of the following circumstances:

- *You need to express some small, self-evident behavior.* You could try to assign the behavior an *Intention Revealing Selector* [2] name, and encapsulate it in a method in a specific class, but the name would convey little additional information beyond what can be inferred by directly inspecting the code.
- *You need to save some context for later execution.* Behaviors operate on some local context. This context along with the behavior that operate on it have to be saved and evaluated a later time. This can happen under the following conditions:
 - *You need to encapsulate how certain objects interact.* The way objects interact changes frequently so the objects have to be expressed such that, in order to minimize coupling, they do not reference each other explicitly.
 - *You need to configure an object for later access.* The configuration is a one-shot deal and there is little chance that the code will be reused. The configuration code does not really belong in the interface of the caller or callee, but may require full access to the state and behavior of the former.

Additionally, the following may apply:

- *You need to adapt an interface.* The client object cannot collaborate with the object that provides the required behavior due to an interface mismatch. The caller object cannot be modified to the interface of the callee because it has been designed as a

reusable component that expects a well-defined interface from the objects it interacts with. In other words, you are in a position to make use of the *Adapter* [14] pattern and the adapter behavior is not worth its own class or method.

- *You need to implement a series of algorithms.* The algorithms can be written in a couple of lines and have identical interfaces so they can be used interchangeably. The algorithms do not require access beyond their local context except for the input parameters. In other words, you are in a position to make use of the *Strategy* [14] pattern and the strategy behavior is not worth its own class or method.

3.6.2 Forces

- *Objects should be as simple as possible, but not simpler.* An object is defined as a collection of related behaviors working on the same data. This promotes modularity and information hiding, qualities that tend to dissipate when the object has either too much functionality, too much state, or too much of both.
- *Classes should have an optimum number of methods.* Behaviors in the public interface are usually implemented as *Composed Methods* [2]. They are coded in terms of operations at the same level of abstraction, which can be found in the protected or private interface of the same class. Splitting complex methods this way improves the readability of the code but, at the same time, increases its complexity.

A possible solution to express a simple, one-shot behavior, would be to encapsulate the behavior in a private method in some class. If many such behaviors have to be written, the number of private helper methods will outnumber the rest of the methods, and therefore increase the complexity of the class.

- *Systems should have an optimum number of classes.* Classes in a system represent either concepts in the problem domain, or are used as helpers to glue the problem domain classes to platform-specific frameworks. The number of helper classes has to be minimized in order to increase the communicability of the system.

Another possible solution to express a simple, one-shot behavior, would be to create a class to store the behavior. This, however, introduces new helper classes, and

therefore increases the complexity of the system.

- *Classes, methods, and variables should have meaningful names.* Good names provide insight into the purpose and design of a system, they reveal its inner workings, and communicate themes and variations of the present abstractions. Class names should convey the purpose of the class in the system, while its subclasses names should convey the difference from the base class (see *Simple Superclass Name* and *Qualified Subclass Name* [2]). Methods should be given *Intention Revealing Selectors* [2], according to which the name describes what the method is trying to accomplish. Similarly, variables should be given *Role Suggesting Instance Variable Names* or *Role Suggesting Temporary Variable Names* [2] such that the name reflects the role the variable plays in the computation.
- *Abstractions are sometimes self-evident in certain contexts.* For many abstractions, any one name can only hint at how the abstraction works or how it might be useful. Full understanding comes when the abstraction is used in a specific context. In some cases, there is little benefit to name the abstraction: its meaning is self-evident from the context in which is used.
- *Behaviors need access to local and context state.* Behaviors are defined as methods in some class. They have access to local state stored in temporary variables, and to context state stored in the instance/class variables of the class and its superclasses. Small, self-evident behaviors also need this kind of access if they are to replace full-fledged methods.

3.6.3 Problem

How do you express simple, possibly one-shot behaviors that do not seem to belong in the interface of any class?

3.6.4 Solution

Write the behaviors as λ – Functions.

The notion of λ – *Functions* comes from λ – *Calculus* [6], a language developed in the 1930s by Alonzo Church to help with the formalization of programming languages and programming in general. λ – *Functions* are used to introduce abstractions into a system, where each abstraction is defined by some state (bound and/or free variables) and some behavior (sequences of expressions that may recursively include other λ – *Functions*). These abstractions can be evaluated in either *applicative order*, where the occurrences of the variables in the function’s body are replaced by the value of the argument expressions, or in *normal order*, where the variables are replaced by the unevaluated argument expressions. In the context of the application, the functions are considered first class objects, in other words functions can be passed as arguments to other functions, used for return values, or assigned to variables.

λ – *Functions* can be named by assigning them to a variable and using the variable as a placeholder for the entire function. The decision whether to name the function or not stems from how the behavior is to be used. One names a function only when that function can be reused in other parts of the application or in future applications; when the behavior is used only once, it is not necessary to assign it a name. Since, by definition, λ – *Functions* are nameless sequences of actions declared and used in the context where are needed, they are particularly suited to express simple one-shot behaviors.

<i>Language</i>	<i>λ – Function equivalent</i>
Smalltalk	blocks
Java	anonymous inner classes
Lisp/Scheme	lambda-functions
C/C++	function pointers

Table 3.2: λ – *Function* equivalents in different languages.

Although primarily used as a model for functional languages and functional programming, variants of λ – *Functions* have been introduced in object-oriented languages, most notably Smalltalk, and recently Java, as shown in Table 3.2. Smalltalk supports such functions via blocks, a way to represent deferred sequences of actions. These actions are compiled by the compiler into executable objects stored in the body of the method where they are defined. Block objects have full access to the context of the method, can be assigned to

variables, or can be passed as arguments to other methods. The code inside blocks is evaluated at a later time, when requested to do so by sending the messages `value`, `value:`, `value:value:`, or `valueWithArguments:`.

Java can simulate λ – *Functions* through anonymous inner classes. As opposed to a top-level class which has to be defined in the context of a package, an inner class can be defined in the context of a class or a method. A further refinement of the inner class, the anonymous inner class, can only be defined in the context of an expression. Inner classes can be declared as normal classes, i.e., they can have a name and be instantiated many times in the context where they have been defined, but anonymous inner classes are only instantiated once, in the expression where they have been defined. Therefore, anonymous inner classes are either declared as right-hand sides in assignment expressions or as arguments in method calls, in much the same fashion as blocks are declared in Smalltalk.

Although both Smalltalk blocks and Java inner classes are first class objects, provide deferred code execution, and can be used as language specific λ – *Functions*, they are not entirely similar. They differ in two aspects: interface signature granularity and return behavior. Smalltalk blocks provide the equivalent of a one method interface whereas a Java inner class allows the definition of more than one method. A hardcoded return in a Smalltalk block causes execution to jump in the context of the method where the block is defined, whereas in Java, an explicit return in the method of an inner class resumes execution in the caller of that method.

Some languages offer a very restrictive implementation of λ – *Functions*. In C and C++, λ – *Functions* can be approximated by function pointers. However, the use of function pointers still forces the user to represent simple one-shot behavior in a named function, removed from the context where it makes sense.

3.6.5 Example

One recurring problem in the design of good container classes is how to provide a way to sort the elements of the container in some specified order without revealing the underlying representation of the container, and without having to create subclasses specialized by the sorting algorithm. One common approach is to use a single container object that can

delegate sorting behavior to a sort-policy object which is provided by the client. The main difference is how the sort-policy objects are implemented. While some libraries choose to implement them as distinct classes, some implement them using the λ – *Function* equivalent of the chosen implementation language.

Lisp and Scheme, both functional languages using λ – *Calculus* as their underlying model, use a λ – *Function* to directly specify the sorting criteria for a list, as shown in the following example, which returns (1 2 3 4) as the sorted list:

```
(sort (lambda (a b) (< a b)) '(4 2 1 3))
```

In Smalltalk, the same result can be accomplished by using blocks:

```
#(4 2 1 3) asSortedCollection: [:a :b | a < b]
```

In Java, the same result can be achieved by using an anonymous inner class that implements the `Comparator` interface. The `Comparator` interface provides protocol for comparing two objects and returns a negative integer, zero, or a positive integer if the first object is less than, equal to, or greater than the second. The anonymous inner class can then be passed to a sort method along with the collection that needs sorting:

```
Vector collection = new Vector();
collection.add(new Integer(4));
collection.add(new Integer(2));
collection.add(new Integer(1));
collection.add(new Integer(3));

Comparator comparator = new Comparator() {
    public int compare(Object source, Object target) {
        int sourceInt = ((Integer)source).intValue();
        int targetInt = ((Integer)target).intValue();
        return(sourceInt<targetInt ? -1
            : (sourceInt==targetInt ? 0 : 1));
    }
}

Collections.sort(collection, comparator);
```

As you can see from the above example, a sorting criterion can be specified inline, as an argument to the sort method. The service provider is the container object which can be configured with a sort criterion by its clients. This configuration is done without extra classes or methods, in a way that keeps the design simple and understandable.

3.6.6 Resulting Context

- By expressing behavior with blocks in Smalltalk and anonymous inner classes in Java instead of creating helper methods and classes, the system as a whole becomes more understandable and maintainable. This is mainly because the system complexity is reduced by minimizing the number of classes and methods. However, lambdas should not be seen as a substitute for helper methods and classes; lambdas should only be used to express behavior that is self-evident in the context where it is needed.
- The use of the lambdas may not simplify the system, because code written in λ – *Function* format can be difficult to read. Such code use should be limited to those functions that are very small (no more than one or two method invocations) and whose use is well-understood. If the meaning of the code in a λ – *Function* is not self-evident, then it maybe needs to be refactored, an opportunity to create either new classes or methods.

For example, in the Java version of the sorted collection presented above, we can clean up the code by moving the comparison logic into the class of the argument objects of the compare method, in this case Integer:

```
...
// Source in client class.
Comparator comparator = new Comparator() {
    public int compare(Object source, Object target) {
        return ((Integer)source).compareTo((Integer)target);
    }
}
Collections.sort(collection, comparator);
...
// Source in Integer class.
public int compareTo(Integer target) {
```

```
int sourceInt = this.value;
int targetInt = target.value;
return(sourceInt<targetInt ? -1
       : (sourceInt==targetInt ? 0 : 1));
}
...
```

This, however, will not work if the source and target objects are of different classes. In this case, a different approach, such as double dispatching [2, 14], must be used.

- When switching between lambda functions and classes, a Java application needs not be modified extensively because the interface of top-level, inner, or anonymous classes is the same to all clients. However, in Smalltalk, since blocks can only be evaluated using a fixed protocol (value, value:, value:value:, or value-WithArguments:), two possibilities exist:
 - The block protocol can be added to the new class. The methods value, value:, value:value:, or valueWithArguments: are added to the new class to implement the desired behavior directly or to call the appropriate methods. This is better in the short run, while prototyping, because it avoids changing all the methods that used to evaluate the block.
 - The methods that evaluate the blocks can be modified to use the protocol of the new class. This is better in the long run because the implementation of these methods can be more easily grasped, specially if the methods in the new class are given *Intention Revealing Selector* [2] names.
- In Smalltalk, blocks cannot inherit from other blocks whereas, in Java, inner classes can subclass any class in the current scope.
- In Java, all variables and parameters accessed from within an inner class *must* be declared final because of potential synchronization problems.

3.6.7 Rationale

λ – *Functions* resolve the forces presented in Section 3.6.2 as follows:

- *Objects should be as simple as possible, but not simpler.* λ – *Functions* are defined as first class objects, a collection of related behaviors working on some data. The encapsulated state and behavior promote modularity and information hiding, as does any other object in the system.
- *Classes should have an optimum number of methods.* By encapsulating state and behavior in λ – *Functions*, the number of helper methods otherwise required to express the same functionality is reduced, and the complexity of the class that would have had to store the methods is kept to a minimum.
- *Systems should have an optimum number of classes.* By encapsulating state and behavior in λ – *Functions*, the number of helper classes otherwise required to express the same functionality is reduced and the complexity of the system that would have had to store the classes is kept to a minimum.
- *Classes, methods, and variables should have meaningful names.* The behavior expressed with λ – *Functions* is meant to be self-evident and this is accomplished by keeping the code short and assigning meaningful names to the classes, methods, and variables referenced within the lambda.
- *Abstractions are sometimes self-evident in certain contexts.* λ – *Functions* are nameless sequences of actions whose intent is better understood by looking at the encapsulated state and behavior directly, in the context where is needed, rather than at some name that describes it.
- *Behaviors need access to local and context state.* λ – *Functions* have full access to the behavior and state of the context where they are defined. For example, in Smalltalk, a block has access to a local context (block temporary variables), the method context in which is defined (method temporary variables, method arguments), the class context (instance variables, class variables, instance methods), and the global context (global variables).

3.6.8 Related Patterns

The following patterns from [14] can be implemented using the lambda pattern:

Adapter. The *Adapter* pattern is used to convert the interface of a class to another interface that client objects expect. In this case, rather than implement an *Adapter* class, the interface adaptation can be done using a lambda function. In Java, interfaces with more than one method can be easily adapted using anonymous inner classes containing multiple methods. In Smalltalk, however, this can only be accomplished by using a block for each of the methods to be adapted (for example, the *PluggableAdapter* example in Section 3.6.9 provides blocks for getting, setting, and updating the model).

Bridge. The *Bridge* pattern is used to decouple an abstraction from its implementation so the two can vary independently. The implementation of the abstraction, as described in [14], is provided in concrete classes but can also be provided as lambda functions.

State. The *State* pattern permits an object to alter its behavior when its internal state changes in such a way that the object appears to change its class. This is accomplished by having the object keep track of a current state object that can be substituted with different other state objects. Rather than have a hierarchy of state classes, a single state object can be used. This state object can be configured with lambda functions that will execute on state transitions (see Section 3.6.9 for an example).

Strategy. The *Strategy* pattern is used to define a series of interchangeable algorithms. These algorithms can be implemented as lambda functions rather than full classes. For example, the criterion for sorting collections from Section 3.6.5 has been represented as a lambda function rather than as a class.

3.6.9 Known Uses

Smalltalk Model-View-Controller

The *Model-View-Controller* framework in VisualWorks Smalltalk uses adapters (variations on the *Adapter* [14] pattern which are presented in [1]) to provide a level of indirection

between the view, controller and the model. Two of the most used adapters are the `ProtocolAdaptor` and `PluggableAdaptor` which convert the messages `value` and `value:` sent by the view/controller pair to the interface of the model. The `AspectAdaptor`, a concrete subclass of `ProtocolAdaptor`, translates `value` and `value:` into protocol understood by the model, whereas the `PluggableAdaptor` converts the messages into arbitrary actions defined by blocks.

Figure 3.4: MVC adapters.

To illustrate this concept, consider the issue of unit conversion. Suppose, due to internationalization constraints, that the height of a person object needs to be converted in the user interface from inches to meters. The person object provides the protocol `height` and `height:` to access and modify the height value in inches only. One possible solution is to implement the methods `heightInMeters` and `heightInMeters:` that perform the conversion and then call the `height` and `height:` methods. Then, the person object can be adapted to the input field via an instance of `AspectAdaptor` as follows:

```

anInputFieldView model: (
    AspectAdaptor
    accessWith: #heightInMeters
    assignWith: #heightInMeters:)

```

However, this solution complicates the interface of the person object by adding the methods necessary to do the conversion. Furthermore these methods do not belong in the application model, they are an artifact of the user interface. A better solution would be

to express `heightInMeters` and `heightInMeters:` as λ - *Functions* using an instance of `PluggableAdaptor`. This way, the conversion itself is done in blocks, with the blocks being defined in the context of the class that creates and links the user interface to the domain model:

```
anInputFieldView model: (  
  PluggableAdaptor new  
    getBlock: [:model | model height * 0.3]  
    putBlock: [:model :value | model height: (value / 0.3)]  
    updateBlock: [:model :aspect :param | "do nothing"])
```

Smalltalk HotDraw Tools

HotDraw is a framework that helps with the construction of drawing editors. Such editors usually provide a tool palette and a drawing area where the user can manipulate graphic figures in different ways based on the currently selected tool. Internally, tools make use of a finite state machine in which transitions are made based on the current mouse event and the figure under the mouse cursor. Transition into a new state causes the action block associated with that state to be evaluated.

Figure 3.5: Selection tool internal state.

For example, consider the implementation of the selection tool. Using this tool, the user can select or unselect figures by clicking in the drawing area, or move the currently selected figures by dragging the mouse with the left button pressed. In order to implement

the finite state machine for the tool, as shown in Figure 3.5, the developer constructs the state machine out of ToolState and TransitionTable objects, and provides a command block for each state, as follows:

State 1: Compute the figure under the cursor. Add this figure to the set of currently selected figures if SHIFT is pressed, else make this figure be the current selection set. Also remember the current mouse cursor location lastPoint.

Tool states

```
at: 'Selection Tool Select'
put: (ToolState
  name: 'Selection Tool Select'
  command: [:tool :event |
    | drawing lastPoint figure |
    drawing := tool drawing.
    lastPoint := tool cursorPointFor: event.
    tool valueAt: #lastPoint put: lastPoint.
    figure := drawing figureAt: lastPoint.
    tool sensor shiftDown
      ifTrue: [drawing toggleSelection: figure]
      ifFalse: [(drawing isSelected: figure)
        ifFalse: [drawing selection: figure]]]).
```

State 2: Compute the current mouse location newPoint. Move the set of currently selected figures by the amount of newPoint - lastPoint. Make lastPoint be the value of newPoint.

Tool states

```
at: 'Selection Tool Move Figure'
put: (ToolState
  name: 'Selection Tool Move Figure'
  command: [:tool :event |
    | delta newPoint |
    newPoint := tool cursorPointFor: event.
    delta := newPoint - (tool valueAt: #lastPoint).
    tool valueAt: #lastPoint put: newPoint.
    tool drawing selections do: [:each |
      each translateBy: delta]]).
```

State 3: Do nothing.

```
Tool states
  at: 'End State'
  put: (EndToolState name: 'End State' command: [:tool :event | ]).
```

As seen from above, the entire tool behavior is realized without extending any of the *HotDraw* code. Tools are constructed from already available objects, which are configured with one-shot behaviors in the form of block objects, which are provided by the developer inline, in the context of the application that makes use of the *HotDraw* framework.

Java Abstract Windowing Toolkit

Java uses anonymous inner classes in the *Abstract Windowing Toolkit* and the *Swing* user interface frameworks. These frameworks provide a clear separation between the application and the user interface via an intermediary layer that links the two. This layer consists of callbacks that the application must register with the framework in order to respond to events in the user interface. This layer is implemented with inner classes in order to avoid having to subclass the user interface classes for every application and having to handle events in huge case statements.

Consider, for example, the following code which shows a portion from the implementation of a search dialog:

```
public class SearchDialog {
    private Frame frame;
    private Button searchButton;
    ...
    private addSearchButton() {
        searchButton = new Button("Search");
        searchButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    search();
                }
            }
        );
    }
}
```

```
        frame.add(searchButton);  
    }  
    ...  
    private void search() {...};  
}
```

The method `addSearchButton` is called during the initialization of a `SearchDialog` instance. This method creates the "Search" button to which it binds an action listener, which happens to be an instance of an anonymous inner class instance. Now, whenever the user presses the button, `actionPerformed` will be called by the framework to execute the `search` method.

3.7 Summary

This chapter contributes a system of software patterns towards the goal of this thesis. These patterns can be used to link user interfaces to application models in a consistent way. The *View Event Handler* pattern can be used in conjunction with the *Lambda* pattern to handle the events triggered by the interface, and *Complete Update* and *Multiple Update* patterns to update the user interface from the application model. In the next chapter we present translators that use these patterns to link user interfaces represented in a platform-independent format to application models.

Chapter 4

Methodology

Our objective is to simplify the development and maintenance of cross-platform user interfaces. To achieve this goal, we propose to represent user interfaces in a platform-independent format that can be easily translated to other platforms or directly interpreted. We introduced the first aspect of our approach in Chapter 3 where we presented a system of patterns that can be used to consistently attach user interfaces to application models. The second aspect of our approach is the development of the Interface Markup Language (IML), a platform-independent language for representing user interfaces. The third aspect is the development of translators that can read user interfaces written in IML and generate code for different target platforms. The generated code will contain the window definition in the target language accompanied by stubs that link the interface to the application model according to the patterns presented in Chapter 3.

The Interface Markup Language is derived from and supports all the features of the eXtensible Markup Language. We present an overview of the eXtensible Markup Language in Section 4.1, followed by the description of the Interface Markup Language in Section 4.2. Finally, in section 4.3 we discuss the implementation of concrete translators that convert Smalltalk user interfaces to the markup format, and convert the resulting IML document to Microsoft Windows resources, HTML forms, Java Swing and AWT interfaces.

4.1 XML Overview

The eXtensible Markup Language(XML) has been created by a working group of the World Wide Web Consortium as a subset of Standard Generalized Markup Language(SGML) and is intended to replace the HyperText Markup Language(HTML) as the format for information on the World Wide Web. The design of XML is based on years of experience with both SGML and HTML so the final draft, released in February 1998, represents a robust format that can be used to represent hierarchical structured data.

4.1.1 XML Structure

XML in itself is not a markup language. It is a meta-language designed to define special-purpose markup languages. XML allows certain items to be defined as part of a markup language. Definitions for these items are stored in the *document type definition*, or DTD file:

Elements. An element is simply a container of either text, elements, or a mixture of both. A person element for example can be represented as:

```
<PERSON>
  John Doe
  <ADDRESS>...address 1...</ADDRESS>
  <ADDRESS>...address 2...</ADDRESS>
</PERSON>
```

where the tag <PERSON> marks the start of the element, the text John Doe represents the content of the element along with two other address elements, and the tag </PERSON> marks the end of the element. Different tags, <ADDRESS> and </ADDRESS>, are used to represent the address elements which only contain text.

Although normal elements can be empty, some are designed to be empty by default. Such elements cannot contain text or any other elements and are represented with a single tag instead of start and end tags. These elements are marked up like <name /> where name is the name of the element.

Attributes. Attributes are used to add information to the elements. They are listed inside the start tag as a list of name/value pairs, where the name and value are separated by the equal sign, and the value is represented in either single or double quotes. For example, rather than represent the name of the person as content, we can represent it as attributes. The same can be done to the address element content:

```
<PERSON lastname="Doe" firstname="Joe">
  <ADDRESS street="..." number="..." />
  <ADDRESS street="..." number="..." />
</PERSON>
```

This new representation makes the person element more structured, and therefore easier to search. Some attributes may be required to have values, some attributes may not, and some other may have an implied value if one is not provided.

Entities. An entity is a mechanism to substitute individual characters, text, or images into a document. They are represented by the name of the entity preceded by the ampersand character and followed by a semicolon. For example, if the person elements are to use a default last name entity, they would look like:

```
<PERSON lastname=&defaultLastName; firstname="Joe">
  <ADDRESS street="..." number="..." />
  <ADDRESS street="..." number="..." />
</PERSON>
```

where `&defaultLastName;` is declared in the document type definition to represent the default last name.

A special type of entities, parameter entities, provide the same functionality as normal entities but can only be used inside the document type definition. A parameter entity used inside a document is treated as regular text.

Processing Instructions. A processing instruction encodes information to be used by the application that processes the information. They are identified by the name of the application and the application instruction. For example,

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

is used to instruct the application named `xml`, in this case the parser, that the information contained in the document follows the XML 1.0 standard.

Notations. A notation is used to specify non-XML data. It consists of a name and an attribute type that specifies that an element contains non-XML data. When the element is parsed, it is up to the processing application to do something with the element contents.

The *document type definition*, or DTD file, uniquely identifies a markup language. Such definition files exist for the Channel Definition Format(CDF) language used to store information about the Internet Channels, the Mathematical Markup Language(MML) used to express mathematical equations, and many others [9]. For the person example mentioned above, the definition file contains definitions for the person and address elements:

```
<!ELEMENT ADDRESS (#PCDATA)>
<!ELEMENT PERSON (ADDRESS*)>
```

For both the declaration of person and address elements, the definition includes a statement of what other elements can nest inside them. The address element can only contain elements of type `#PCDATA`, or character data, whereas person elements can contain other `ADDRESS` elements. Elements that can nest inside other elements are usually separated by `|` or `,` and followed by an occurrence indicator. A `+` indicates one or more, a `?` indicates zero or one, and a `*` indicates zero or more elements are allowed. In this case, we defined a person to have zero or more addresses.

The attributes for the person and address elements are declared as follows:

```
<!ATTLIST ADDRESS
  street CDATA #REQUIRED
  number CDATA #REQUIRED>
<!ATTLIST PERSON
  lastname CDATA #REQUIRED
  firstname CDATA #REQUIRED>
```


Attribute definitions first list the element that owns the attributes. Then, for every attribute, its name, type, and default value are declared. Attributes are usually declared of type CDATA, or character data and their default value is specified as either some text or as #IMPLIED or #REQUIRED. When the value is implied, the application that processes a document using this definition has to provide a value for this attribute if the element does not provide one. If the value is required, the element must provide a value for this attribute. For the person example, all attributes are declared as text and required to be specified.

The named entity defaultLastName for the last name is defined in the document definition file as follows:

```
<!ENTITY defaultLastName "Doe">
```

4.1.2 XML Linking

Besides the parent/child relationship imposed by the hierarchical nature of markup, it is sometimes useful to create relationships between sibling nodes or nodes at different positions in the hierarchy. This can be done by using the ID, IDREF, and IDREFS attribute types. The value of an attribute of type ID must be an XML name unique within the document. Elements can then point to other elements by having their attribute of type IDREF list the ID value of the element it links to (or a list of IDs as value for the IDREFS attribute if it links to more than one element):

```
<PERSON id="1024" lastname="Doe" firstname="John"/>
<PERSON idref="1024" lastname="Doe" firstname="Jane"/>
```

In this example, the person named Jane Doe is associated with person named John Doe. Note however, that the type of relationship is not specified. To overcome this problem, the World Wide Web Consortium currently has two other standards under work for linking information in XML, namely XLink [10] and XPointer [11].

XPointer

XPointers provide a mechanism to address the internal structures of XML documents. Any elements, text, and other parts of XML documents can be addressed either by absolute or

relative location, whether or not they provide an explicit ID value. An XPointer consists of a series of location terms, each of which specifies a location relative to a previous location term. Each location term has an XPointer keyword that indicates how the document tree is to be traversed and can have arguments such as an element index, type, or attribute:

```
root().child(1, PERSON).child(2, ADDRESS)
```

For this example, the location term returns the second child element of type ADDRESS of the first child of type PERSON or, the second address of the first person. The beginning address of an XPointer can be the root of the document tree or some element id (via the `root()` and `id(name)` keywords). Location can be specified relative to an element by identifying children, descendants, ancestors, and siblings paths (via the `child()`, `descendant()`, `ancestor()`, `preceding()`, `following()`, `psibling()`, and `fsibling()`). The relative location terms can take arguments an element index, type, or attribute.

XPointers have the major disadvantage that they depend on the stability of the document they point into. Any change that modifies the structure of the target document, or the order of elements in the target document, can change the information the XPointer references.

XLink

XLinks provide a mechanism to create relationships among the internal structures of an XML document. The relationships that can be created are either one-to-one, one-to-many in the sense that source and targets are specified. However, XLinks can provide relationships that just exist, for example, relationships that state something about an element or a group of elements. The relationships can also be unidirectional, bidirectional, or multidirectional.

Relationships are created by either introducing a link element into the document or adding link attributes to an existing element. The link element can be identified by the special attributes `xml:link` or `xml:attributes`. The `xml:link` is used to specify the type of the link, either simple or extended, and `xml:attributes` is used to map XLink attributes in case of collisions with already existing attribute names.

Simple links are used to specify one-to-one relationships and have the following definition:

```
<!ATTLIST simple
  xml:link CDATA #FIXED "simple"
  href CDATA #REQUIRED
  title CDATA #IMPLIED
  role CDATA #IMPLIED
  show (embed|replace|new) #IMPLIED
  actuate (auto|user) #IMPLIED
  behavior CDATA #IMPLIED
  content-role CDATA #IMPLIED
  content-title CDATA #IMPLIED
  inline (true|false) "true">
```

In order to locate XML documents and portions of documents, the locator value in the href attribute may contain either a Uniform Resource Identifier or a fragment identifier, or both. Any fragment identifier for pointing into the XML document must be an XPointer or an element name. If both the Uniform Resource Identifier and the document fragment are present, they must be separated by the symbol #.

The title and role attributes specify the title of the relationship and its role. The attributes show, actuate, and behavior indicate how the link should be traversed, for example, show indicates whether the information pointed to should be embedded in the current element or not, actuate indicates whether the information is to be updated automatically or not, and behavior can store special instructions understood by the processing application. If a link is declared inline, then the information the link points to is stored as the current element contents and its title and role are listed in the content-title and content-role attributes.

Extended links provide additional attributes that allow associations between any number of resources not just one local-to-remote resources. We will not discuss these links because they are not relevant to the structure of our proposed Interface Markup Language.

4.1.3 XML Processing

Given its origins in *Standard Generalized Markup Language*, XML is well suited for the creation of large structured documents. For example, for a book type definition, special elements can be created to represent chapters, sections, paragraphs, figures, and tables. These documents can then be converted to different other formats, for example, to HTML to be available on the World Wide Web, to PostScript to be printed, or to some index format to be available for search engines.

Document processing is usually performed by an XML processor, as shown in Figure 4.1. The processor uses a parser to parse the document(*.xml) and its corresponding document definition file(*.dtd) but it sometimes also parses a style file(*.css or *.xsl) that contains information on how the document file is to be processed. For example, if the person example from Section 4.1.1 is to be converted to HTML and the last name must be emphasized, then the last name attribute will be associated with the style in the style file.

Figure 4.1: XML processor.

One possible task an XML parser can perform on a document is to check whether it is valid and well-formed. A document is valid when it conforms to the rules in the data type definition file and is well-formed if it can be parsed according to the grammar rules provided in the XML specification.

4.1.4 User interfaces in XML

In summary, we chose to represent user interfaces using XML for the following reasons:

- It is an accepted, well understood technology that has been successfully used in a number of areas to represent hierarchical data.
- It provides a way to reference elements from other elements via the XLink and XPointer specifications.
- It is cross-platform technology. XML has actually been designed to represent structured information on the web, therefore it is portable by design.
- There are many tools to read, write, and query documents. These tools include document editors, word processors, and parsers that are available for the most popular platforms and programming languages.
- Existing parsers are designed to validate and verify documents based on their document type definition.

In the next section we overview our proposed Interface Markup Language, we describe its document type definition, and present the list of widgets the Interface Markup Language supports by default.

4.2 IML Overview

This section presents the design of the Interface Markup Language, a markup language defined in XML, whose purpose is to represent graphical user interfaces. IML is intended to serve as an exchange format for user interface tools, or as an intermediary format for translators that convert the static representation of graphical user interface between different platforms.

IML can represent both flat and hierarchical window structures. Widgets are represented as arbitrarily nested components, each with its own attributes. For example, the `ToDoList` example introduced in Chapter 3, Figure 3.2, is represented as follows using IML:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE IML SYSTEM "iml.dtd">
<IML>
  <WINDOW
    name="ToDoListWindow"
    caption="ToDoList"
    bounds="100 100 400 300">
    <TEXTBOX
      name="NewItemView"
      layout="0 0 100 0 12 12 -98 37"
      onChange="changedNewItemView">
    </TEXTBOX>
    <LISTBOX
      name="ItemsView"
      layout="0 0 100 100 12 43 -98 -12"
      onChange="changedItemsView">
    </LISTBOX>
    <PUSHBUTTON
      name="AddButton"
      label="Add"
      mnemonic="A"
      layout="100 0 100 0 -92 12 -12 37"
      onClicked="addToDoItem">
    </PUSHBUTTON>
    <PUSHBUTTON
      name="RemoveButton"
      label="Remove"
      mnemonic="R"
      layout="100 0 100 0 -92 43 -12 68"
      onClicked="removeToDoItem">
    </PUSHBUTTON>
  </WINDOW>
</IML>
```

In the following sections we present an overview of the document type definition declaring the XML element types used by the Interface Markup Language. The widgets, with their properties and events, defined in the document type definition, have been chosen

based on their availability in the most popular toolkits and frameworks. The full listing of the document type definition is included in Appendix A.

4.2.1 Widgets

The Interface Markup Language defines an XML element for every widget. The attributes of the XML element correspond to the most commonly used properties and events for that widget (the event attributes are distinguished from the property attributes by the <on> prefix). The top most widgets are either windows or dialogs, which along with forms, are the only widgets that can manage other widgets:

```
<!ENTITY % window.properties '
    caption CDATA #IMPLIED
    bounds CDATA #IMPLIED
    %component.properties;'>
<!ENTITY % window.events '
    %component.events;'>
<!ELEMENT WINDOW (GENERIC|PROPERTY|EVENT|FORM|TABBEDPANE|LABEL|
    PUSHBUTTON|RADIOBUTTON|CHECKBOX|LISTBOX|COMBOBOX|TEXTBOX|
    TOOLBAR|STATUSBAR|TABLE|TREE|SCROLLBAR|PROGRESSBAR|SLIDEBAR|
    MENU|REFERENCE)*>
<!ATTLIST WINDOW
    %window.properties;
    %window.events;>
```

This definition declares a window as a component element; for example, it includes all the component properties, such as font, backcolor, forecolor, and change events, such as open, close, activate, and deactivate. The window specific properties include its caption, and bounds. Dialogs and forms are defined in a similar fashion. The declarations for other widgets, such as push buttons, list boxes, and other widgets, located in Appendix A, follow a similar definition pattern. We will not discuss these in the interest of brevity.

Additionally, widgets must specify position information. Position can be either fixed or relative. The simplest form, fixed positioning, is specified via the bounds attribute, which contains four integers specifying the left, top, right, and bottom coordinates of the

widget. For relative positioning, the widgets are laid out according to a simple grid algorithm. Widgets are positioned on a relative grid inside a container area. The grid has origin $\langle 0, 0 \rangle$ and corner $\langle 100, 100 \rangle$ and is relative to the size of the container so, for example, when the container changes size, the grid coordinates will change pixel size. Additionally, the widgets can have pixel offsets for the left, top, right, and bottom coordinates into the relative grid. Relative positioning is specified via the `layout` attribute which contains eight integers representing four relative grid coordinates and four offset coordinates.

4.2.2 Generic Widgets

No matter how comprehensive the list of widgets is, there will always be a new component or a property that the IML specification does not support. To accomodate such situation, IML provides a definition for a generic widget:

```
<!ENTITY % generic.properties '
    layout CDATA #IMPLIED
    type CDATA #REQUIRED
    %label.properties;'>
<!ENTITY % generic.events '
    %label.events;'>
<!ELEMENT GENERIC (GENERIC|PROPERTY|EVENT|FORM|TABBEDPANE|LABEL|
    PUSHBUTTON|RADIOBUTTON|CHECKBOX|LISTBOX|COMBOBOX|TEXTBOX|
    TABLE|TREE|SCROLLBAR|PROGRESSBAR|SLIDEBAR|SEPARATOR|REFERENCE)*>
<!ATTLIST GENERIC
    * %generic.properties;
    %generic.events;>
```

Generic widgets cannot be validated by the parser, for example, the parser cannot check a generic widget element against its definition in the document type definition. Therefore, it is up to the application to validate and translate generic widgets properly. Generic widgets are identified by their type.

Generic widgets usually have generic properties and events. These can also be added to widgets defined by the IML specification. Again, it is up to the application to interpret these properly, since the parser cannot validate them against the document type definition.

Generic properties are identified by their type and value, and generic events by their type and selector:

```
<!ENTITY % property.properties '
    type CDATA #REQUIRED
    value CDATA #REQUIRED'>
<!ENTITY % property.events ''>
<!ELEMENT PROPERTY (#PCDATA)>
<!ATTLIST PROPERTY
    %property.properties;
    %property.events;>

<!ENTITY % event.properties '
    type CDATA #REQUIRED
    selector CDATA #REQUIRED'>
<!ENTITY % event.events ''>
<!ELEMENT EVENT (#PCDATA)>
<!ATTLIST EVENT
    %event.properties;
    %event.events;>
```

4.2.3 Reference Widgets

Besides the parent/child relationship imposed by the widget tree, it is sometimes useful to associate information between nodes at different positions in the hierarchy. The reason for this is to reuse some already developed user interface, or portions of it, as part of a new interface without having to directly copy its definition. The referenced portion of the user interface can be either local or remote to the current document. References are specified using simple XLink references, as follows:

```
<!ENTITY % reference.properties '
    xml:link (simple|extended) "simple"
    href CDATA #REQUIRED'>
<!ENTITY % reference.events ''>
<!ELEMENT REFERENCE (#PCDATA)>
<!ATTLIST REFERENCE
```

```
%reference.properties;  
%reference.events;>
```

The reference element declares two attributes. The first is `xml:link` which identifies the element as a link and is hardcoded to `simple`. The second attribute value, `href`, points to the referenced element. If the referenced widget is local to the document, the `href` specifies only the name of the target widget, if the widget is in another document, the `href` must contain the URI of the document and the name of the widget.

4.2.4 Tab order

The user can direct input focus to a particular widget either by clicking the mouse inside the widget, or by keyboard. The keyboard interface allows the user to cycle through a circular list of the widgets in the interface until the desired widget has input focus. The order of the widgets in the list is very important, for example, in a dialog, the widgets must be traversed left-to-right, then top-to-bottom, usually by pressing the Tab key.

The Interface Markup Language does not provide special constructs to specify the tab order for the widgets. Instead it relies on the order the widgets are specified. A translator can interpret the tab order for the widgets to be an inorder traversal of the widget tree.

4.2.5 Discussion

IML allows the specification of elements that match the most commonly used widgets in the available toolkits. For every widget class that IML represents, there is a special element named after that widget that also has a collection of the most commonly used properties and events for that widget. An alternative to this format would be to represent everything as a generic component composed of properties, events, and other generic components. The document type definition for the alternative format would look like the following:

```
<!ELEMENT COMPONENT (PROPERTY, EVENT, COMPONENT)*>  
<!ATTLIST COMPONENT  
    type CDATA #REQUIRED>  
<!ATTLIST PROPERTY  
    type CDATA #REQUIRED
```

```
    value CDATA #REQUIRED>
<!ATTLIST EVENT
    type CDATA #REQUIRED
    selector CDATA #REQUIRED>
```

Using the alternative format, the `ToDoList` example of Figure 3.2 can be represented as follows:

```
<?XML VERSION="1.0" STANDALONE="no"?>
<!DOCTYPE IML SYSTEM "iml.dtd">
<IML>
  <COMPONENT type="window"
    <ATTRIBUTE type="name" value="ToDoListWindow"/>
    <ATTRIBUTE type="caption" value="ToDoList"/>
    <ATTRIBUTE type="bounds" value="100 100 400 300"/>
    <COMPONENT type="textbox"
      <ATTRIBUTE type="name" value="NewItemView"/>
      <ATTRIBUTE type="layout" value="0 0 100 0 12 12 -98 37"/>
      <EVENT type="onChanged" value="changedNewItemView"/>
    </COMPONENT>
    ...
  </COMPONENT>
</IML>
```

We decided not to represent user interfaces using this format because its document type definition does not allow for proper validation. Using the above type definitions, user interfaces may contain duplicate components and attributes, or attributes with identical names but with different types. The format we propose in this thesis has detailed definitions for each widget, thus making it possible to perform full validation in the case when only standard widgets are used. The only shortcoming of our approach is that generic widgets, properties, and events are left to the processor to validate.

4.3 IML Translators

So far we introduced the Interface Markup Language as the platform-independent format to represent user interfaces. One of the objectives we have is to convert existing user

interfaces to the markup language, a process known as up-translation. The reason for this is so that we can perform down-translation, the process to convert user interface markup to different target frameworks, possibly implemented in different programming languages. Ideally, new user interfaces will be developed directly in IML, but legacy user interfaces have to be converted to IML by means of up-translation.

Up-translators are harder to implement than down-translators. This is mostly because they need to parse the user interface definition, and parsers for the source language may not be readily available. A better approach is to write a program in the source language that instantiates and traverses the window while exporting markup language to a file. We took the latter approach to implement a translator from VisualWorks Smalltalk to IML. This translator is introduced in Section 4.3.1.

Down-translators are easier to implement because generic XML parsers exist that can parse IML. We implemented our translators in Java, and the XML parser that we used has been implemented by Sun and offers both tree-based and event-based parsing. The tree-based parser builds a *Document Object Model* [9] parse tree as a data object in memory and hands this tree to application for processing. The event-based parser fires an event every time it encounters an element during the parsing process.

Figure 4.2: IML object model.

Rather than use the tree-based parser, we chose to use the event-based parser and construct our own widget tree. We chose to implement our own tree in order to have the nodes in the tree represent interface objects, as opposed to XML objects. The IML tree consists

of objects of type `IMLComponent`, as shown in Figure 4.2. Every component has properties, events, and other components. A component property has a type and value, whereas an event has a type and a method selector string.

Figure 4.3: IML translator object model.

The architecture for the translator is shown in Figure 4.3. The abstract class `IMLTranslator` calls the `IMLComponentBuilder` to build a component from an input source. The `IMLComponentBuilder` implements the event interface for the `XMLParser` and gets notified every time the parser encounters the start and end of an element. Internally, the component builder uses a stack to construct the component and, once constructed, hands it to `IMLTranslator` for processing. Concrete subclasses of `IMLTranslator` implement the actual processing.

We present down-translators to Microsoft Windows resources in Section 4.3.2, HTML forms in Section 4.3.3, and Java Swing/AWT frameworks in Sections 4.3.4 and 4.3.5.

4.3.1 Smalltalk-to-IML Translator

User interfaces in VisualWorks Smalltalk are built visually with the canvas tool by selecting widgets from a palette and placing them on the canvas. The canvas tool saves the entire

window structure as a static array in class methods, usually `windowSpec` under the *interface specs* protocol. For example, the specification for the `ToDoList` example introduced in Chapter 3 looks as follows:

```
!ToDoList class methods!
windowSpec
  "UIPainter new openOnClass: self andSelector: #windowSpec"
  <resource: #canvas>
  ^#(#FullSpec
    #window:
    #(#WindowSpec
      #label: 'Todo List'
      #bounds: #(#Rectangle 397 336 690 511 )
      #isEventDriven: true )
    #component:
    #(#SpecCollection
      #collection: #(
        #(#SequenceViewSpec
          #layout: #(#LayoutFrame 8 0 37 0 -92 1 -8 1 )
          #model: #itemsHolder
          #callbacksSpec:
            #(#UIEventCallbackSubSpec
              #valueChangeSelector: #changedItemsView )
          #useModifierKeys: true
          #selectionType: #highlight )
        #(#InputFieldSpec
          #layout: #(#LayoutFrame 8 0 8 0 -92 1 33 0 )
          #model: #newItemHolder
          #callbacksSpec:
            #(#UIEventCallbackSubSpec
              #valueChangeSelector: #changedNewItemView ) )
        #(#ActionButtonSpec
          #layout: #(#LayoutFrame -88 1 8 0 -8 1 33 0 )
          #name: #AddButton
          #model: #addToDoItem
          #label: 'Add'
          #defaultable: true )
        #(#ActionButtonSpec
```

```
#layout: (#LayoutFrame -88 1 37 0 -8 1 62 0 )  
#name: #RemoveButton  
#model: #removeToDoItem  
#label: 'Remove'  
#defaultable: true ) ) ) )
```

The user interface creation process starts with a call to the open class method, during which the `windowSpec` method is evaluated and the returned array is parsed and compiled into an instance of `UISpecification`. This object is further passed to an instance of `UIBuilder` which uses it along with the currently selected `UILookPolicy` and `UIFeelPolicy` to generate a composite object made of `View` instances.

In order to export VisualWorks user interfaces to IML, the array representation stored in the `windowSpec` method has to be parsed and expressed in terms of IML elements. In order to do this four classes have been implemented, `IMLComponent`, `IMLProperty`, `IMLEvent`, and `IMLSpecification`. The first three classes mirror the structure of IML elements directly and are only responsible for printing themselves out on a stream. An `IMLSpecification` object keeps track of the top element and prints out the proper file headers after which it recursively calls the print methods for the elements in the document.

Rather than parsing the static array directly, a `UISpecification` is first obtained from the array, and then traversed to generate the `IMLSpecification` object. The `UISpecification` uses the *Composite* [14] pattern to represent arbitrarily nested view specifications; subclasses of `UISpecification` store the information needed to generate subclasses of `View`, for example a `RadioButtonSpec` contains information about radio buttons and knows how to generate `RadioButtonView` instances. To traverse the `UISpecification`, an `IMLSpecEnumerator` has been implemented as a subclass of `UISpecEnumerator`; it makes use of the *Visitor* [14] pattern to traverse a specification tree and generate the `IMLSpecification` object.

4.3.2 IML-to-Resource Translator

The resource translator translates IML documents to Microsoft Windows resource formats. ResourceTranslator subclasses the abstract IMLTranslator and overrides the translation methods for every component. Each component translation method prints the component on a separate line, type first, followed by a resource identifier, the bounds of the widget, and its properties:

```
#include "resources.h"
ToDoListWindow DIALOG 50, 50, 200, 150
BEGIN
    EDITTEXT newItemView, 6, 6, 145, 12
    LISTBOX itemsView, 6, 21, 145, 122
    PUSHBUTTON "Add", addButton, 154, 6, 40, 12
    PUSHBUTTON "Remove", removeButton, 154, 21, 40, 12
END
```

The resource identifiers are defined as a sequence of integers in the `resources.h` file. This file can then be included into any C or C++ application so that widgets can be referenced by name not by some number:

```
#define ToDoListWindow 1000
#define newItemView 1001
#define itemsView 1002
#define addButton 1003
#define removeButton 1004
```

We had to overcome the following problems while implementing the IML-to-resource translator:

- *Windows resources are restricted to a flat structure.* To get around this problem, the translator prints all the components as children of the dialog resource.
- *Resources do not support window objects.* Resources have been traditionally used to represent only dialogs; for windows, the dialog resource is loaded and changed dynamically to a window. To circumvent this problem, IML window elements have to be printed out as dialogs.

- *Resources use absolute positioning.* Since resources only represent dialogs, and dialogs are not resizable, all the layout information has to be converted into normal coordinates.
- *Resources do not support all the IML widgets.* We solve this problem by printing the widgets that do not have a resource equivalent as user defined widgets.

4.3.3 IML-to-Forms Translator

The forms translator is implemented as a subclass of `IMLTranslator`. It overrides the translation methods for all the widgets to print out form elements. This is a direct translation from one markup language to another:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML><BODY>
<FORM>
  <INPUT
    type="text" name="newItemView"
    style="position: absolute;
      left: 12px; top: 12px; width: 290px; height: 25px;">
  <SELECT
    type="select" multiple name="itemsView"
    style="position: absolute;
      left: 12px; top: 43px; width: 290px; height: 245px;">
  <INPUT
    type="button" name="addButton" value="Add"
    style="position: absolute;
      left: 308px; top: 12px; width: 80px; height: 25px;">
  <INPUT
    type="button" name="removeButton" value="Remove"
    style="position: absolute;
      left: 308px; top: 43px; width: 80px; height: 25px;">
</FORM>
</BODY></HTML>
```

We had to overcome the following problems during the implementation of the IML-to-forms translator:

- *Forms are restricted to a flat structure.* To get around this problem, the translator prints all the components as children of the main form.
- *Forms do not support window and dialog objects.* Since forms are used inside a web browser, there is no point to have windows and dialogs. To circumvent this problem, IML window and dialog elements are printed out as forms.
- *Forms do not use positioning.* Forms elements are considered text elements, for example, placing forms between paragraph tags `<p>` and `</p>` makes them behave like paragraphs. However, with the latest stylesheet standards for absolute positioning, forms can be laid out with either absolute or relative positioning. The parser uses the `style` attribute to specify the absolute positioning¹ of all form elements.
- *Forms do not support all the IML widgets.* Since there is no way for the user to implement new form elements, the translator simply skips the widgets that do not map to any of the form elements.

4.3.4 IML-to-Swing Translator

`SwingTranslator` subclasses `IMLTranslator` and overrides the methods that perform widget translation. The translation is more elaborate than all the previous ones, as it requires the following steps, in the order specified:

1. *Print package imports.* The package import statements allow the Java compiler to find the path to the collaborators of the current class. The packages printed include the container packages, the AWT and Swing packages, as follows:

```
import java.util.*;
import java.lang.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
```

¹As of this writing, this only works in Microsoft Internet Explorer.

2. *Print class definition start.* This statement declares a new class. It gets printed when encountering either a window or a dialog element. The name of the class is created by concatenating the name of the element with the suffixes Window or Dialog. The `ToDoList` example class declaration prints as follows:

```
public class ToDoListWindow {
```

3. *Print widget variables.* The widget variables are declared private so they cannot be accessed in subclasses. The names of the widgets are created by concatenating the name of the element with the its Java type:

```
private JFrame jToDoListFrame;  
private JTextField jNewItemTextField;  
private JScrollPane jItemsScrollPane;  
private JList jItemsList;  
private JButton jAddButton;  
private JButton jRemoveButton;
```

4. *Print pattern variables.* The translator generates code following the *Complete Update* pattern presented in Section 3.3 and, more specific, the *Infinite Event-Handle-Update* variant. Therefore, in order to break infinite looping, a boolean variable is needed to flag the updates:

```
private boolean updating = false;
```

5. *Print pattern update methods.* The update helper methods take care of setting the updating state to either true or false. The `isUpdating` method is used to test whether the window is updating in order to avoid infinite loops:

```
private boolean isUpdating() {  
    return updating;  
}  
private void beginUpdating() {  
    updating = true;
```

```
}  
private void endUpdating() {  
    updating = false;  
}
```

The complete update method sets the updating variable to true and calls all the individual update methods:

```
private void update() {  
    beginUpdating();  
    updateNewItemTextField();  
    updateItemsScrollPane();  
    updateItemsList();  
    updateAddButton();  
    updateRemoveButton();  
    endUpdating();  
}
```

6. *Print pattern changed methods.* The changed methods will not execute if the window is updating. Also, every change method will invoke an update at the end, according to the *View Event Handler* pattern introduced in Section 3.2. The user has to write the code that stores the state of the local user-defined variables into the widgets:

```
private void changedItemsList() {  
    if(isUpdating()) return;  
    // user code begin  
    // user code end  
    update();  
}
```

7. *Print widget accessors.* The widget accessors are declared protected and are the preferred method to access the widget variables. It is better to use the accessors since they are responsible for lazy initializing the variables. For example, the accessor for the AddButton is printed as follows:

```
private JButton getAddButton() {
    if(jAddButton != null) return jAddButton;
    jAddButton = new JButton("Add");
    jAddButton.setName("AddButton");
    jAddButton.setMnemonic('A');
    jAddButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent theEvent) {
            addTodoItem();
        }
    });
    return jAddButton;
}
```

8. *Print the show() method.* This method is used to open the window or dialog. It knows which method returns the frame and delegates the show message to it:

```
public void show() {
    getTodoListFrame().show();
}
```

9. *Print the main() method.* This method is printed as a convenience method to aid testing the class. It simply creates an instance of the window or dialog class and opens it:

```
public static void main(String[] args) {
    TodoListWindow window = new TodoListWindow();
    window.show();
}
```

We had to overcome the following problems implementing the translator:

- *Java widgets do not support scrolling.* However scrolling can be added by wrapping the widget with a JScrollPane. Whenever an IML component has scrollbars, the translator has to print two widgets, one for the actual element, and one for the scroll wrapper.

- *Java uses a different layout mechanism.* Java lays out widgets by using layout managers, objects that implement the `LayoutManager` and `LayoutManager2` interfaces. To solve this problem, we implemented a `GridLayout` manager that mimics the IML layout mechanism and implemented the translator to generate layout code using this layout manager.

4.3.5 IML-to-AWT Translator

The `AwtTranslator` class subclasses `SwingTranslator`, and only overrides certain methods to print out the widgets differently. This is because both AWT and Swing offer the same functionality, but with slightly different widgets.

4.4 Summary

In this chapter we introduced the Interface Markup Language, a platform-independent language for representing user interfaces. IML can be easily translated between different frameworks implemented in different programming languages by:

- Writing up-translators to convert user interfaces from the source framework to the user interfaces expressed using the Interface Markup Language. We designed and implemented such a translator that converts VisualWorks Smalltalk user interfaces to IML.
- Writing down-translators that convert user interfaces from the Interface Markup Language representation to other frameworks and programming languages using the patterns outlined in Chapter 3. We designed and implemented different translators that convert IML to Windows resources, HTML forms, Swing and AWT frameworks.

Chapter 5

Related Work

In order to simplify the development and maintenance of cross-platform user interfaces, we proposed to represent user interfaces in a platform-independent format. We introduced such a format, implemented using the Interface Markup Language, in Chapter 4. However, using an intermediate, platform-independent format is not unique. In this chapter, we investigate how other projects have used similar formats to implement cross-platform interfaces. We present approaches that use a proprietary format for Microsoft Windows resources in section 5.1 and three approaches that use XML, the User Interface Markup Language (UIML) in section 5.2, the Swing Markup Language (XwingML) in section 5.3, and the Extensible User Language (XUL) in section 5.4.

We will analyze these approaches according to the same criteria that we introduced in Chapter 2:

Structure. We explain how the user interface is represented, and what the relationship between widgets is. Although some user interfaces represent widgets as a linear structure, most user interfaces arrange widgets in hierarchical fashion, which resembles the tree structure promoted by the *Composite* [14] pattern. Each widget has a parent (windows and dialogs usually don't have parents, or their parent is the desktop) and children. This information is used to traverse the widget tree and propagate events through the inheritance path.

Properties. We explain how the widgets can be customized. Widgets are data-centered so,

they can be configured by changing their property values without implementing any new code. Properties can be changed either internally by the application, or externally, by the user through interaction with the interface. Properties have well defined data types, the most common ones being boolean, string, and integer.

Layout. We explain how widgets are laid out spatially in relation to their parents and siblings, and how this relation changes when the parent resizes. Usually, the widgets keep track of their position and size relative to their parents. When the position or size of the parent changes, the widgets modify their position and size according to a layout algorithm. The simplest layout is absolute positioning, for example, the widget never changes position or size. Others include layout managers (Java), grids (VisualWorks), or struts and springs [5].

Updating. We explain how the user interface interacts with the application model. Usually, widgets update their properties when some application aspect changes, and then the display is updated to reflect the new state of the widget. Additionally, other parts of the application may be notified of the changes, if they previously registered an interest in the widget (via callbacks or event listeners).

5.1 Windows Resources

Resources represent binary data that can be compiled into the application executable file to be loaded and used at run-time. Both Windows and Mac OS provide resources formats for icons, cursors, menus, and most importantly windows and dialogs. Resources are used mostly because they can be generated with visual tools and can be localized easily by using different resource sets for different languages. However, resources only encode static information, and applications that need to modify the structure of menus, windows, and dialogs at run-time must provide additional code to do so.

Usually resources are created and edited by development tools that accompany the compiler for the target platform. Such tools include the resource editor that comes with the Visual C++ [24] compiler for Windows and ResEdit [7] and Resorcerer [23] tools available for MacOS. The resource editor for Windows generates a resource description file that is

compiled with the resource compiler `rc.exe` into its binary equivalent that can be later linked into the executable. ResEdit and Resorcerer do not use an intermediate text format, they directly generate the resource fork that can be later linked into the executable.

We have built the `ToDoList` example with the Visual C++ Resource Editor and the resource text is as follows:

```
IDD_DIALOG1 DIALOG DISCARDABLE 0, 0, 197, 138
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "ToDoList"
FONT 10, "System"
BEGIN
    DEFPUSHBUTTON    "&Add", IDADD, 151, 6, 40, 13
    PUSHBUTTON       "&Remove", IDREMOVE, 151, 21, 40, 13
    EDITTEXT         IDC_EDIT1, 6, 6, 142, 13, ES_AUTOHSCROLL
    LISTBOX           IDC_LIST1, 6, 21, 142, 111,
                    LBS_SORT | LBS_NOINTEGRALHEIGHT |
                    WS_HSCROLL | WS_VSCROLL | WS_TABSTOP
END
```

Structure

All resources are listed in the resource file one after another. Some of the resources contain an arbitrary mix of subparts that are specified using Pascal-like syntax, between `BEGIN` and `END` tags. For the `ToDoList` example presented in Chapter 3, the dialog contains subparts of type `DEFPUSHBUTTON`, `PUSHBUTTON`, `EDITTEXT`, and `LISTBOX` which are the equivalent of the push-button, text-edit, and list-box widgets.

Properties

The resource file format allows for both dialog and widgets properties to be specified easily. Properties are either numeric, alphanumeric, or boolean. Numeric and alphanumeric properties are specified inline after the property name; for example the caption property is `CAPTION` followed by the `"ToDoList"` string. The boolean properties are specified using C syntax for bitwise `|` and `&` operators to either enable or disable properties; for example the list box widget is sorted (`LBS_SORT`), has both horizontal and vertical scrollbars

(WS_HSCROLL and WS_VSCROLL), and is also a tab stop (WS_TABSTOP).

Layout

Windows resources do not contain layout information. They are designed to provide the shape of dialogs only, where dialogs do not resize. Resources can be instantiated in memory and dynamically attached to a resizing frame, i.e. a window frame, but in this case, the resize events and layout must be handled manually. However, for the dialog resources, absolute positioning of the widgets in the dialog must be specified. The four numbers in the definition of the dialog and the corresponding widgets specify the x, y coordinates and the width and height. The coordinates are dialog coordinates (a function of the dialog font so, if the font is changed, the dialog changes size accordingly). Pixels can be mapped to dialog units in the following way:

```
mapPixelsToDialogUnits
| baseUnits newX newY |
baseUnits := UserLibrary default getDialogBaseUnits.
newX := (self x * 4 / baseUnits lowWord) rounded.
newY := (self y * 8 / baseUnits highWord) rounded.
^newX@newY
```

Similarly, dialog units can be mapped to pixels as follows:

```
mapDialogUnitsToPixels
| baseUnits newX newY |
baseUnits := UserLibrary default getDialogBaseUnits.
newX := (self x * baseUnits lowWord / 4) rounded.
newY := (self y * baseUnits highWord / 8) rounded.
^newX@newY
```

Using these algorithms to map between pixels and dialog units, we get a rough conversion of 1 dialog unit per 2 pixels using the system font. Also, we must take in account the borders width (3 pixels all around by default) and title bar height (25 pixels by default).

Updating

Each resource is identified on the first line by the resource type and an id. A resource file usually contains many resources of the same type, but each resource has to be uniquely identified with a numeric id. For example, in the `ToDoList` resource listed at the beginning of Section 5.1, the resource is declared of type `DIALOG` and uniquely identified by `IDD_DIALOG1` which is defined in a header file as some integer:

```
#define IDD_DIALOG1 1001
#define IDADD        1002
#define IDREMOVE     1003
#define IDC_EDIT1     1004
#define IDC_LIST1     1005
```

The program calls API functions to load and use different resources based on the unique id of each resource. For example, in Windows, a dialog resource is loaded with the `LoadDialog` function, whereas in MacOS, the dialog resource is loaded with `GetCNewDialog`.

Once the resources are loaded, the application calls other APIs provided in the interface toolkit to store and retrieve information from the widgets. The interaction between the widget in the application can be done directly by the application through the toolkit API, or through some framework using the *Model-View-Controller* [4] or *Model-View-Presenter* [21] patterns, in which case, it is up to the framework to call the appropriate API functions.

5.1.1 Discussion

Windows resources were designed to share window, dialog, and menu resources across applications running on the Microsoft Windows operating system. Structurally, user interface resources are inferior to the Interface Markup Language, since the widgets cannot nest arbitrarily. One major disadvantage is that resources assume absolute widget positioning, so no layout information can be included. After the resource is loaded to memory and instantiated, the application can update the user interface using any mechanism it desires. This is different from the Interface Markup Language which provides a simple and consistent updating mechanism, which we outlined in Chapter 3.

5.2 User Interface Markup Language

Harmonia has developed the User Interface Markup Language [15], or UIML, as a generic solution to represent user interfaces across different types of networked devices. Such devices include not only personal computers but others, such as personal digital assistants and telephones, that provide a wide variety of input and output options. Each such type of device has to run customized user interfaces due to restrictions imposed by the actual hardware. For example, the interface for a personal computer is designed to display on a large resolution screen and take keyboard and mouse input, while the user interface for a personal digital assistant must display on a small resolution screen and accept pen input. In the case of telephones, the user interface is furthermore restricted to audio feedback, one or two line screen, and input from a numeric keypad only.

Harmonia introduces the concept of dynamic user interfaces to overcome the shortcomings of the current static user interfaces. The main disadvantage of static user interfaces is that they present the same interface to all users. Users must work and think the way the developers of the interface expect them to work, and their customization of the user interface is limited to setting colors, fonts, modifying the toolbars and hiding the toolbars and status bars. By contrast, dynamic user interfaces are both configurable and adaptable. The user interface can be configured either by the network administrator (all users will see the same interface initially) or by the user (each user will see a modified version of the interface suited to her own needs). Furthermore, since the same application is accessible from different types of devices, the application adapts its user interface to the device the user happens to be working on. The customization is performed based on a user profile that indicates the device being used, the user's role, the user's experience level and even the user's fatigue level.

This is all being done with the use of the User Interface Markup Language, a declarative language that specifies what should be present in the user interface, but not how it is to be represented. The user interface is represented as a set of interface elements with which the user interacts. The set of interface elements are arranged based on the user profile, and each interface element contains data (text, sound and images) used to communicate information to the user. The mapping between the interface elements and the actual interface supported

on the device is specified using a style sheet. The runtime interaction is done with events (local - between interface elements or global - between interface elements and objects that represent the application model).

User interfaces are described by UIML in five sections:

Description. The description section lists all the interface elements that the user interacts with. Each element has `name` and `class` attributes. The `class` attribute specifies the element type and the `name` attribute identifies an instance of the element. The `name` has to be unique within the scope of the user interface and the `type` has to be unique within the scope of the application. Both `name` and `class` are used for reference purposes, either by the application or by the style file. The section does not specify what widget the element should be mapped to or the actual function of the element.

For example, we can represent the `ToDoList` example from Chapter 3 as follows:

```
<description>
  <element name="ToDoList" class="Application"/>
  <element name="NewItem" class="TextData"/>
  <element name="Items" class="ListData"/>
  <element name="SelectedItem" class="ListSelectionData"/>
  <element name="AddToDoItem" class="Action"/>
  <element name="RemoveToDoItem" class="Action"/>
</description>
```

This listing describes the application model for the `ToDoList` example. The `ToDoList` itself is described by the element `ToDoList` of class `Application`. The application manipulates a list of `ToDo` items represented by the `Items` element. Out of these items, one item can be selected and one item can be edited. The former is represented by the element `SelectedItem` of type `ListSelectionData` and the latter by the element `NewItem` of type `TextData`. The application can add the edited item to the list of items or remove the selected item from the list of items. These are elements of type `Action` named `AddToDoItem` and `RemoveToDoItem`. The types of these elements have been chosen arbitrarily.

Structure. The structure section specifies what subset of elements from the description section are relevant to a certain device and how these elements are organized. The elements are identified by their name, as declared in the description section above. This section may also contain elements that enhance the user interface but do not receive or generate events, such as static text, images and menu separators.

For example, one possible structure for the `ToDoList` example looks like the following:

```
<structure>
  <element name="ToDoList">
    <element name="NewItem" />
    <element name="Items" />
    <element name="SelectedItem" />
    <element name="AddToDoItem" />
    <element name="RemoveToDoItem" />
  </element>
</structure>
```

This `ToDoList` structure allows for both selecting and editing of `ToDo` items. Furthermore, by including the `AddToDoItem` and `RemoveToDoItem` elements, new items can be added to the list and existent ones removed. By contrast, a `ToDoList` that only displays the items looks as follows:

```
<structure>
  <element name="ToDoList">
    <element name="Items" />
  </element>
</structure>
```

Data. The data section lists data that the application needs, mostly strings and images to display in the interface as labels for menu items or buttons. Each data item has a name attribute that uniquely identifies its content.

For example, `ToDoList` data is represented as follows:

```
<data>
  <content name="ToDoList">ToDoList</content>
  <content name="AddToDoItem">&Add</content>
  <content name="RemoveToDoItem">&Remove</content>
</data>
```

We chose to represent the label for the window as `ToDoList` and labels for the add and remove buttons as `&Add` and `&Remove` where the `&` character is placed before the mnemonic character.

Style. The style section contains information that is used to map the elements hierarchy to the user interface widgets specific to different device types. There usually is a style file for every device and toolkit that the user interface is expected to run on. Each item in this section maps the class of some element to some widget value on the target device and specifies the type of the mapping.

For example, here is the source that maps the data model for the `ToDoList` example to Java's Swing interface library:

```
<style>
  <attribute
    class="Application" type="rendering"
    value="javax.swing.JFrame"/>
  <attribute
    class="TextData" type="rendering"
    value="javax.swing.JTextField"/>
  <attribute
    class="ListData" type="rendering"
    value="javax.swing.JList"/>
  <attribute
    class="Action" type="rendering"
    value="javax.swing.JButton"/>
</style>
```

The `ToDoList` application itself is mapped to a `JFrame` object, elements of type `TextData` are mapped to `JTextField`, `ListData` elements to `JList`, and

Action elements to JButton objects. Other choices exist too, we could have mapped ListData elements to JTable or JComboBox objects and Action elements to JMenuItem objects.

Events. The events section describes the events responsible for the synchronization between elements and between elements and the application. Some events are device dependent but UIML allows these events to be written in generic terms that can be later mapped to device specific events via the style section. Each event has a trigger and one or more of source, destination and action. Events can be triggered either by the user, by the application, or by the underlying system. Other attributes include the name, which is unique within the interface description, and the class, used as an identifier into the style section.

The events for the TodoList example are declared as follows:

```
<events>
  <event
    name="ChangedNewItem" class="TextChanged"
    source="NewItem">
    <action target="TodoList" method="changedNewItem"/>
  </event>
  <event
    name="ChangedItems" class="ListChanged"
    source="Items">
    <action target="TodoList" method="changedItems"/>
  </event>
  <event
    name="AddTodoItem" class="ActionPerformed"
    source="AddTodoItem">
    <action target="TodoList" method="addTodoItem"/>
  </event>
  <event
    name="RemoveTodoItem" class="ActionPerformed"
    source="RemoveTodoItem">
    <action target="TodoList" method="removeTodoItem"/>
  </event>
```


</events>

The `ToDoList` `NewItem` and `Items` elements generate `TextChanged` and `ListChanged` events, whereas the `AddToDoItem` and `RemoveToDoItem` elements generate `ActionPerformed` events. All events have the application as the target, and perform the method specified by the `method` attribute.

UIML expresses interactions between users and the device via the elements listed in the description and structure sections. UIML does not make any assumptions about the user interface, such that an interface definition can be rendered on any interface medium. The runtime components consist of the interface server and the UIML renderer. The interface server combines the interface definition, style sheet and content from the database into a UIML description. The UIML renderer maps the UIML definition into a programming language and API such as Windows MFC or Java Swing. The runtime engine monitors all events and maps generic interface events to scripts or application methods. All text, images and sounds referenced in the data section are stored in a database, independent of the actual UIML file, for internationalization purposes.

Structure

The relationship between widgets is specified indirectly in the description and structure sections of the format. The description section lists all the interface elements that the user interacts with, and the structure section specifies which subset of these interface elements will be used on a certain device and how they are organized. The elements in the structure section are mapped at runtime to widgets according to mappings specified in the style section of the file and the resulting user interface can be traversed and manipulated in normal fashion.

Properties

Since interface elements are mapped to actual widgets only at runtime, the properties have to be mapped at runtime too. Interface element properties are usually specified in the style section of the file and may reference data from the data section. Since there is a style file

for every toolkit that the user interface will run on, the properties can be customized for every toolkit, so different properties can be set on different platforms.

Layout

Because interface elements are mapped to actual widgets only at runtime, the way these widgets are positioned with respect to their parent and siblings, and the way they behave when the parent resizes, is dependent on the target user interface framework that the elements get mapped to. Consequently, layout information is specified in the style file and gets mapped at runtime along with the widgets and their attributes.

Updating

User interface updating is performed by the application code. The only information specified in the UIML file are the events that can occur and which elements respond to what events. The events are described in the events section and get mapped at runtime to actual events available on the given platform.

5.2.1 Discussion

Overall, the User Interface Markup Language project is far more ambitious than the Interface Markup Language. UIML distinguishes between the application, application interface, and user interface. It represents the application interface in XML and the user interface by other means, such as cascading style sheets. This layering is necessary because UIML is designed to run on very different platforms, from cellular phones to computers, and the application interface may adapt itself to the resources provided by the host device, which may also trigger changes in the user interface. By contrast, the Interface Markup Language only describes the application user interface and the events it can trigger. However, we think that the Interface Markup Language can be used to describe the user interface part of an UIML-based application, especially on computers running standard graphical user interfaces.

5.3 Swing Markup Language

Bluestone has developed the Swing Markup Language [30], or XwingML, to allow users to build Java user interfaces based on the Swing framework in XML. These user interfaces are delivered to the client along with the application, where an interface renderer creates the user interface. Representing Java interfaces in XwingML has certain advantages. The user interface of an application can be changed without modifying the application code, and different user interfaces can be created depending on the platform that the application runs on. The downside of this approach is that dynamically generated user interfaces tend to be slow to create.

The XML Document Type Definition file for XwingML contains the definition for all the widgets and event listeners defined in the Java packages `javax.swing`, `javax.swing.event` and parts of `java.awt` and `java.awt.event`. The definition for `ToDoList` example introduced in Chapter 3 is represented as follows:

```
<XwingML>
  <Classes>
    <Instance name="ChangedNewItem" className="XMLChangedNewItem"/>
    <Instance name="ChangedItems" className="XMLChangedItems"/>
    <Instance name="AddToDoItem" className="XMLAddToDoItem"/>
    <Instance name="RemoveToDoItem" className="XMLRemoveToDoItem"/>
  </Classes>
  <JFrame
    name="ToDoListFrame" title="ToDoList"
    x="100" y="100" width="400" height="300">
    <GridBagLayout/>
    <JTextField
      name="NewItemField"
      documentListener="ChangedNewItem"
      gridx="RELATIVE" gridy="RELATIVE"
      gridwidth=1 gridheight=1
      fill="BOTH" insets="12, 12, 3, 3"
      anchor="CENTER"
      weightx="1.0" weighty="0.0">
    <JButton
      name="AddButton" text="Add"
```

```

        actionListener="AddTodoItem"
        gridx="RELATIVE" gridy="RELATIVE"
        gridwidth="REMAINDER" gridheight=1
        fill="BOTH" insets="12, 3, 3, 12"
        anchor="CENTER"
        weightx="0.0" weighty="0.0">
    <JScrollPane
        gridx="RELATIVE" gridy="RELATIVE"
        gridwidth=1 gridheight=1
        fill="BOTH" insets="3, 12, 12, 3"
        anchor="CENTER"
        weightx="1.0" weighty="1.0">
        <JList
            name="ItemsList"
            listSelectionListener="ChangedItems">
    </JScrollPane>
    <JButton
        name="RemoveButton" text="Remove"
        actionListener="RemoveTodoItem"
        gridx="RELATIVE" gridy="RELATIVE"
        gridwidth=1 gridheight=1
        fill="HORIZONTAL" insets="3, 3, 0, 12"
        anchor="NORTH"
        ipadx="10" ipady="0"
        weightx="0.0" weighty="0.0">
</JFrame>
</XwingML>

```

This listing defines a `JTextField` element for the Todo item text field, a `JList` element nested inside a `JScrollPane` element for the Todo items list, and `JButton` elements for the buttons that add and remove Todo items. The properties for each widget are specified as attributes to the corresponding elements, for example, layout information for the text field is specified with the attributes `gridx`, `gridy`, `gridwidth`, `gridheight`, `fill`, `insets`, `anchor`, `weightx`, and `weighty`. Since listeners can be reused between different widgets, they are declared separately, as a list of `Instance` elements inside a `Classes` element. Each listener has a name and a class name, where the

name is used for referencing within the XwingML definition, and the class name is the Java application class that implements the listener interface.

Since the Swing Markup Language directly mirrors the Swing user interface framework, the structure, properties, layout, and updating are identical to the Swing framework and have been described in Section 2.2.3.

5.3.1 Discussion

XwingML has been designed to represent only Swing interfaces. User interfaces represented with XwingML will only work with the Swing library, on the platforms that Swing works on. Translators can be written to convert user interfaces described with XwingML to other platforms but, we believe, XwingML does not cover user interfaces in a generic way. For example, XwingML cannot describe user defined widgets, properties, or events; in order for this to be possible, the XwingML document type definition has to be extended to include new types. By contrast, the Interface Markup Language does not have these shortcomings.

5.4 XML-based User Interface Language

The Mozilla Internet browser, HTML editor, mail and news-readers, and all other associated applications use a cross-platform library to implement a similar look and feel across all supported platforms. The library is named XPToolkit and provides the means to load and dynamically create a user interface from an external specification. The specification is coded using an XML-based User Interface Language, short named XUL [25].

Applications implemented using the toolkit consist of a series of packages, where each package is made of widgets (to provide the look and feel) and services (to provide functionality) that work together through the *Application Object Model* (AOM). Package configuration is loaded at runtime from a XUL file. In other words, the application provides the services, the toolkit provides the widgets and the means to interpret the XUL file, and the XUL file specifies how the user interface is built from the available widgets and how the services get triggered by the widgets.

A package is divided into five components:

Contents. The contents section consists of an XML description of the user interface. The XML describes the user interface in two namespaces, HTML and XUL, where XUL information extends the widgets provided by HTML. The contents of the `ToDoList` example introduced in Chapter 3 can be then specified as follows:

```
<window>
  <text></text>
  <select></select>
  <titledbutton value="Add"></titledbutton>
  <titledbutton value="Remove"></titledbutton>
</window>
```

Appearance. The appearance section contains information stored in Cascading Style Sheets format that can be used to customize the widgets stored in the contents section. For example, the size of the window in the `ToDoList` example can be specified using the following style:

```
#window {
  width: 400;
  height: 300;
}
```

This style can then be associated with the window component of the `ToDoList` by specifying the style in the class attribute of the element in the XUL file, as follows:

```
<window class="window">
</window>
```

Behavior. The behavior section specifies widget behavior, services, and the *Application Object Model*. The widget behavior is coded in JavaScript methods which get called when widgets trigger events. These event-to-method associations are declared using normal JavaScript syntax, as attributes to the widget element. The attribute name is

the name of the event and the attribute value is the name of the function that gets invoked when the event occurs. The `TodoList` example behavior can be coded as follows:

```
<window>
  <text onChange="changedNewItemView()"></text>
  <select onClick="changedItemsView()"></select>
  <titledbutton onClick="addTodoItem()"></titledbutton>
  <titledbutton onClick="removeTodoItem()"></titledbutton>
</window>
```

Services perform application-specific tasks. They are message based, where the messages are usually initiated by widgets. The implementation is really ignorant of the details of the user interface. It exposes functions that allow messages to be sent to the service, services can be used by multiple user interfaces. Services can be written in JavaScript, C++, or a mixture of both.

The *Application Object Model* is the means by which all applications in the Mozilla suite and XPTToolkit inter operate. It is the primary mechanism for working with the XPTToolkit from both compiled code and scripts. It defines objects that compose the user interface and the relationship between them. It is a superset of the *Document Object Model* [9], or DOM, and therefore can still be accessed using DOM protocol by applications that use a DOM-only interface.

DOM has been extended for the purposes of XUL with `XULDocument` and `XULElement` interfaces. JavaScript has also been extended with functions that are similar to the standard functions but take XUL specific arguments.

Locale. The locale section contains all localizable strings used by the user interface. Storing the strings in a separate section makes it easier to switch between different string tables for different locations without having to change the information from any other sections.

Platform. The platform section contains platform-specific information that augments the contents, appearance and behavior sections in order to make sure the user interface implements the necessary look and feel for the given platform.

By enforcing this separation among the components of the package, it becomes possible to swap out individual pieces of the package. For example, the look and feel of an existing package can be changed by replacing the appearance and platform sections. Even the behavior section can be replaced to achieve a different behavior for the same interface.

The next subsections describe XUL-based user interfaces according to the criteria set forward at the beginning of this chapter.

Structure

The structure of the user interface described in a XUL file is similar to the other approaches presented so far. There are two types of main widgets, windows and dialogs that can contain other types of widgets.

Properties

The properties of each widget are represented as XML-style attributes, inline with each element.

Layout

XUL performs layout using boxes. Boxes are ideal for a layout in which the size of the components can change, as is the case in which the labels text changes width because of localization or when the look and feel policy changes.

Boxes lay out their children using a constraint-based system in which fixed, relative, and intrinsic sizing of controls are supported. Widgets are arranged either horizontally left to right, or vertically top to bottom (the type of box is specified using the align attribute). All the widgets inside a horizontal box will take the height of the tallest widget and all objects inside a vertical box will take the width of the widest widget.

Fixed widget sizes can be set via the width and height attributes. As can be seen from the `ToDoList` example below, the width and height of a widget can be set in pixels. The other ways of specifying numeric values in XML, by percentage for example, are not supported.

Box containers also allow widgets that can change size and position. When there is extra space left in a box, a flexible widget will change size to fill the extra space. This flexibility factor is specified with the `flex` attribute, a numerical value that indicates how flexible a widget is. For example, a widget with a flex of 2 is twice as flexible as a widget with a flex of 1. This way, the space available in a box is divided among flexible widgets based on how flexible they are.

If no size and position information is specified, then the box layout algorithm will query the widgets inside the box for their preferred size and position. Windows and dialogs are programmed to resize themselves around their children widgets in such a way that they open as big as they need to be.

Widget size can be further constrained with the attributes `min.width`, `max.width`, `min.height`, `max.height`. These attributes only apply to flexible objects and specify the minimum and maximum allowed width and height of a widget.

For example, the layout for the `ToDoList` example introduced in Chapter 3 can be represented by using two vertical boxes, one to keep track of the text field and list box, and one for the push buttons:

```
<window>
  <box align="vertical" style="width: 100px">
    <text style="width: 100px; height: 100px"></text>
    <select style="width: 100px; height: 100px"></select>
  </box>
  <box align="vertical" style="width: 100px">
    <titledbutton style="width: 100px; height: 100px">
    </titledbutton>
    <titledbutton style="width: 100px; height: 100px">
    </titledbutton>
  </box>
</window>
```

Note that this definition does not explicitly state how the two vertical boxes are to be laid out. By default, windows and dialogs are designed to behave like boxes and have normal box attributes (in this definition, the window element is a horizontal box). Other widgets that behave like horizontal boxes include toolbars and menubars. Popup menus

and toolboxes are examples of widgets with vertical box properties.

Updating

Broadcasters and observers provide a mechanism that allows multiple components to observe a single element. The observed element can broadcast state and event information to its observers.

Any element in a XUL document can be a broadcaster or an observer. An element becomes an observer either by programming it after the XUL document is loaded or by the use of observer nodes. Observer nodes are made children of the element node that is to become an observer and provide a couple of attributes: the `element` attribute represents the id of the node of the component that broadcasts information, the `attribute` attribute represents the aspect that the observer is watching on the broadcaster, the `mapto` attribute is used to specify an observer attribute that will be set/unset with the value specified in the `attribute` attribute, and the `event` attribute to indicate an event that the observer is interested in.

5.4.1 Discussion

XUL has been designed to represent user interfaces in a platform independent manner only in the context of Mozilla's applications suite. Mozilla applications are designed around the *Application Object Model*, and XUL is tied into this framework also. By contrast, the Interface Markup Language does not make any assumptions about the underlying application model. The application handles events from the user interface and updates the interface using the generic software patterns presented in Chapter 3.

Chapter 6

Conclusion

The objective of the thesis was to simplify the development and maintenance of cross-platform user interfaces. The approach we pursued was to represent user interfaces in a platform-independent format that can be easily translated to any desired platform. We have made the following contributions to meet these objectives:

Interface Markup Language. We have developed a markup language to express user interfaces in a platform-independent format. This language is implemented using the eXtensible Markup Language, or XML, a new standard for representing information on the web and structured information in general.

The Interface Markup Language captures the structure, properties, and layout of platform-independent user interfaces. It can represent a widget tree as a hierarchical structure of XML elements, where the widget properties and events map into corresponding XML element attributes.

The widgets are laid out according to a simple grid algorithm. Widgets are positioned inside a container area on a grid with origin $\langle 0, 0 \rangle$ and corner $\langle 100, 100 \rangle$. The grid is relative to the size of the container so, when the container changes size, the grid coordinates will change pixel size. Additionally, the widgets can have pixel offsets into the relative grid.

User interfaces represented with the Interface Markup Language do not make any assumptions about the underlying application. The application handles events from

the user interface and updates the interface according to a system of software patterns.

The Interface Markup Language simplifies the development of cross-platform user interfaces by providing a unique representation format across platforms. The language can be used as an exchange format by user interface builders, or as an intermediate format for translators between platforms.

User interface patterns. We have developed a system of patterns that can be used to link user interfaces to application models in a consistent way. The *View Event Handler* pattern can be used in conjunction with the *Lambda* pattern to handle the events triggered by the interface, and *Complete Update* and *Multiple Update* patterns to update the user interface from the application model.

The *View Event Handler* pattern specifies that, in order for an application to handle event notifications from widgets, it should create an event-handler method that only pulls information from the widget. The event-handler method is a simple, one-shot behavior, and can be implemented using the *Lambda* pattern.

The *Complete Update* pattern states that, in order for the application to update the user interface, it should assume that the entire interface is out-of-date and update everything (push information into all the widgets). Since this may prove too expensive, special variants of this pattern make sure that only the interface parts that need updating are updated.

The *Multiple Update* pattern specifies that, in order for the application to update other widgets in other user interfaces, it should invoke the update behavior of its parent and child views. This should be done by maintaining a reference to the parent and children widgets, and requesting all of them to update when an update request is received.

These patterns simplify the development and maintenance of cross-platform user interfaces by providing implementation consistency not only across different views but across platforms too. By consistently using the push-pull mechanism, these patterns improve code legibility and are well-suited for code-generator output.

User interface translators. We have developed a framework that uses the patterns to convert the interface markup language to different toolkits and frameworks. Specific translators convert Smalltalk VisualWorks user interfaces to the markup language, and others convert the markup to operating system resources, HTML forms, Java Swing and AWT frameworks.

The translators reduce the cost of cross-platform interface development and maintenance by providing tools that perform Interface Markup Language up-translation and down-translation. Moreover, the framework is easily extensible, so additional translators can be added at any time.

6.1 Future Work

We would like to further develop our approach in the following ways:

- *Extend the Java facilities.* We would like to implement an interpreter that will generate user interfaces dynamically from interface markup descriptions. This approach is useful for client-server applications: the interpreter resides on the client and after it receives the application and its interface description, it generates the platform-specific user interface and attaches it to the application.
- *Extend the Interface Markup Language.* We would like to represent certain user interface aspects separately from the user interface. For example, fonts and colors can be represented in a style file; this way, the interface look can be changed only by changing the style file, not the user interface itself.

Appendix A

IML Document Type Definition

This appendix lists the document type definition for the Interface Markup Language presented in Section 4.2. The definition follows the standard W3C Recommendation for XML 1.0 from February 10, 1998 [8].

```
<!-- Interface Markup Language v1.0.0 -->

<!ENTITY % color.properties '
    foreColor CDATA #IMPLIED
    backColor CDATA #IMPLIED'>

<!ENTITY % font.properties '
    fontName CDATA #IMPLIED
    fontSize CDATA #IMPLIED
    fontStyle (regular|bold|italic) "regular">

<!ENTITY % mouse.events '
    onMouseUp CDATA #IMPLIED
    onMouseDown CDATA #IMPLIED
    onMouseMove CDATA #IMPLIED'>

<!ENTITY % keyboard.events '
    onKeyUp CDATA #IMPLIED
    onKeyDown CDATA #IMPLIED'>

<!ENTITY % keyboard.shortcuts '
    modifier1 (null|ctrl|alt|shift) "ctrl"
    modifier2 (null|ctrl|alt|shift) "null"
    character CDATA #REQUIRED'>
```

```

<!ENTITY % component.properties '
    name CDATA #IMPLIED
    enabled (true|false) "true"
    %color.properties;
    %font.properties;'>

<!ENTITY % component.events '
    onChanged CDATA #IMPLIED
    %mouse.events;
    %keyboard.events;'>

<!ENTITY % text.label.properties '
    label CDATA #IMPLIED
    mnemonic CDATA #IMPLIED
    horizontalTextAlignment (left|right|center) "right"
    verticalTextAlignment (top|bottom|center) "center">

<!ENTITY % image.label.properties '
    image CDATA #IMPLIED
    horizontalImageAlignment (left|right|center) "left"
    verticalImageAlignment (top|bottom|center) "center">

<!ENTITY % label.properties '
    %text.label.properties;
    %image.label.properties;
    %component.properties;'>
<!ENTITY % label.events '
    %component.events;'>
<!ELEMENT LABEL (#PCDATA|PROPERTY|EVENT)*>
<!ATTLIST LABEL
    layout CDATA #IMPLIED
    %label.properties;
    %label.events;>

<!ENTITY % button.properties '
    layout CDATA #IMPLIED
    %label.properties;
    %keyboard.shortcuts;'>
<!ENTITY % button.events '
    onClicked CDATA #IMPLIED
    %label.events;'>
<!ELEMENT PUSHBUTTON (#PCDATA|PROPERTY|EVENT)*>
<!ATTLIST PUSHBUTTON
    %button.properties;
    %button.events;>

```

```
<!ENTITY % radiobutton.properties '
    layout CDATA #IMPLIED
    value (true|false) "false"
    %label.properties;
    %keyboard.shortcuts;'>
<!ENTITY % radiobutton.events '
    onClicked CDATA #IMPLIED
    %label.events;'>
<!ELEMENT RADIOBUTTON (#PCDATA|PROPERTY|EVENT)*>
<!ATTLIST RADIOBUTTON
    %radiobutton.properties;
    %radiobutton.events;>

<!ENTITY % checkbox.properties '
    layout CDATA #IMPLIED
    value (true|false) "false"
    %label.properties;
    %keyboard.shortcuts;'>
<!ENTITY % checkbox.events '
    onClicked CDATA #IMPLIED
    %label.events;'>
<!ELEMENT CHECKBOX (#PCDATA|PROPERTY|EVENT)*>
<!ATTLIST CHECKBOX
    %checkbox.properties;
    %checkbox.events;>

<!ENTITY % combobox.properties '
    layout CDATA #IMPLIED
    editable (true|false) "false"
    %label.properties;'>
<!ENTITY % combobox.events '
    onClicked CDATA #IMPLIED
    %label.events;'>
<!ELEMENT COMBOBOX (#PCDATA|PROPERTY|EVENT)*>
<!ATTLIST COMBOBOX
    %combobox.properties;
    %combobox.events;>

<!ENTITY % textbox.properties '
    layout CDATA #IMPLIED
    %label.properties;'>
<!ENTITY % textbox.events '
    %label.events;'>
<!ELEMENT TEXTBOX (#PCDATA|PROPERTY|EVENT)*>
<!ATTLIST TEXTBOX
    %textbox.properties;
    %textbox.events;>
```



```
<!ENTITY % scroll.properties '
    horizontalScrollbar (true|false) "true"
    verticalScrollbar (true|false) "true"
    %label.properties;'>
<!ENTITY % scroll.events '
    %label.events;'>

<!ENTITY % listbox.properties '
    layout CDATA #IMPLIED
    %scroll.properties;'>
<!ENTITY % listbox.events '
    %scroll.events;'>
<!ELEMENT LISTBOX (#PCDATA|PROPERTY|EVENT)*>
<!ATTLIST LISTBOX
    %listbox.properties;
    %listbox.events;>

<!ENTITY % textedit.properties '
    layout CDATA #IMPLIED
    %scroll.properties;'>
<!ENTITY % textedit.events '
    %scroll.events;'>
<!ELEMENT TEXTEDIT (#PCDATA|PROPERTY|EVENT)*>
<!ATTLIST TEXTEDIT
    %textedit.properties;
    %textedit.events;>

<!ENTITY % table.properties '
    layout CDATA #IMPLIED
    %scroll.properties;'>
<!ENTITY % table.events '
    %scroll.events;'>
<!ELEMENT TABLE (#PCDATA|PROPERTY|EVENT)*>
<!ATTLIST TABLE
    %table.properties;
    %table.events;>

<!ENTITY % tree.properties '
    layout CDATA #IMPLIED
    %scroll.properties;'>
<!ENTITY % tree.events '
    %scroll.events;'>
<!ELEMENT TREE (#PCDATA|PROPERTY|EVENT)*>
<!ATTLIST TREE
    %tree.properties;
    %tree.events;>
```

```
<!ENTITY % range.properties '
    minimum CDATA #IMPLIED
    maximum CDATA #IMPLIED
    position CDATA #IMPLIED
    step CDATA #IMPLIED
    %label.properties;'>

<!ENTITY % range.events '
    %label.events;'>

<!ENTITY % scrollbar.properties '
    layout CDATA #IMPLIED
    orientation (horizontal|vertical) "horizontal"
    %range.properties;'>
<!ENTITY % scrollbar.events '
    %range.events;'>
<!ELEMENT SCROLLBAR (#PCDATA|PROPERTY EVENT)*>
<!ATTLIST SCROLLBAR
    %scrollbar.properties;
    %scrollbar.events;>

<!ENTITY % progressbar.properties '
    layout CDATA #IMPLIED
    orientation (horizontal|vertical) "horizontal"
    %range.properties;'>
<!ENTITY % progressbar.events '
    %range.events;'>
<!ELEMENT PROGRESSBAR (#PCDATA|PROPERTY EVENT)*>
<!ATTLIST PROGRESSBAR
    %progressbar.properties;
    %progressbar.events;>

<!ENTITY % sidebar.properties '
    layout CDATA #IMPLIED
    orientation (horizontal|vertical) "horizontal"
    minorScale CDATA #IMPLIED
    majorScale CDATA #IMPLIED
    %range.properties;'>
<!ENTITY % sidebar.events '
    %range.events;'>
<!ELEMENT SLIDEBAR (#PCDATA|PROPERTY EVENT)*>
<!ATTLIST SLIDEBAR
    %sidebar.properties;
    %sidebar.events;>

<!ENTITY % separator.properties ''>
```

```

<!ENTITY % separator.events ''>
<!ELEMENT SEPARATOR (#PCDATA)>
<!ATTLIST SEPARATOR
    %separator.properties;
    %separator.events;>

<!ENTITY % toolbar.properties '
    layout CDATA #IMPLIED
    position (left|top|right|bottom) "top"
    floating (true|false) "true"
    %component.properties;'>
<!ENTITY % toolbar.events '
    %component.events;'>
<!ELEMENT TOOLBAR (PROPERTY|EVENT|PUSHBUTTON|
    RADIOBUTTON|CHECKBOX|COMBOBOX|LABEL|SEPARATOR)*>
<!ATTLIST TOOLBAR
    %toolbar.properties;
    %toolbar.events;>

<!ENTITY % statusbar.properties '
    layout CDATA #IMPLIED
    %component.properties;'>
<!ENTITY % statusbar.events '
    %component.events;'>
<!ELEMENT STATUSBAR (PROPERTY|EVENT|LABEL|SEPARATOR)*>
<!ATTLIST STATUSBAR
    %statusbar.properties;
    %statusbar.events;>

<!ENTITY % menu.properties '
    %label.properties;'>
<!ENTITY % menu.events '
    %label.events;'>
<!ELEMENT MENU (PROPERTY|EVENT|MENU|PUSHBUTTON|
    RADIOBUTTON|CHECKBOX|SEPARATOR)*>
<!ATTLIST MENU
    %menu.properties;
    %menu.events;>

<!ENTITY % window.properties '
    caption CDATA #IMPLIED
    bounds CDATA #IMPLIED
    %component.properties;'>
<!ENTITY % window.events '
    %component.events;'>
<!ELEMENT WINDOW (GENERIC|PROPERTY|EVENT|FORM|TABBEDPANE|LABEL|
    PUSHBUTTON|RADIOBUTTON|CHECKBOX|LISTBOX|COMBOBOX|TEXTBOX|

```

```

    TOOLBAR|STATUSBAR|TABLE|TREE|SCROLLBAR|PROGRESSBAR|SLIDEBAR|
    MENU|REFERENCE)*>
<!ATTLIST WINDOW
    %window.properties;
    %window.events;>

<!ENTITY % dialog.properties '
    caption CDATA #IMPLIED
    bounds CDATA #IMPLIED
    %component.properties;'>
<!ENTITY % dialog.events '
    %component.events;'>
<!ELEMENT DIALOG (GENERIC|PROPERTY|EVENT|FORM|TABBEDPANE|LABEL|
    PUSHBUTTON|RADIOBUTTON|CHECKBOX|LISTBOX|COMBOBOX|TEXTBOX|
    TOOLBAR|STATUSBAR|TABLE|TREE|SCROLLBAR|PROGRESSBAR|SLIDEBAR|
    MENU|REFERENCE)*>
<!ATTLIST DIALOG
    %dialog.properties;
    %dialog.events;>

<!ENTITY % form.properties '
    layout CDATA #IMPLIED
    %label.properties;'>
<!ENTITY % form.events '
    %label.events;'>
<!ELEMENT FORM (GENERIC|PROPERTY|EVENT|FORM|TABBEDPANE|LABEL|
    PUSHBUTTON|RADIOBUTTON|CHECKBOX|LISTBOX|COMBOBOX|TEXTBOX|
    TABLE|TREE|SCROLLBAR|PROGRESSBAR|SLIDEBAR|REFERENCE)*>
<!ATTLIST FORM
    %form.properties;
    %form.events;>

<!ENTITY % tabbedpane.properties '
    layout CDATA #IMPLIED
    %label.properties;'>
<!ENTITY % tabbedpane.events '
    %label.events;'>
<!ELEMENT TABBEDPANE (GENERIC|PROPERTY|EVENT|FORM|TABBEDPANE|
    LABEL|PUSHBUTTON|RADIOBUTTON|CHECKBOX|LISTBOX|COMBOBOX|
    TEXTBOX|TABLE|TREE|SCROLLBAR|PROGRESSBAR|SLIDEBAR|REFERENCE)*>
<!ATTLIST TABBEDPANE
    %tabbedpane.properties;
    %tabbedpane.events;>

<!ENTITY % generic.properties '
    layout CDATA #IMPLIED
    type CDATA #REQUIRED

```

```

    %label.properties;'>
<!ENTITY % generic.events '
    %label.events;'>
<!ELEMENT GENERIC (GENERIC|PROPERTY|EVENT|FORM|TABBEDPANE|LABEL|
    PUSHBUTTON|RADIOBUTTON|CHECKBOX|LISTBOX|COMBOBOX|TEXTBOX|
    TABLE|TREE|SCROLLBAR|PROGRESSBAR|SLIDEBAR|SEPARATOR|REFERENCE)*>
<!ATTLIST GENERIC
    %generic.properties;
    %generic.events;>

<!ENTITY % property.properties '
    type CDATA #REQUIRED
    value CDATA #REQUIRED'>
<!ENTITY % property.events ''>
<!ELEMENT PROPERTY (#PCDATA)>
<!ATTLIST PROPERTY
    %property.properties;
    %property.events;>

<!ENTITY % event.properties '
    type CDATA #REQUIRED
    selector CDATA #REQUIRED'>
<!ENTITY % event.events ''>
<!ELEMENT EVENT (#PCDATA)>
<!ATTLIST EVENT
    %event.properties;
    %event.events;>

<!ENTITY % reference.properties '
    xml:link (simple extended) "simple"
    href CDATA #REQUIRED'>
<!ENTITY % reference.events ''>
<!ELEMENT REFERENCE (#PCDATA)>
<!ATTLIST REFERENCE
    %reference.properties;
    %reference.events;>

<!ELEMENT IML (GENERIC|PROPERTY|EVENT|WINDOW|DIALOG|FORM|
    TABBEDPANE|LABEL|PUSHBUTTON|RADIOBUTTON|CHECKBOX|LISTBOX|
    COMBOBOX|TEXTBOX|TOOLBAR|STATUSBAR|TABLE|TREE|SCROLLBAR|
    PROGRESSBAR|SLIDEBAR|MENU)*>

```

Appendix B

IML TodoList Example

This appendix lists the source code for the `TodoList` example depicted in Figure B.1 and introduced in Chapter 3. The `TodoList` application consists of a collection of todo items, where each item is a `String`. The user can add new items to the list by typing the new todo item in the text field and clicking the `Add` button. The user can also remove items from the list by selecting an item and clicking the `Remove` button. The `Add` button is disabled if the text field is empty, and the `Remove` button is disabled when there is no selection in the todo items list.

Figure B.1: `TodoList` example.

In Section B.1 we list the definition of the TodoList example expressed using the Interface Markup Language introduced in Section 4.2. In Section B.2 we list the source for the working TodoListWindow application. This source code has been generated with the IML-to-Swing translator presented in Section 4.3.4 and augmented with the required code to make the application work. The code makes use of two helper classes, GridLayout and GridConstraint, whose source we list in Sections B.3 and B.4.

B.1 IML Source

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE IML SYSTEM "iml.dtd">
<IML>
<WINDOW
  name="TodoListWindow"
  caption="TodoList"
  bounds="100 100 400 300">
  <TEXTBOX
    name="NewItemView"
    layout="0 0 100 0 12 12 -98 37"
    onChange="changedNewItemView">
  </TEXTBOX>
  <LISTBOX
    name="ItemsView"
    layout="0 0 100 100 12 43 -98 -12"
    onChange="changedItemsView">
  </LISTBOX>
  <PUSHBUTTON
    name="AddButton"
    label="Add"
    mnemonic="A"
    layout="100 0 100 0 -92 12 -12 37"
    onClicked="addTodoItem">
  </PUSHBUTTON>
  <PUSHBUTTON
    name="RemoveButton"
    label="Remove"
    mnemonic="R"
    layout="100 0 100 0 -92 43 -12 68"
    onClicked="removeTodoItem">
  </PUSHBUTTON>
</WINDOW>
</IML>
```

B.2 TodoListWindow Source

```
import java.util.*;
import java.lang.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class TodoListWindow {
    private JFrame jTodoListFrame = null;
    private JTextField jNewItemView = null;
    private JScrollPane jItemsScrollPane = null;
    private JList jItemsView = null;
    private JButton jAddButton = null;
    private JButton jRemoveButton = null;

    private boolean updating = false;

    private String newItem = null;
    private Vector items = null;
    private String selectedItem = null;

    private JFrame getTodoListFrame() {
        if(jTodoListFrame != null) return jTodoListFrame;
        jTodoListFrame = new JFrame();
        jTodoListFrame.setTitle("TodoList");
        jTodoListFrame.setName("TodoListFrame");
        jTodoListFrame.setBounds(100, 100, 400, 300);
        jTodoListFrame.getContentPane().add(getNewItemView());
        jTodoListFrame.getContentPane().add(getItemsViewScrollPane());
        jTodoListFrame.getContentPane().add(getAddButton());
        jTodoListFrame.getContentPane().add(getRemoveButton());

        GridLayout layout = new GridLayout();
        layout.setConstraints(getNewItemView(), 0, 0, 100, 0, 12, 12, -98, 37);
        layout.setConstraints(getItemsViewScrollPane(), 0, 0, 100, 100, 12, 43, -98, -12);
        layout.setConstraints(getAddButton(), 100, 0, 100, 0, -92, 12, -12, 37);
        layout.setConstraints(getRemoveButton(), 100, 0, 100, 0, -92, 43, -12, 68);
        jTodoListFrame.getContentPane().setLayout(layout);
        jTodoListFrame.addWindowListener(new WindowListener() {
            public void windowOpened(WindowEvent event) {
                update();
            }
            public void windowClosing(WindowEvent event) {
                System.exit(0);
            }
        });
    }
}
```



```
    }
    public void windowClosed(WindowEvent event) {
        // do nothing
    }
    public void windowIconified(WindowEvent event) {
        // do nothing
    }
    public void windowDeiconified(WindowEvent event) {
        // do nothing
    }
    public void windowActivated(WindowEvent event) {
        // do nothing
    }
    public void windowDeactivated(WindowEvent event) {
        // do nothing
    }
}
});
return jTodoListFrame;
}

private JTextField getNewItemView() {
    if(jNewItemView != null) return jNewItemView;
    jNewItemView = new JTextField();
    jNewItemView.setName("NewItemView");
    jNewItemView.getDocument().addDocumentListener(new DocumentListener() {
        public void changedUpdate(DocumentEvent event) {
            changedNewItemView();
        }
        public void insertUpdate(DocumentEvent event) {
            changedNewItemView();
        }
        public void removeUpdate(DocumentEvent event) {
            changedNewItemView();
        }
    });
    return jNewItemView;
}

private JScrollPane getItemViewScrollPane() {
    if(jItemsScrollPane != null) return jItemsScrollPane;
    jItemsScrollPane = new JScrollPane(
        getItemView(),
        JScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS);
    jItemsScrollPane.setName("ItemsScrollPane");
    return jItemsScrollPane;
}
```

```
private JList getItemsView() {
    if(jItemsView != null) return jItemsView;
    jItemsView = new JList();
    jItemsView.setName("ItemsView");
    jItemsView.addListSelectionListener(new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent event) {
            changedItemsView();
        }
    });
    return jItemsView;
}

private JButton getAddButton() {
    if(jAddButton != null) return jAddButton;
    jAddButton = new JButton();
    jAddButton.setName("AddButton");
    jAddButton.setText("Add");
    jAddButton.setMnemonic('A');
    jAddButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            addTodoItem();
        }
    });
    return jAddButton;
}

private JButton getRemoveButton() {
    if(jRemoveButton != null) return jRemoveButton;
    jRemoveButton = new JButton();
    jRemoveButton.setName("RemoveButton");
    jRemoveButton.setText("Remove");
    jRemoveButton.setMnemonic('R');
    jRemoveButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            removeTodoItem();
        }
    });
    return jRemoveButton;
}

private boolean isUpdating() {
    return updating;
}

private void beginUpdating() {
    updating = true;
}
```

```
}

private void endUpdating() {
    updating = false;
}

private void update() {
    beginUpdating();
    updateNewItemView();
    updateItemsViewScrollPane();
    updateItemsView();
    updateAddButton();
    updateRemoveButton();
    endUpdating();
}

private void updateNewItemView() {
    //user code begin
    if(!jNewItemView.getText().equals(newItem)) {
        jNewItemView.setText(newItem);
    }
    //user code end
}

private void updateItemsViewScrollPane() {
    //user code begin
    //user code end
}

private void updateItemsView() {
    //user code begin
    jItemsView.setListData(items);
    jItemsView.setSelectedValue(selectedItem, true);
    //user code end
}

private void updateAddButton() {
    //user code begin
    jAddButton.setEnabled(newItem != null);
    //user code end
}

private void updateRemoveButton() {
    //user code begin
    jRemoveButton.setEnabled(selectedItem != null);
    //user code end
}
```

```
private void changedNewItemView() {
    if(isUpdating()) return;
    //user code begin
    if(jNewItemView.getText().length() == 0) {
        newItem = null;
    } else {
        newItem = jNewItemView.getText();
    }
    //user code end
    update();
}

private void changedItemsViewScrollPane() {
    if(isUpdating()) return;
    //user code begin
    //user code end
    update();
}

private void changedItemsView() {
    if(isUpdating()) return;
    //user code begin
    selectedItem = newItem = (String)jItemsView.getSelectedValue();
    //user code end
    update();
}

private void changedAddButton() {
    if(isUpdating()) return;
    //user code begin
    //user code end
    update();
}

private void changedRemoveButton() {
    if(isUpdating()) return;
    //user code begin
    //user code end
    update();
}

private void addTodoItem() {
    iters.addElement(newItem);
    selectedItem = newItem;
    update();
}
```

```
private void removeTodoItem() {
    items.removeElement(selectedItem);
    selectedItem = null;
    newItem = null;
    update();
}

public TodoListWindow() {
    initialize(new Vector(), null);
}

public TodoListWindow(Vector items, String selectedItem) {
    initialize(items, selectedItem);
}

private void initialize(Vector items, String selectedItem) {
    this.items = items;
    this.selectedItem = newItem = selectedItem;
}

public void show() {
    getTodoListFrame().show();
}

public static void main(String[] args) {
    Vector items = new Vector();
    items.addElement("private release testing");
    items.addElement("integrate changes");
    items.addElement("global release testing");
    items.addElement("package application");
    items.addElement("ship fixes to clients");
    String selectedItem = (String)items.elementAt(3);

    TodoListWindow window = new TodoListWindow(items, selectedItem);
    window.show();
}
} // EOF
```

B.3 GridLayout Source

```
import java.util.*;
import java.awt.*;
```

```
public class GridLayout implements LayoutManager2 {
    Hashtable components = new Hashtable();
    Hashtable constraints = new Hashtable();

    public void addLayoutComponent(Component comp, Object cons) {
        if(comp != null) components.put(comp.getName(), comp);
        if(cons != null) constraints.put(comp.getName(), cons);
    }

    public void addLayoutComponent(String name, Component comp) {
        components.put(name, comp);
        constraints.put(name, new GridConstraint());
    }

    public float getLayoutAlignmentX(Container parent) {
        return (float)0.5;
    }

    public float getLayoutAlignmentY(Container parent) {
        return (float)0.5;
    }

    public void invalidateLayout(Container parent) {
        // Do nothing.
    }

    public void layoutContainer(Container parent) {
        Insets insets = parent.getInsets();
        float width = (float)(parent.getSize().width - insets.left - insets.right) / 100;
        float height = (float)(parent.getSize().height - insets.top - insets.bottom) / 100;

        float leftPosition, rightPosition;
        float topPosition, bottomPosition;

        Enumeration enum = components.keys();
        while(enum.hasMoreElements()) {
            String name = (String)enum.nextElement();
            Component comp = (Component)components.get(name);
            GridConstraint cons = (GridConstraint)constraints.get(name);

            leftPosition = width * cons.leftPosition + cons.leftOffset + insets.left;
            rightPosition = width * cons.rightPosition + cons.rightOffset + insets.left;
            topPosition = height * cons.topPosition + cons.topOffset + insets.top;
            bottomPosition = height * cons.bottomPosition + cons.bottomOffset + insets.top;

            comp.setLocation((int)leftPosition, (int)topPosition);
        }
    }
}
```

```
        comp.setSize(
            (int)(rightPosition - leftPosition),
            (int)(bottomPosition - topPosition));
    }
}

public Dimension maximumLayoutSize(Container parent) {
    Enumeration enum = components.elements();
    Dimension dim = new Dimension();
    while(enum.hasMoreElements()) {
        Component comp = (Component)enum.nextElement();
        dim.width = dim.width + comp.getMaximumSize().width;
        dim.height = dim.height + comp.getMaximumSize().height;
    }
    return dim;
}

public Dimension minimumLayoutSize(Container parent) {
    Enumeration enum = components.elements();
    Dimension dim = new Dimension();
    while(enum.hasMoreElements()) {
        Component comp = (Component)enum.nextElement();
        dim.width = dim.width + comp.getMinimumSize().width;
        dim.height = dim.height + comp.getMinimumSize().height;
    }
    return dim;
}

public Dimension preferredLayoutSize(Container parent) {
    Enumeration enum = components.elements();
    Dimension dim = new Dimension();
    while(enum.hasMoreElements()) {
        Component comp = (Component)enum.nextElement();
        dim.width = dim.width + comp.getPreferredSize().width;
        dim.height = dim.height + comp.getPreferredSize().height;
    }
    return dim;
}

public void removeLayoutComponent(Component comp) {
    components.remove(comp.getName());
}

public void setConstraints(Component comp,
    int leftPosition, int topPosition, int rightPosition, int bottomPosition) {
    GridConstraint cons = new GridConstraint(
        leftPosition, topPosition, rightPosition, bottomPosition);
```

```
        addLayoutComponent(comp, cons);
    }

    public void setConstraints(Component comp,
        int leftPosition, int topPosition, int rightPosition, int bottomPosition,
        int leftOffset, int topOffset, int rightOffset, int bottomOffset) {
        GridConstraint cons = new GridConstraint(
            leftPosition, topPosition, rightPosition, bottomPosition,
            leftOffset, topOffset, rightOffset, bottomOffset);
        addLayoutComponent(comp, cons);
    }
} // EOF
```

B.4 GridConstraint Source

```
public class GridConstraint {
    int leftPosition = 0;
    int rightPosition = 0;
    int topPosition = 100;
    int bottomPosition = 100;
    int leftOffset = 0;
    int rightOffset = 0;
    int topOffset = 0;
    int bottomOffset = 0;

    public GridConstraint() {
        // Do nothing.
    }

    public GridConstraint(
        int leftPosition, int topPosition, int rightPosition, int bottomPosition) {
        this.leftPosition = leftPosition;
        this.rightPosition = rightPosition;
        this.topPosition = topPosition;
        this.bottomPosition = bottomPosition;
    }

    public GridConstraint(
        int leftPosition, int topPosition, int rightPosition, int bottomPosition,
        int leftOffset, int topOffset, int rightOffset, int bottomOffset ) {
        this.leftPosition = leftPosition;
        this.rightPosition = rightPosition;
        this.topPosition = topPosition;
        this.bottomPosition = bottomPosition;
    }
}
```



```
        this.leftOffset = leftOffset;
        this.rightOffset = rightOffset;
        this.topOffset = topOffset;
        this.bottomOffset = bottomOffset;
    }
} // EOF
```

Bibliography

- [1] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison-Wesley, 1998.
- [2] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1998.
- [3] M. Bradac and Fletcher. A Pattern Language for Developing Form Style Windows. *Pattern Languages of Program Design 3*, 1998.
- [4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.
- [5] Luca Cardelli. Building User Interfaces by Direct Manipulation. *ACM SIGGRAPH Symposium on User Interface Software*, pages 152–166, 1988.
- [6] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.
- [7] Apple Computer. ResEdit Reference. <http://developer.apple.com/techpubs/mac/resedit/resedit-2.html>, 1999.
- [8] World Wide Web Consortium. Extensible Markup Language. <http://www.w3.org/TR/1998/REC-xml-19980210>, 1998.
- [9] World Wide Web Consortium. The User Interface Domain. <http://www.w3.org/UI/>, 1999.
- [10] World Wide Web Consortium. XML XLink Requirements. <http://www.w3.org/TR/1998/WD-xlink-19980303>, 1999.

- [11] World Wide Web Consortium. XML XPointer Requirements. <http://www.w3.org/TR/1998/WD-xptr-19980303>, 1999.
- [12] Dennis J. M. J. de Baar, James D. Foley, and Kevin E. Mullet. Coupling Application Design and User Interface Design. *Chi '92*, pages 259–266, May 1992.
- [13] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley, New York, 1993.
- [14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, New York, 1995.
- [15] Harmonia. User Interface Markup Language. <http://www.harmonia.com>, 1999.
- [16] Donald Hearn and M. Pauline Baker. *Computer Graphics*. Prentice Hall, New Jersey, 1994.
- [17] Christian Janssen, Anette Weisbecker, and Jurgen Ziegler. Generating User Interfaces from Data Models and Dialogue Net Specifications. *Interchi '93*, pages 418–423, April 1993.
- [18] Dan R. Olsen Jr. A Programming Language Basis for User Interface Management. *Chi '89*, pages 175–176, 1989.
- [19] Dan R. Olsen Jr. *Developing User Interfaces*. Morgan Kaufmann, San Francisco, 1998.
- [20] Dan R. Olsen Jr., Brett Ahlstrom, and Douglas Kohlert. Building Geometry-based Widgets by Example. *Chi '95 Mosaic of Creativity*, pages 35–42, May 1995.
- [21] International Business Machines. MVP: Model-View-Presenter. <http://www.ibm.com/java/education/mvp.html>, 1999.
- [22] Aaron Marcus. Graphical User Interfaces. *Handbook of Human-Computer Interaction*, pages 423–440, 1997.
- [23] Mathemaesthetics. Resorcerer. <http://www.mathemaesthetics.com>, 1999.

-
- [24] Microsoft. Visual C++. <http://msdn.microsoft.com/visualc/>, 1999.
 - [25] Mozilla. XUL Language Specification. <http://www.mozilla.org/xpfe/languageSpec.html>, 1999.
 - [26] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, pages 71–85, November 1990.
 - [27] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferreny, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering*, pages 347–365, June 1997.
 - [28] Brad A. Myers and Mary Beth Rosson. Survey on User Interface Programming. *Chi '92*, pages 195–202, May 1992.
 - [29] Gurminder Singh and Mark Green. Automatic Generation of Graphical User Interfaces. *Graphics Interface '86*, pages 71–76, 1986.
 - [30] Bluestone Software. XwingML. <http://www.bluestone.com/xml/XwingML/>, 1999.