Studiengang:	Informatik
Prüfer:	Prof. Dr. T. Ertl
Betreuer:	Dipl. Inf. Simon Stegmaier
begonnen am:	16. Mai 2002
beendet am:	22. November 2002
CR-Klassifikation:	H.5.2, I.3.6, C.2.4

Diplomarbeit Nr. 2017

A Semantic Description Language for Platform-Independent Graphical User Interfaces

Christoph Pfisterer

Institut für Visualisierung
und Interaktive Systeme
Universität Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Abstract

Graphical user interface toolkits exhibit striking similarities in functionality and visual appearance. Yet, each one demands adherence to its proprietary implementation interface from the application developer. Platform-independent solutions exist, but their perceived drawbacks hinder large-scale adaption.

This thesis provides a new perspective by complementing platform abstraction with other benefits found in some toolkits today. Network transparency and programming language abstraction are gained through the use of a remote protocol. An XML-based description language reduces coding effort and enables interactive dialog editors. The language is taken to a higher level by including elements to denote the behavior of the interface in addition to its structure.

Existing solutions are examined to provide an overview of useful features, but also to learn about shortcomings. Based on this examination, the architecture of the new system is then designed. The viability of the approach is demonstrated by a prototype implementation. It is based on the Qt toolkit and BEEP, a network protocol framework.

Deutsche Zusammenfassung

Verschiedene Toolkits zur Erstellung von grafischen Benutzungsoberflächen weisen erstaunliche visuelle Ähnlichkeiten auf. Dennoch hat jedes Toolkit eine eigene Programmierschnittstelle, die der Programmierer verwenden muss. Plattformunabhängige Lösungen wurden entwickelt, werden aber wegen befürchteter Nachteile nur schleppend angenommen.

Diese Arbeit zeigt mit einem integrierten Ansatz neue Perspektiven auf. Dazu wird Plattformunabhängigkeit mit anderen heute verfügbaren Techniken ergänzt. Ein generisches Kommunikationsprotokoll erlaubt nicht nur Netzwerktransparenz, sondern auch Unabhängigkeit von einer bestimmten Programmiersprache. Eine XML-basierte Beschreibungssprache reduziert den Kodierungsaufwand und erlaubt die Erstellung von interaktiven Dialog-Editoren. Die Sprache enthält auch semantische Elemente, die über die Struktur hinaus auch das Verhalten der Benutzungsoberfläche spezifizieren.

In der Arbeit werden zunächst existierende Lösungen untersucht. Dabei werden Hinweise auf nützliche Funktionen gesammelt, aber auch Nachteile herausgestellt. Der folgende Entwurf einer neuen Architektur basiert auf diesen Betrachtungen. Die erarbeiteten Konzepte werden anhand einer Beispielimplementierung demonstriert. Diese basiert auf dem Toolkit Qt sowie BEEP, einem Framework für Netzwerkprotokolle.

Contents

Chapter 1

Introduction

*"That's what would be so interesting, Pooh.
Not being quite sure until afterwards."*
Eeyore in *The House at Pooh Corner*

1.1 Motivation

Developing applications with a graphical user interface (GUI) today requires commitment to an application programming interface (API) with limited scope. Since an API requires that interface behavior be pinned down by large amounts of code, research efforts have gone into cross-platform programming interfaces that use a declarative textual description of the GUI. Runtime mapping or code generation are used to drive the native API, directed by the description. Unfortunately most attempts in this area are limited to a syntactical level and bound to a specific toolkit, so there is a need for further research in this area.

1.2 Goals

The goal of this thesis is to develop a semantic description language for graphical user interfaces. It should incorporate the benefits found in existing solutions, but avoid their shortcomings. The task is broken into several units:

1. Researching and evaluating existing approaches to describe graphical user interfaces using XML.
2. Defining a new platform-independent description language for graphical user interfaces. The language includes processing of user interaction on a semantic level. The language should be able to abstract from the used devices and their form of interaction in a sensible way. The results from unit 1 should be taken into account.
3. Developing a concept to connect the user interface and application-specific code across machine boundaries, using suitable communication protocols. The objective is to eliminate execution of application code on the client side.
4. Demonstrating the developed concepts with a prototype implementation using Qt and C++.

All parts of the thesis should aim at generality and independence of concrete programming languages and platforms. Established methods in this area, like the Model–View–Controller pattern, object-oriented component systems, and scripting architectures, should be considered and integrated.

1.3 Overview

The results of examining the state of the art in GUI programming and GUI description languages are presented in Chapter 2.

An outline of the solution is given in Chapter 3. It is followed by in-depth coverings of the various aspects, which are covered in turn on the design, description, and prototype implementation level.

Chapter 4 describes the abstract object model, used by both the application and the widget set to gain platform and language independence.

Chapter 5 presents the widget set and its general workings.

The next three chapters cover the semantic concepts devised to make widgets come to life. Chapter 6 talks about automatic data synchronization, Chapter 7 presents the event and action system, and Chapter 8 covers the special needs of top-level windows.

The protocol for remote communication is the topic of Chapter 9.

The system is evaluated in Chapter 10 by porting an example application.

Chapter 11 finishes the report with a conclusion. It collects the results and shows what further work is required to put the system to good use.

1.4 The Name of the Beast

Every project needs a name. The presented system was dubbed "SUIT", an acronym for "Semantic User Interface Thingy". The name is not only short, but also lends itself to the creation of icons. For example, the icon for a KDE implementation could show the KDE mascot Konqi wearing a suit.

Chapter 2

Existing Solutions

"Organized it. Which means – well, it's what you do to a Search, when you don't all look in the same place at once."
Rabbit in *The House at Pooh Corner*

2.1 The State of the Art

Before actual toolkits are examined, this section tries to give an overview of the state of the art in graphical user interfaces.

2.1.1 Traditional GUI Programming

Graphical user interfaces is an area of computer science that surprisingly has developed many common techniques, but no common standards. To the contrary—commercial PC operating system vendors use the look and feel of their products as a distinguishing attribute. It is exploited in marketing and protected as a trademark or even as "intellectual property".

In the workstation market and the free software community—on Unix based systems, that is—the X Window System [41, 42] ("X11" for short) sets a common standard for accessing graphics hardware. However, X11 only covers drawing commands, screen subdivision and input event propagation. That led to the creation of a wide variety of toolkits that are used to draw actual interface elements. While these can be used concurrently thanks to the common foundation, each of them implements its own look and feel, leading to large inconsistencies.

There is one thing that all toolkits have in common, though: each one has its own unique programming interface (API). To make a program available on multiple platforms, the part of the program communicating with the toolkit usually must be rewritten or adapted with significant effort.

2.1.2 Widgets

It is common practice to compose a GUI from specialized reusable components, so-called widgets or controls. Each widget occupies a discrete part of the screen. There are different kinds of widget, each with characteristic appearance and behavior. Thus widgets fit optimally into the world of object-oriented programming.

Widgets are usually arranged in hierarchies, forming visual groups, dialogs and applications. This hierarchy is analog to a spatial "contained-in" relation, and also proves useful for delegating control within the application. These so-called widget trees can easily be represented using a structured markup language like XML [4].

Since widgets are generic components, they must be customized for the application. This is done by setting variables, often called properties.

2.1.3 Description Languages

Writing code to instantiate all the widgets that make up a dialog can be tedious and redundant. To reduce that effort, many toolkits offer interactive dialog editors that save the composed widget tree into a file that can then be loaded by the application at run time. One can distinguish binary formats (e.g. Win32 resources, Mac OS resources, the Cocoa NIB format) and text-based structured description languages (e.g. Qt Designer, GLADE). The latter ones feature better (human) readability. They are often based on XML, which allows tools from other vendors to be used if required.

2.1.4 Platform Independence

Software vendors invest large amounts of resources into their software products. As a result, they prefer to sell their products to as wide a range of customers as possible. This range of customers is limited by the platforms a product runs on. The traditional way to make a program available on several platforms—rewriting the relevant sections to the platform-specific APIs—requires considerable time and effort, so it is not always economical.

Indeed, similarity between platforms can be striking. Figure 2.1 shows some basic widgets as rendered by different platforms and toolkits. From this results the desire to have one uniform API that can be used to cover several platforms at once. There are two fundamental approaches to create such an API.

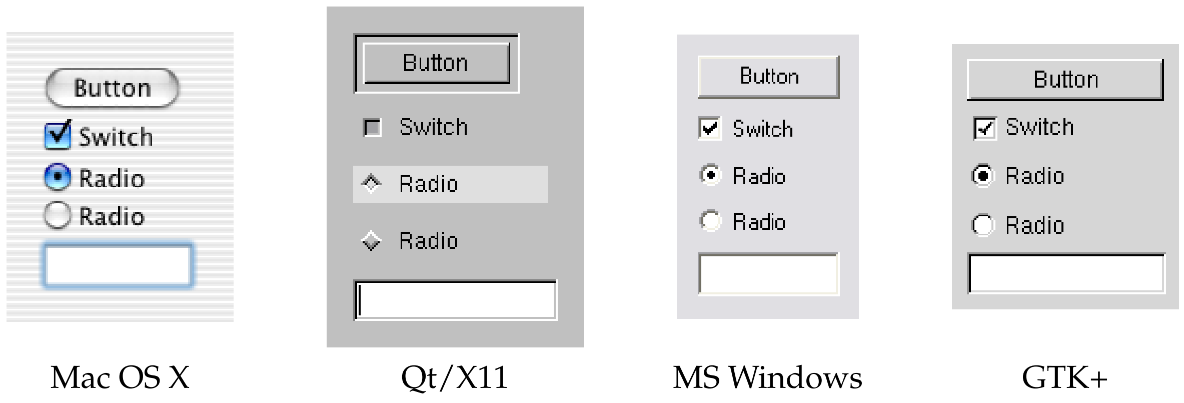


Figure 2.1: Different Platforms, Same Buttons

The first approach is porting an existing API to another platform. The interfaces of one platform are reimplemented, ultimately mapping them to the new platform's native interfaces. Functionality missing from the native interfaces must be emulated. This approach focuses on absolute compatibility; in many cases the look and feel of the original platform is preserved.

The other approach is creating a new, platform-independent API from scratch. This higher-level API is then implemented on each target platform using the respective native APIs. With this approach one can choose to use the native look and feel on each platform or to create a new look and feel to be used on all platforms. The latter implies that widgets are drawn by the implementation of the platform-independent API, circumventing the native toolkit.

All categories presented here are realized in both commercial and free products. There are some hybrid approaches, too. For example, Qt [30] uses native widgets on Win32, but implements its own look and feel on X11.

2.1.5 The World Wide Web

The World Wide Web has made SGML-style markup languages popular, but has also impacted user interface technologies. With browsers available on any platform, the HTML/HTTP combination has become an attractive alternative target "platform" for application user interfaces. Its use of client-side forms and page requests presents a radically different interaction metaphor. Considerable effort has gone into investigating web application design and architectures that unify both traditional GUIs and web interfaces.

2.1.6 Abstract Description Languages

Recently there has been some work on abstract, platform-independent GUI description languages. Some of these projects are driven by the advances of the World Wide Web and E-Commerce technologies. Their focus is on larger-scale abstraction to allow a unified treatment of different device classes and interaction styles, covering mobile devices and speech interfaces in addition to traditional GUIs. Other languages were created simply to implement an interactive dialog editor for a platform-independent toolkit.

2.2 Existing Description Languages

This section examines existing, XML-based description languages, based on the following criteria:

- Level of generality: What platforms, devices and toolkits are ultimately covered by the language? Does it use an existing platform-independent toolkit or does it try to map to various toolkits?
- Kind of generality: Can the language be used for different targets? Can a concrete description be reused for different targets? In other words, where does the mapping to the target platform occur?

- Relation to component models: Does the language promote the use of a component model for application code? Does it favor a specific component model or programming language? Or is the question of interfacing with the application left open?
- Semantic mechanisms: What mechanisms does the language provide to express interface behavior within the language instead of writing code?
- Remote access: Do the language and its environment allow remote display of applications? The network transparency built into X11 is disregarded for this purpose, since it is not an achievement of the toolkit in question.

Table 2.1 gives an overview of the results. It lacks data on ISL and Stöttner's UIML extensions as both lack actual definitions or implementations that could be assessed.

2.2.1 XUL

XUL [15] was developed and implemented as part of the Mozilla Project [26]. The goal was to ease the development of the GUI of this cross-platform web browser. The language unites XML tags for Mozilla's portable GUI components with HTML tags for layout purposes. XUL is used to describe dialogs and the widgets that make them up. JavaScript is used to describe behavior; it lets the developer manipulate widgets and access C++ components.

XUL is not intended for general applications. It is focused on the web browser suite it was designed for, its main purpose being customization and extension. As an example, Mozilla's chat client is implemented entirely in XUL and JavaScript.

Under the aspect of generality XUL comes off well. Both the language and concrete descriptions can be used without changes on any of the platforms supported by Mozilla. At the time of this writing those are MS Windows, Mac OS Classic, Mac OS X, BeOS, OS/2, OpenVMS, Linux, and a range of other Unix derivatives. Application code is expected to be written to XPCOM [40], Mozilla's own component

Point	XUL	UIML	ISL	Qt	GLADE
Desktop Platform Support	+	∘	−	+	∘
Non-Desktop Platform Support	−	+	−	∘	∘
Language Portability	+	+	+	+	+
Description Portability	+	−	+	+	+
Component Model	+	−	∘	+	−
Semantic Mechanisms	−	+	−	∘	∘
Remote Access	−	−	+	−	−

Legend: + present ∘ present, but limited − absent

Table 2.1: Results of the Examination

model. XUL does not contain special semantic constructs, but instead provides scripting hooks modeled after HTML. Network transparency is not provided out-of-the-box, but could likely be added using XPCOM.

2.2.2 UIML

The goal of UIML [1, 28] is the uniform description of user interfaces across a wide range of devices. In addition to PCs and workstations, this covers PDAs, mobile phones, and even speech interfaces (VoiceXML). Unfortunately, only the language itself is uniform, while actual descriptions are tailored to the target device.

UIML positions itself as a meta-language, similar to XML. As such UIML does not define UI components, but leaves that task to the developer of concrete mapping tools. UIML also lacks a clear statement on the communication of UI and application code. On the other hand, there are explicit elements to describe behavior as part of the interface description. Rules can be defined that cause certain actions when an event occurs.

UIML has gone through several years of development and refinement. The third version of the specification is currently under development. Various commercial tools using UIML are available.

In conclusion, it can be said that UIML's generality covers an exceptionally wide range of platforms, but lacks portability for concrete descriptions. Even porting between different PC platforms—which is a fairly small gap to cross—requires a part of the description to be rewritten. The specification ignores the issues of both component models and networking support. The language features support for semantic notation. The user can define complex events and handle the reaction to these within the UI description by manipulating widgets. However, there is no automatic synchronization with application variables.

2.2.3 Proposals to Extend UIML

In [35] Stöttner gives an overview of existing description languages and presents his own approach. It is based on UIML, but extends it to provide automatic property lists via "selectors" and reuse of description parts. The goal is to improve platform independence, i.e. to allow the same description to be used on different platforms.

At the time of publication there was neither a specification nor a prototype implementation. It could not be determined if the project is still active. As a result, it is impossible to provide a conclusive assessment of the approach.

2.2.4 IML

Göschka and Smeikal postulate a new paradigm for the abstract description of user interfaces in [11]. Their Interaction Markup Language (IML) is intended to be completely independent of the device used. The place of widgets as building blocks is taken by interactions. The authors cite E-Business applications as a possible target; these applications should be usable from arbitrary devices. Java Swing, HTML, and WML are mentioned as target technologies.

At the time of publication neither a language definition nor a prototype implementation was available. The authors recognize that automatically mapping interactions to actual GUIs is largely an unsolved problem. They seek solutions using artificial intelligence methods. It is not clear whether this sketchy approach can be successful. While the paper analyzes and underlines the shortcomings of traditional methods, it fails to point out the benefits of the new proposal.

It is impossible to provide a conclusive assessment of IML as neither a language definition nor an implementation are available.

2.2.5 ISL

The Interface Specification Language (ISL) [14] was defined by Hodes and Katz as part of the MASH project [22]. It describes the interfaces of “services”, which boil down to a generalization of remote objects. The language defines the methods exposed by a service and their parameters. This description is automatically mapped to a simple GUI. As it stands the language is not suited to create general GUIs. The primary field of application is remote control of simple devices or applications. The paper cites remote control of a room’s lighting as an example.

The focus of the MASH project [22] is on distributed collaboration and streaming multimedia. The implementation mentioned in [14] is still included in the current source code distribution of the MASH toolkit [23], but apparently hasn’t seen further development.

ISL has a high level of generality. Both the language and concrete descriptions are portable between the supported platforms. This portability is reached through a massive cutback in functionality, however. The idea of “services” on which ISL is based is not without similarities to object-oriented component models. While there are no semantic mechanisms, there is an RPC-like network interface.

2.2.6 Qt Designer

Qt [30] is a commercially developed platform-independent toolkit written in C++. Version 3.0 supports MS Windows, Mac OS X, and X11 on Unix as well as certain embedded systems. The MS Windows and Mac OS X versions use native widgets. The other versions use a custom look and feel and do their own drawing. Qt also provides non-graphical operating system abstraction, including file system and database access.

Qt extends the C++ programming language to provide its own runtime type information, communication mechanism ("signals" and "slots"), and component model ("properties", "object trees").

A dialog editor called Qt Designer is shipped with Qt. It allows the developer to compose dialogs interactively and to save them to an XML file format. The language primarily defines the widgets and their layout, but also provides for signal and slot connections. Thus some simple semantic aspects of the UI can be implemented without writing code. Since Qt itself is platform independent, so are the Qt Designer descriptions. However, the mechanism does not account for screen size differences, which can become a constraint when targeting embedded devices and desktop platforms simultaneously.

In conclusion, Qt provides good generality within the supported platforms. Both the language and the descriptions are interchangeable. The range of platforms covers most mainstream systems, but doesn't always provide optimal integration, e.g. by using a custom look and feel on X11. The specification language has rudimentary semantic concepts. Qt comes with its own component model that is a priori tied to C++. There are no explicit provisions for remote communication.

2.2.7 GLADE

GLADE [9] is a dialog editor for the GTK+ toolkit [12] and the GNOME desktop environment [10]. The interactive editor generates XML files and the C code to create widgets from these files.

The generality is limited by that of GTK+. GTK+ was originally created as an X11 toolkit, but has since been ported to Win32, BeOS, and frame-buffer devices (for integration into embedded systems) [12]. Most ports maintain GTK+'s specific look and feel, and as a result don't integrate well with existing environments. Both the language and the descriptions are usable on all supported platforms. There is no explicit reference to a component model. While GTK+ uses a class hierarchy, the native API is in plain C and uses non-OO techniques like callbacks with user data pointers. GLADE provides neither semantic mechanisms nor remote communication.

2.3 Platform-Independent Toolkits

To complement the survey, some platform-independent toolkits that don't provide explicit description languages are also briefly examined.

2.3.1 Java AWT

The Abstract Window Toolkit (AWT) [16] is the first attempt to provide a platform-independent GUI toolkit for the Java programming language. Today, the AWT has been obsoleted by Swing, described below.

AWT's approach is to map a set of abstract widgets to the corresponding native widgets of the current platform. This happens at run-time and is supported by the use of layout management objects instead of fixed size specifications. Unfortunately, the AWT suffers from a range of implementation problems.

2.3.2 Java Swing

Swing—or more precisely the Java Foundation Classes [18] with Swing as the main component—is the second attempt to provide a platform-independent GUI toolkit for Java. Swing builds on the AWT, but replaces AWT's native widgets with widgets that are drawn by Swing itself. Swing comes with its own look and feel, but also includes an architecture to customize the look and feel of the whole toolkit. This allows Swing to imitate the native look and feel of the current platform.

2.3.3 FOX Toolkit

The FOX toolkit [8] is a platform-independent toolkit written in C++. According to the developers the focus is on efficiency. FOX implements its own look and feel, independent of the target platform.

FOX contains some approaches to reduce the amount of code that must be written for an application. For instance, widgets can be connected through simple data pipes.

FOX is available as Open Source software. Currently, there are implementations for MS Windows and for X11 under Unix.

2.3.4 wxWindows

wxWindows [39] is a platform-independent toolkit written in C++. It uses native widgets where possible. The programming interface and its extension of the C++ language through macros are modeled after Microsoft's MFC library.

At the time of this writing there are implementations for MS Windows, GTK+ on Unix, Motif on Unix, Mac OS Classic and Mac OS X. The developers of wxWindows are also working on a dialog editor and an XML description language, but this work hasn't progressed enough to justify a closer investigation.

Chapter 3

The Idea

"What does Crustimoney Proseedcake mean? For I am a Bear of Very Little Brain, and long words Bother me."
"It means the Thing to Do."
Pooh and Owl in *Winnie-the-Pooh*

3.1 Introduction

The overall goal of this thesis is to create a framework to ease the development and deployment of graphical user interfaces for applications. Most of the specific goals are motivated by reducing the amount of code to be written: Platform independence removes the need to write several GUI implementations. Network capabilities make deployment more flexible without requiring additional code. Finally, a structured description language is used to specify the behavior of the interface in addition to its structure.

The following sections give an outline of the proposed architecture.

3.2 Decomposition of Application and Interface

The main point in the architecture is the decomposition of traditional monolithic GUI applications into two separate entities: the application core logic and the graphical user interface. The decomposition follows the spirit of the Model–View–Controller and Document–View design patterns. However, this separation is both more formal and more complete. The two parts communicate using a fixed generic protocol. While the protocol can be "implemented" using local function calls, this is no longer a requirement. The two parts may reside in different processes or even on different machines. This reduces dependencies and allows for different technologies to be used.

In itself, such a decomposition is nothing unusual—traditional client/server programming has been doing this for a long time. It is generality that sets the proposed architecture apart. The GUI side is not implemented using application-specific code, but using a textual description that is interpreted by generic code.

This second aspect creates a conflict. On the one hand, it is desirable to group all interface-specific functionality together, leaving just the passive application core on the other side of the wire. On the other hand, such a complete separation usually requires application-specific code in a Turing-complete programming language to be executed on the GUI side. Thus the challenge is to express as large a part of the interface behavior as possible using the textual description, maximizing expressible functionality while avoiding Turing completeness.

3.3 A Look at Model–View–Controller

To clarify the decomposition and its effects, one can relate the architecture of SUIT to the Model–View–Controller design pattern (MVC) [20].

The MVC pattern postulates that a GUI application has three distinct parts: The *model* keeps the application state and allows other parts of the application to discover and manipulate this state. The *view* is responsible for presenting the application state to the user in a suitable way. The *controller* is responsible for operating and interpreting user interaction, manipulating the model and the view as appropriate.

Note that there is some disagreement about the tasks of the view. In the SmallTalk world, the view talks directly to the model and handles any necessary mapping itself. In the NeXTStep/Cocoa world, however, the view usually only consists of generic objects (the widgets) and has no direct connection to the model. Instead, the controller is responsible for transferring data both ways and mapping it on the way. Both systems have their merits, but for this consideration, the classic system with three way communication will be used; it is shown in Figure 3.1.

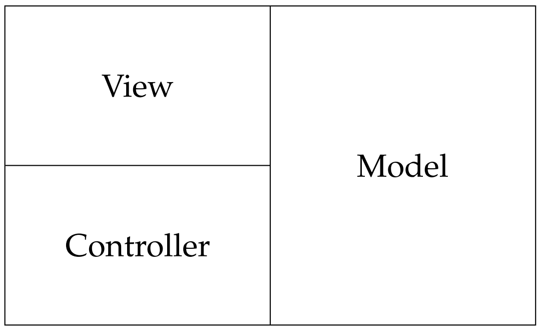


Figure 3.1: A Traditional MVC Application

The ideal decomposition as shown in Figure 3.2 breaks the application down into the model on one side and the view and controller, together implementing the user interface, on the other side. All user interface aspects are removed from the application core.

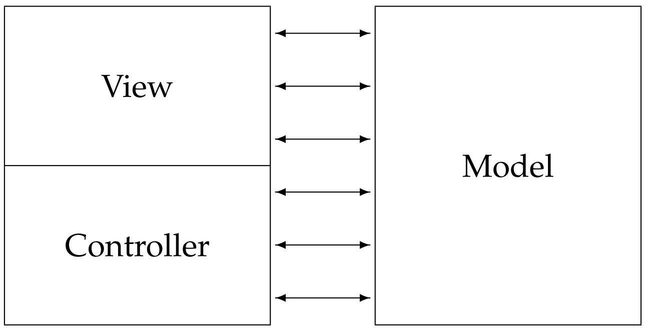


Figure 3.2: The Ideal Decomposition

Unfortunately, this clean decomposition is not possible if the GUI side uses just a declarative description language as proposed. There may be complex tasks in the view and controller domains that require the expressiveness of a proper programming language. Those tasks can only be implemented on the application side of the decomposition, as front-ends to the model. Still, in many cases the description language suffices and the GUI side can access the model directly. An integrated description language also blurs the destinction between view and controller, which Figure 3.3 indicates with a dashed line.

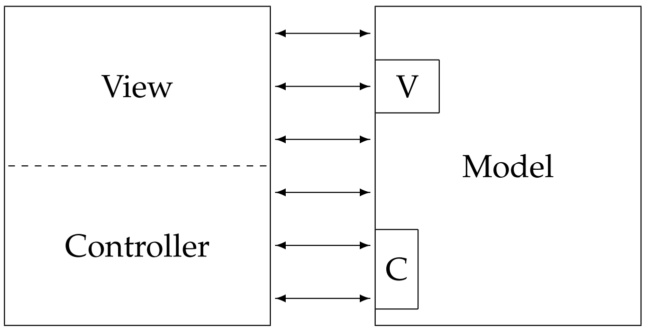


Figure 3.3: The Decomposition in Reality

3.4 Platform-Independent Toolkit

The architecture includes a platform-independent toolkit (or widget set) that operates on a high level of abstraction. This has several advantages. It is easier to map high-level specifications down to actual toolkits because there is more room to adapt to special requirements of the toolkit. The abstract toolkit handles low-level mechanisms like mouse button events internally, relieving the application of the burden of dispatching. This also simplifies the interface between the application and the toolkit.

The toolkit is mainly aimed at unifying the various PC and PC-like platforms in use today. However, it is hoped that the high level of abstraction provides the generality required to adapt the system to other classes of devices, in particular low-resolution devices like PDAs and 3D GUIs. In an ideal case, the same widget set could be used and existing interface descriptions could be adapted. In any case, the architecture allows for one application to be accessed through different interfaces, which can be tailored for different needs without changing the core application logic.

3.5 Textual Description With Semantics

Instead of providing a procedural API to the abstract toolkit, the architecture mandates the use of a structured description language based on XML to specify the interface. Together with the decomposition into two entities, this gives the actual implementations of the GUI part the margin to adapt to different platforms. Adding semantics to the description makes the interface more responsive since it reduces communication with the application core logic. It also allows the GUI side to take over the active role and manipulate the passive application core as requested by the user.

At the same time, a declarative notation allows higher-level concepts to be used. For example, one can specify that two variables are to be kept synchronized. The task of identifying all events that can change one of the variables and providing appropriate handling code is offloaded from the developer to a generic program.

XML was chosen because it has a number of advantages. It is human readable, editing tools and parsers are widely available. In addition, XML is based on Unicode, so it is a good base for internationalization and localization.

3.6 Object Model

The division into two parts makes it necessary to have a formal model of the application, or more precisely its data access and manipulation interface. Since the communication protocol eliminates language restrictions, it is natural to specify a high-level object model that maps to different object-oriented programming languages and component systems. Since component systems are already in wide use, this also promotes code reuse and lowers the migration barrier.

3.7 Networking Capabilities

The decomposition of application core and graphical user interface makes it possible to put the two pieces on different computers as long as they can communicate through a network. In a typical scenario, the application runs on a central server with local access to application data and computing power while the application's interface is displayed on a worker's PC on her desk.

This idea isn't new—X11 was designed to be network transparent right from the start. Unfortunately, X11 puts the network connection between the graphics engine and the toolkit. An X11 toolkit receives low-level mouse and keyboard events over the network and sends drawing commands back, introducing significant latency. The toolkit used on the application's host determines the look and feel, not the one on the displaying host. Finally, the idea of built-in network transparency never really caught on in the PC world. Instead, client/server computing and Java were designed and promoted to close this gap.

The proposed architecture uses two methods to reduce the effect of network latency. First, the native toolkit on the displaying machine is used. Input events are handled locally by the native toolkit and the abstract toolkit layer. For example, text editing in a text entry widget can happen without application involvement until the user leaves the widget. At that time, the entered text can be transmitted asynchronously and in one piece. Second, putting semantics into the declarative interface description can reduce communication further by handling some application-specific interactions on the displaying machine.

Chapter 4

The Object Model

"You know how it is in the Forest. One can't have anybody *coming into one's house. One has to be* careful.*"*
Rabbit in *Winnie-the-Pooh*

4.1 The Participants

The object model is the foundation for the integration of application code and user interface, crossing programming language borders as well as network connections. It defines the scheme used to describe the public interface of the application. Interfaces that are described using this uniform model can be accessed by both descriptive user interfaces and scripting systems.

4.1.1 Design Considerations

The design of the object model must take established component systems into account to ensure the best possible compatibility with those existing systems. It is crucial to minimize the effort required to port an existing application to the new model. Luckily, this isn't hard—object-oriented component systems tend to be very similar and compatible for exactly these reasons.

The decisions were guided by examining three existing systems: the JavaBeans component system [17, 13], the Qt object model [30, 31], and the abstract application model used by AppleScript [7]. JavaBeans was itself designed with interoperability and integration as the primary goal, taking existing enterprise-level architectures into account; it thus provides a good background.

4.1.2 Objects and Classes

An "application" is modeled as a set of "objects". Each object is an instance of a "class". All objects that belong to the same class have the same properties and behavior. One usually talks about the class as a whole, meaning all objects of that class.

Following standard object-oriented practice, a class can be declared to inherit from another class, meaning that all members (variables, methods, etc.) of the base class are implicitly made available in the subclass as well. If the subclass declares a member of the same name as the base class, the member of the subclass takes precedence. The process of subclassing can be repeated, creating an inheritance hierarchy.

A class can provide "properties" and "methods". Both are identified by a name that follows the usual rules for identifiers. It must be assumed that there is only one namespace for properties, methods, and the implicit access methods used to implement properties. Thus name conflicts between properties and methods must be avoided by the programmer.

It is assumed that the system provides runtime information about the available classes. This means that given an object, the system is able to name its class and provide lists of properties and methods. It turns out that this is not required per se for regular operation. However, the glue code used by a concrete implementation will likely rely on runtime information of some kind.

4.1.3 Properties

A property represents an instance variable. However, direct variable access is replaced with implicit access methods implemented by the class. That gives the class the flexibility to react to value changes and to store the properties in any way it prefers. In addition to these accessor functions, there is a notification system to alert other objects of value changes.

Properties are identified by a name. They have a type and can be either read-only or read-write. A means to statically express ranges of valid values for numeric types has been considered for this work, but has not been implemented. Validity can still be enforced by the accessor function if required.

While the details of property accessor functions are left to the actual component model and interface glue, the behavior of the "set" functions with respect to change notifications must be uniformly specified. Setting a property is an asynchronous operation, i.e. it does not return a value. It does cause a change notification to be sent, though. No change notification should be sent if the new value is identical to the old one. This reduces unnecessary traffic and helps to prevent loops. If validation fails and the new value is rejected, a change notification must be sent using the old (and still current) value. These conventions allow the application to converge onto a consistent state without sacrificing reaction time.

This model scales well from function calls to messages in a distributed system. To read a value, a read request is sent and answered with a value response. To change a value, a write request is sent. The write request is not answered directly, but can cause a change notification message to be sent back.

4.1.4 Methods

A method is a function associated with an object. It can be called from outside the object to initiate a certain action. Each method has a list of parameters, which may be empty. Methods do not return a value as they are often called asynchronously. However, a method can manipulate properties of the object it is called on, or initiate further actions, e.g. by calling other public methods.

Methods are identified by a name. The parameters are identified by their index and have a fixed type.

Advanced concepts like named parameters, default values, and overloading cannot be supported because they are not universally supported. For example, C++ supports overloading but not named parameters, Python does the opposite. These features can also involve significant overhead for dispatching when calling a method. If required, these concepts can be emulated within the limits of the object model. Instead of passing several parameters, a single List or Map value can be passed to a front-end method for further dispatching.

4.1.5 Concepts Not Used

Most component systems have an explicit “event” concept (“signals” in Qt). The object model proposed here does not, for a variety of reasons. An event is a mechanism for asynchronous notification where the sender of the notification doesn’t know the recipients in advance and the connection is established by the recipient or an outside entity. In a system with a descriptive semantic GUI there is no need for such a system transmitting from the application side to the GUI side. Changes in the application are instead propagated through property change notifications. Events that occur inside the interface (i.e. user interactions) are handled by the semantic description and translated into property changes and method calls. Events that need to be propagated inside the application code only are outside of the scope of the abstract object model—it is expected that an existing component system is used and then mapped to the abstract model at the interface to the GUI.

In the Qt object model, properties can have “reset” methods in addition to “get” and “set” methods. They can be used to set the property to a context sensitive default value. This extension of the property mechanism is uncommon and can be emulated by providing an additional property that makes the current default value available. Therefore reset methods are not included in the abstract object model.

4.2 Data Types

In addition to composite objects, the object model uses a range of value types. They are used for property values and method parameters. Most of them are predefined, but they can be combined to form complex data structures if required.

First of all, it is possible to leave the type of an entity unspecified. This can be represented as a special type, *Generic*.

A *Boolean* represents a boolean truth value; it is either *true* or *false*.

An *Integer* holds a 32-bit signed integer value. Implementations may use 64-bit signed integers instead, but must be prepared to shorten values to 32 bits before encoding and sending them to their protocol peer.

Strings are sequences of Unicode characters. Any implementation must be prepared to deal with arbitrary Unicode characters in the encoding defined by the protocol. Strings may be converted to other characters sets (e.g. ISO Latin 1), but this still requires proper handling of encoded Unicode characters (e.g. UTF-8).

There is a single *Float* type to represent non-integer real numbers. Transmission is done in a text format, so implementations are free to choose a storage format and precision. It is recommended to use at least “double” precision.

Enumerations are explicitly supported. Values are transmitted using their names as strings.

Arrays or lists of arbitrary length are represented using the type *List*. Lists are typed, i.e. they can be declared to contain only elements of a certain type. Untyped Lists are represented as Lists of Generics. Lists can be constructed of any type that would be a valid type for a single value, including Lists themselves. Note that actual component models have varying support for typed Lists. Automatic conversion between typed and untyped Lists may happen. Implementations that only support untyped Lists can simply ignore the issue.

A *Map* is a set of values indexed through string keys. This structure goes by many names in different programming languages: associative array, string map, hash, dictionary. Records can also be represented as Maps. Since Maps are often implemented using hash tables, no ordering information is preserved, just the primary mapping.

Finally, there is the *ObjectReference* type. It allows relations between objects to be expressed and to be changed dynamically. The storage is defined by the implementation, but will usually be some kind of pointer. The textual representation of object references varies with scope and will be explored in later chapters.

4.3 XML Encoding for Values

Both the description language and the remote protocol use XML and need a way to represent data values. This section describes the common encoding.

4.3.1 Simple Values

A *String* is written using the `string` tag, with the string as the content of the tag. Strings can use all Unicode characters representable in XML.

Integer and *Float* values are marked up in a similar fashion, using the standard decimal representation and the `int` and `float` tags, respectively.

Enumerations are encoded using the `enum` tag with the string name of the value as the content.

Booleans work the same, but using the `boolean` tag and the values `false` and `true`. (Note: An implementation may accept other values, but this should not be relied upon.)

Listing 4.1: Basic Value Encoding Examples

```
<string>Hello, World!</string>
<int>16</int>
<float>3.1415926</float>
<enum>ReadOnly</enum>
<boolean>false</boolean>
```

4.3.2 Lists

The `list` tag introduces a *List*. Explicitly typed lists give the type of the elements in the attribute `type`, using the tag name of the type. The content of the `list` tag is the required number of value tags. Any value tag may appear as a list element, including another list.

Listing 4.2: List Encoding Examples

```
<list type="int"><int>1</int><int>2</int><int>3</int></list>

<list>
  <int>42</int>
  <string>Don’t panic!</string>
  <list><boolean>true</boolean><int>4</int></list>
</list>
```

4.3.3 Maps

A `map` tag encloses the *Map*. The content is composed of key–value pairs. The `key` tag contains the key string and is immediately followed by an arbitrary value tag.

Listing 4.3: Map Encoding Example

```
<map>
  <key>column</key><int>3</int>
  <key>row</key><int>6</int>
  <key>label</key><string>Grand Total</string>
</map>
```

4.4 Extending Qt

This thesis includes a prototype implementation based on Qt [30]. Qt is a cross-platform application framework—in addition to GUI widgets it also provides application infrastructure and some operating system abstraction.

It turns out that this infrastructure already is very close to the abstract object model proposed above. There is a property system that supports almost all of the required data types. The signal-slot mechanism provides the base for property notifications as well as for generic method invocation. Qt also provides extensive runtime class information about these systems, generated by the meta-object compiler `moc`.

Still, Qt alone is not sufficient as a basis for the implementation. The prototype uses a custom version of Qt with two extensions: object references as a type and property change notifications. The following sections describe these modifications.

4.4.1 Object References

Qt uses the QVariant class to encapsulate values of arbitrary types. In this role QVariant corresponds to the Generic type in the abstract object model. Actually, QVariant covers not just the usual numeric, string, and collection types, but also graphics-related data like coordinates, icons or font specifications. This is not a problem as it can be ignored. There is no type, however, to store pointers to arbitrary objects.

Qt's property system is implemented in terms of QVariant and its supported types. Extending QVariant itself to support object pointers provides backward compatibility and avoids reimplementing the property system.

The change to QVariant also requires changing moc, since it needs to recognize the new type to generate correct dispatching code for property access. Since moc is unable to deal with QObject* as a type specification, QObjectPtr is defined as an alias using typedef. As a consequence, this type name must be used in Q_PROPERTY declarations and in the accessor methods.

To support the new ObjectPtr type, the modified version of QVariant contains additional methods, shown in Listing 4.4. Several generic methods also contain additional code.

Listing 4.4: QVariant Extensions

```
public:
  QVariant( QObjectPtr );
  QObjectPtr toObjectPtr() const;
  QObjectPtr& asObjectPtr();
```

4.4.2 Property Change Notifications

Qt's property system knows "get" and "set" methods, but does not provide a general change notification service. It appears that such notifications are left to the individual programmers, who can exploit the signal-slot mechanism for such notifications where needed.

The abstract object model postulates a general property change notification mechanism. Again, extending Qt's base classes is prefered over an overhaul of the property system. The modified version of Qt adds signals to QObject that are fired when a property changes, as well as a convenience method to be called by "set" methods. Listing 4.5 shows their signatures.

The reason for two signals is that some listeners will be interested in the new value of the property, while others will only care about the fact that there was a change. Providing the new value as a part of the signal's data saves an extra call in the former case.

Listing 4.5: QObject Extensions

```
signals:
  void propertyChange( QObject* obj,
         const char *propertyName );
  void propertyChangeValue( QObject* obj,
         const char *propertyName,
         const QVariant &newValue );
protected:
  void propertyNotify( const char *propertyName );
```

It should be noted that it is important to call `propertyNotify()` at the right time. Since the new value is transmitted to the listeners right away, it must be called after storing the new value in the instance variable. The processing done by listeners could take a while and may even cause further changes, so any locks held should be released before triggering the notification.

Finally, it is the task of each class to ensure that a notification is sent each and every time a property changes. This can be an issue for properties that are not stored, but calculated on the fly from other data. Programmers must also be careful when manipulating instance variables directly in other methods.

4.4.3 Generic Method Invocation

Generic method invocation means that given a method's name as a string, it is possible to run that method. C++ does not support this, and at first glance neither does Qt. However, Qt's runtime support for the signal-slot mechanism can be exploited to implement generic invocation. A closer examination of that mechanism is required before this becomes clear.

A Qt "signal" is similar to what other systems call "events". It allows asynchronous notification of arbitrary listeners. The Qt model calls these listeners "slots". What puts Qt apart is that both signals and slots are regular instance methods with arbitrary parameters (and no return value). A signal can be connected to any slot with a matching parameter list at runtime. Calling the signal method then results in all connected slot methods being called.

Qt achieves this by effectively extending the C++ language. Several keywords are added to class declarations that are hidden from the C++ compiler proper using preprocessor macros. The separate meta-object compiler `moc` interprets these keywords and generates not just meta-data about the class, but also the implementation for signal methods and a generic dispatcher method for the slots.

This slot dispatcher method is called `qt_invoke` and can be used to implement generic method invocation. Before it can be called, the name of the slot to call must be looked up in the class meta-data, and the parameter list must be converted to the expected format. The prototype implementation encapsulates this in a helper function called `invokeSlot`, to prevent exposure of Qt's private data structures. Should Qt's internals change in the future, it will be sufficient to change this one function.

Chapter 5

Widgets

"All right. We're going. Only Dont't Blame Me."
Eeyore in *Winnie-the-Pooh*

5.1 Widgets as Building Blocks

Widgets are the structural elements that make up a graphical user interface. They come in all shapes and sizes and support very different styles of user interaction. But still, there is enough common ground for an abstract model that fits all widgets. Of course, common behavior and features are associated with that model.

For abstraction, a widget can be seen as representing a certain, usually rectangular portion of screen space. This idea may be stretched somewhat by special widgets like menus or invisible grouping widgets, but the important point is that widgets can be put “inside” other widgets, forming the view hierarchy.

The tree of widgets formed this way is much more than a mere representation of spatial relations. It builds paths for data and control flow, separates unrelated parts and welds related ones together. There is a dependency relationship between a contained widget and its containing widget that has made it common to use the terms “parent” and “child” to refer to them. In short, this tree is *the* structure of a graphical user interface. Anything that happens in the user interface is associated with a specific location in the tree and can be said to happen “at” or “to” a widget. (The occasional global event is attributed to the Application widget at the root of the tree.)

The abstract widget model is in fact very similar to the abstract object model used for the application. Widgets are objects belonging to a certain class, and the classes are arranged in an inheritance hierarchy. Widgets also have properties just like application objects—identified by a name, with a fixed type and possibly read-only. What they don’t have is callable methods. Instead, events and actions (to be explored in Chapter 7) are used to manipulate widgets in ways that can’t be represented through properties.

To facilitate data access within the widget tree, widgets can be given a name. When qualified with the widget name, the properties of a widget become accessible as variables. In a traditional GUI toolkit it is sufficient for the application to store data to be displayed into widget properties. In the presented system, however, a description language is used, and accessing the application may cause significant delays. To compensate for this, the developer is given the possibility to create additional storage variables within the widget tree.

These free variables are identified by a name and have a fixed type. They do not become members of a widget like properties, but are registered in the same namespace as widget names. This allows for a greater deal of flexibility. When accessing a free variable, one doesn’t have to care about where in the tree it has been defined as long as it’s visible from the current scope. As opposed to widget names, variables are only visible in the widget where they are defined and its children.

5.2 The View Hierarchy

The section centers around the view hierarchy and its specific issues, namely automatic layout.

5.2.1 The Layout Problem

Designing a user interface for one specific platform is simpler than for a platform-independent toolkit. Widget dimensions and font sizes are fixed and known. The positions and sizes of widgets can be specified directly in pixels. This not only saves overhead, but also gives the interface designer freedom to create a layout that is not just well-structured, but also visually pleasing.

Resizeable windows are the first challenge to these fixed pixel-based layouts. A common solution is tying widgets to certain borders of their containers. If the container is resized, the toolkit automatically moves and resizes the widgets keeping the distance to the borders they are tied to constant. This approach is used in the Motif [27] and Cocoa [5] toolkits.

But the real problem arises if one moves to platform-independent toolkits, more specifically those using each platform's native widgets. Now font sizes differ, as do widget dimensions, margins between and around widgets, and to some degree even the design guidelines. The customary approach is to provide dynamic layout components; this can be observed in Java AWT and Swing [16, 18], GTK+ [12], and Qt [30]. The layout components adapt to the widgets they manage as well as the outside circumstances, e.g. the window size controlled by the user. The actual layout is based on relative arrangement specifications, sometimes complemented by relative measures, e.g. weights.

Since this work aims at platform independence, it uses dynamic layout components modeled after the major toolkits referenced above.

5.2.2 Placing Layout Functionality

Since layout management for child widgets is mostly orthogonal to the multitude of functional container widgets, separating the two is desirable to avoid a great deal of redundancy. There are two common architectures to facilitate this separation. One is to use layout manager objects that live outside the widget hierarchy and mediate between the container and its child widgets; Qt, Java AWT and Java Swing take this approach. The second approach is to make layout managers full scale widgets and allow at most one child widget for regular containers; GTK+ takes this approach.

For this work the second approach was chosen because it has a number of advantages. If layout managers are widgets, there is only one fundamental kind of object; both the textual description and the tree connection can be handled uniformly. In any situation, only two objects communicate (the parent and its children) instead of up to three (the parent, its children, and the layout manager(s) connecting both). If there is only one child, layout managers can be omitted altogether. Making layout managers full widgets also aids reusability on a larger scale: an arrangement of widgets (e.g. the contents of a dialog box) can be reused in other environments (e.g. a floating palette window) simply by moving the top-level layout widget to another container. If the layout is tied to the specific container (the dialog window in this case), this is much harder.

5.2.3 Widget Design Considerations

The most important part of layout management is determining the size of the widgets. Once a layout container has determined its own size and that of its children, actually arranging them—usually in some kind of grid—is relatively straightforward.

How the ideal size of a widget is determined varies with its class. Many simple widgets like text labels or push buttons are able to determine their ideal size based on their content, without the need for further information. Other widgets are expandable and thus ready to use any amount of space that is allocated to them. These widgets require the user to specify constraints in the form of a minimum and an ideal (or "preferred") size. Some widgets even behave differently in horizontal and vertical directions.

In short, while an implementation must find a way to generalize this multitude of individual requirements, the specification is better off using hints tailored to each widget class. In order to provide common terms and some background for talking about layout and geometry management, the next section presents an outline of a possible layout architecture. Of course, actual implementations will have to consider the underlying toolkit and its provisions in this regard.

5.2.4 A Possible Architecture

This architecture is based on some general requirements and definitions:

- Each widget provides its parent with a minimum size, a preferred size, and two flags telling whether it can expand (one for each dimension). The parent widget is responsible for determining the actual size.
- Each widget must be prepared to be assigned more space than its minimum or preferred sizes. In other words, it must be able to align itself in a bigger space, unless it expands to fill all available space.
- Vertical alignment is automatic for all widgets that contain text; they align on the baseline of the first line of text.

- Horizontal alignment defaults to left aligning the widget border. Widgets that cannot expand may have a property to specify centered or right-flush alignment instead. Widgets have no internal horizontal margins.

The layout mechanism accumulates minimum and preferred sizes bottom-up in the widget tree and then propagates actual sizes (partially determined by the user) top-down to the individual widgets. The only problem in this process is met by layout widgets with more than one child widget. They must decide how to distribute any space left after all child widgets have been allocated their minimum size. This is non-trivial if there are two or more expandable child widgets. One possible and reasonably simple strategy is the following:

- If the available space is between the total minimum and preferred size, each widget is assigned space proportionally to its difference between minimum and preferred size.
- If the available space is larger than the total preferred size, space is distributed equally between expandable widgets.
- If no child widgets are marked as expandable, left-over space is not assigned to the widgets. Instead, the default alignment rules are used to align the block of widgets inside the available space.

Note that all calculations and decisions are done independently for horizontal and vertical coordinates. A widget's size specification may change as its content changes. Implementations must be able to propagate and handle such changes.

Layout management is also responsible for placing margins and borders. The main problem here is assuring that there is enough blank space around simple widgets like buttons, but no margins between container widgets, for instance between a scroller and the containing window. Platform-specific features like resize corners and 3D effect borders must be taken into account as special cases.

Automatically determining margins and borders may require layout containers to examine their "environment" in the widget tree closely and across several levels, especially when layout containers are nested within each other. A general model for a widget's requirements could include a "hard border" vs. "soft border" flag as well as minimum inner margins (for containers) and outer margins (for all widgets).

5.3 The Class Hierarchy

This section talks about the class hierarchy and its implications. It also defines the widget classes of the abstract widget set, arranged in functional groups.

5.3.1 Of Interfaces and Implementations

Many widget classes have some common characteristics, so it is only natural to use inheritance to enable code reuse. Unfortunately, different toolkits have significantly different inheritance hierarchies. Figure 5.1 demonstrates this using the inheritance path of the check box widgets in Qt [30], Java Swing [18] and Cocoa [5].

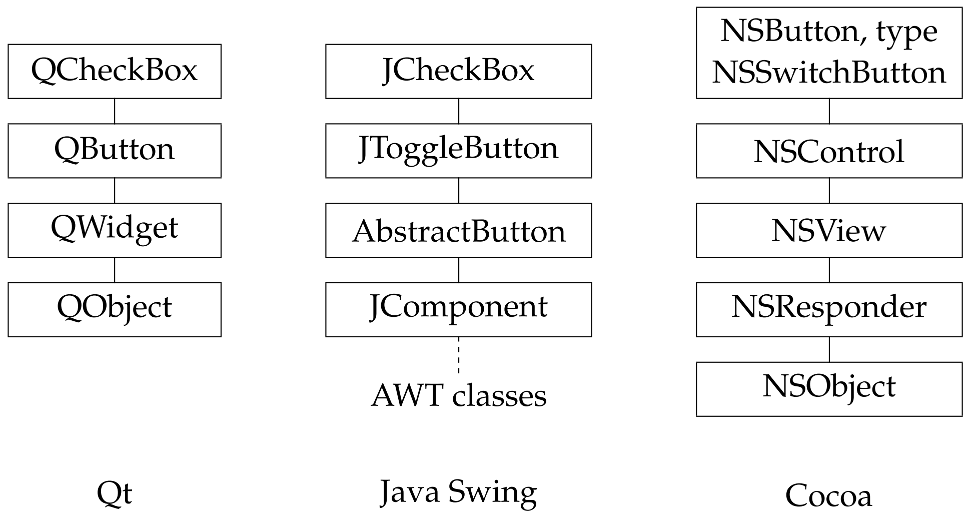


Figure 5.1: Inheritance in Different Toolkits

How does this matter to the application programmer? In the traditional model, the application code manipulates the widgets using the toolkit API. For instance, it calls a method to change the title of a window. Under these circumstances, a distinguished class hierarchy can be advantageous: the same special method is available for document windows, dialogs and palette windows alike.

The picture changes when a description language comes into play and the application code no longer accesses the widgets directly. There is no procedural API that would benefit from inheritance. All property accesses are done using generic calls. It becomes more important *which* properties a widget class supports than *how* it supports them.

As a consequence the abstract widget model does not concern itself (or the application developer) with widget class hierarchies. Instead, it specifies functionality common to all widgets and several rough groups of widget classes with common conventions. This gives actual implementations the flexibility to arrange widget classes into a hierarchy that mirrors that of the native toolkit or that allows for optimum code reuse.

5.3.2 Universal Widget Properties

The following describes properties that are available for all widgets. They are implemented in the common widget base class.

The first group of properties makes the structure of the widget tree available for reading. The `name` property gives the optional widget name assigned by the description. The properties `parent` and `children` give object references to the widget's container widget and to its contained widgets, if any. It should be noted that `parent` is a single value while `children` is a list. These properties are populated automatically from the interface description and cannot be changed programmatically.

Next is the `disabled` flag. It can be used to put any widget into a disabled, non-reacting state. Such widgets usually appear lighter (or "grayed out") to set them visually apart. If a container widget is disabled, its children are implicitly affected as well.

Finally, there is the `layoutInfo` property. It can contain a map of hints for layout containers, e.g. the column and row numbers for a grid layout. The actual keys and their interpretation are defined by the layout container in question. This property should not be changed at runtime as allowing for that would make the implementation of useful automatic layout containers considerably harder.

5.3.3 Basic Widgets

Basic widgets do not have children; they are leaves in the widget tree. They provide the actual functionality of the user interface—as opposed to containers, which are used for organization.

One can further distinguish simple and complex widgets. Most simple widgets display a short piece of text as a label. For uniformity, this property is always named `text`. Complex widgets distinguish themselves by having a non-trivial data model or by requiring several widgets to work together to implement what the user experiences as one interface element. There actually is a smooth transition between simple and complex widgets.

The following list shows possible widget classes, each with a short explanation. Following the goal of this work, they try to be platform independent.

- `TextLabel` is a static text label displaying some text that is not expected to change.
- `TextDisplay` is a text display widget intended for changing text. It can be set to display one line only or to display multiple lines with automatic word wrapping. In the single-line mode the text is shortened to fit the available space as necessary.

- `TextEntry` implements a single-line edit field for entering text.
- `PushButton` is a framed button with some text inside. It can be clicked by the user to cause a certain action.
- `CheckBoxButton` shows a check box with a text label on the side. The check box can be toggled by the user between "on" and "off" states and optionally a third, "undefined" (or "no change") state.
- `RadioButton` and `RadioButtonGroup` implement groups of radio buttons. Radio buttons are very similar to check boxes, but have a different look and different selection behavior. At most one button in a group can be in the "on" state at any one time. This is coordinated by the invisible `RadioButtonGroup` widget.
- `ProgressBar` is a widget that displays the progress of a lengthy operation as an advancing bar in a frame. It also supports an animated "indefinite" mode for operations where the amount of work isn't known in advance.
- The classes `IntValuePopup`, `FloatValuePopup`, `StringValuePopup` and `ChoicePopup` all implement a popup choice menu, but use different data models. `ChoicePopup` is the most generic one, but hard to map to application data structures. The other classes are provided to ease this task.
- `IntValueComboBox`, `FloatValueComboBox` and `StringValueComboBox` combine a pre-defined popup choice menu with a text edit field for entering values that are not present in the menu.
- `Table` and `TableColumn` together implement a list of items shown as a scrollable table with multiple columns. `Table` keeps the data and takes care of geometry management, while several `TableColumn` widgets model the actual columns. `TableColumn` provides for display of a piece of text taken from each row's data item, including optional inline editing. Further columns classes could be added for more elaborate cells.

Due to time constraints, the radio button, progress bar, popup, and combo box widgets are not implemented in the prototype.

5.3.4 Functional Containers

The common trait of functional containers is that they can have at most one child widget. If required, this single child can be a specialized layout container. What follows is again a list that is the result of evaluating various existing toolkits for useful abstract widgets.

- A `GroupBox` draws a box with an optional caption around its single child widget.
- `VerticalScroller` adds a vertical scroll bar to its child widget. Horizontal geometry management is handled normally, but vertical geometry management is decoupled through the use of the scroll bar, meaning that the child widget will always get its preferred size.
- `Scroller` implements a generic scroller, i.e. it adds scroll bars for both horizontal and vertical scrolling to its child widget. As a result, geometry management is fully decoupled.
- `TabContainer` and `TabPage` together implement tabbed views. The container can have any number of pages as its children. The container manages geometry as if all pages were displayed at once in the same space. Only one of the pages is shown at any one time. The container displays a tab for each page that can be clicked by the user to switch to that page. The container has no further properties; the pages have a property to specify their tab caption.
- `Splitter` is an exception in this group as it has two primary child widgets. It arranges the two horizontally or vertically and provides a dividing line that can be dragged by the user to allocate space. Each widget can optionally be collapsed completely.
- `InvisibleContainer` has the task of hiding or showing its contents based on a property flag. Hiding not just makes the contained widget invisible, but removes it from geometry management, so that the space can be used by other widgets.

Of these widget classes, only `TabContainer` and `TabPage` are actually implemented in the prototype.

5.3.5 Layout Containers

Layout containers mediate between functional containers and basic widgets by handling geometry management. For a working user interface it seems sufficient to have a grid-based layout style. This is implemented by the `GridLayout` class. It accesses the `layoutInfo` property of each child to determine its placement in the grid.

For situations where there is only one row or column in the grid, special classes are provided for convenience. They are called `VertBoxLayout` and `HorizBoyLayout`. They don't access the `layoutInfo` property but simply arrange the child widgets in the order in which they are defined.

5.3.6 Application-Level Classes

The widgets to make up the interior of windows and dialogs are now complete. What remains to be covered is the outside, namely top-level windows, menus, and other issues that affect the application as a whole. These three are closely related and must be dealt with as a unit, since this is a point where platforms differ significantly. For example, a Mac OS X application has a single menu bar along the top of the screen and doesn't need to quit when the last open window is closed. In the MS Windows and X11 worlds, menu bars usually reside inside individual windows and there are several ways to deal with multiple documents within one application.

All of these styles have in common that there is one window per document. This even holds true for the MDI style, where these windows are nested inside an application window. The functional difference is that some styles allow the application to have no open windows (this is the case on Mac OS X and with the MDI style), while other styles require at least one window to be open at all times, or taken another way, the application terminates as soon as the last window is closed. Many platforms leave the choice between the two general models to the application developer, and so does the system designed in this thesis. On platforms where only one of the abstract models can be implemented, the other one must be gracefully mapped by the implementation.

Regardless of the window model chosen, the interface can be described by three classes: `Application` represents the application as a whole, individual windows are represented by `Window` and `Dialog`. A menu bar can be associated with either the `Application` widget or with individual `Window` widgets. This can be mapped to both top-of-screen and in-window menu bars, as required by the respective platform style guide. The separate `Dialog` class exists to distinguish windows that should not get a menu bar. Apart from that, there is no difference between `Window` and `Dialog`; both also belong to the group of functional containers. Both the `Application` class and the window classes have a `title` property.

The menus are modeled using four classes. `MenuBar` is at the top of the tree. It represents a horizontal bar of pull-down menus. Its children must be of class `Menu`, which represents one such pull-down menu with a list of entries. Menu entries are usually of class `MenuItem` (a selectable item with a title) or of class `MenuSeparator` (a grouping separator), but they can also be of class `Menu`, generating a hierarchical submenu.

5.4 Describing Widgets

After a short introduction to XML terms, this section develops the description of widgets and their properties in XML.

5.4.1 Coming to Terms With XML

This section gives a short overview over common XML terms, in the hope that it will ease understanding of the following discussion.

An XML document consists of “elements”, also called “tags”. Tags have a name and may have named “attributes”. Tags can contain a mixture of text and more tags. Whitespace between tags is commonly ignored by programs and can be used to indent lines according to the structure.

There are various means to automatically validate XML documents. The one anchored directly in the language [4] is called Document Type Definition, DTD for short. A DTD specifies the elements that may occur in the document, their attributes, and their nesting.

Many different XML parsers are available as libraries, but most follow one of the two common parsing interfaces: DOM and SAX. DOM is the Document Object Model standardized by W3C [21]. A DOM parser creates a tree of internal objects representing the document and makes these objects available to the application using a standard interface. The DOM standard also specifies interfaces for run-time manipulation of tree objects.

SAX stands for Simple API for XML [34] and is a parsing model striving for simplicity and efficiency. Instead of creating representation objects, a SAX parser simply calls application callbacks as it parses the document.

5.4.2 The Structure

The abstract widget model makes it easy to describe widgets in XML, since it is made up of three generic entities. Widgets themselves, properties, and variables are separated by their relations: a widget has several properties and may have several variables attached.

These relations are most easily expressed if each entity is mapped to one XML tag. While it may be possible to map properties to element attributes, this has several disadvantages. Generality is defeated to some extent and it is no longer possible to write a DTD for validation. Such an encoding would also make it harder to come up with a comprehensive scheme for property values and synchronizations.

The role of widgets as the basic structural elements that make up the view hierarchy can be mirrored directly in XML by nesting the tags that describe widgets within each other. A DOM parser can transfer this nesting into a tree of nodes in memory, which can then be translated into implementation objects, keeping the same tree structure. (Of course, any implementation is free to use another parsing model instead.)

Following standard practice, the actual tags were named after the entity they describe: `widget`, `property`, and `variable`. They are described in more detail in the following sections, together with the `widgetref` tag, which extends the list of value tags to allow references to widgets. A comprehensive example is provided at the end.

5.4.3 The widget Tag

With structural relations off-loaded to tag nesting, what is left that must be included in the `widget` tag? At the very least, the class of the widget must be known so it can be instantiated, using default values if necessary. So, the class of a widget is specified in the required attribute `class`.

The widget name has also been mentioned already. It could be treated as just another property. Actually, it is available as a read-only property for any widget. But the name of the widget has a special role. It is used throughout the system to refer to the widget. To fulfill that task, it must be unique and must not change over the lifetime of a widget. So instead of treating it as a regular attribute, it is specified directly in the `widget` tag, using the `id` attribute. This happens to match the current (D)HTML standards and makes it possible to validate name references using standard XML tools. There are also implementation benefits. The widget name is known immediately at the creation time of the widget. If it was a just another property, the implementation would have to ensure correct initialization order with respect to widget references. Property initialization also assumes that a property is writable, while the widget name must not change over time.

5.4.4 The property Tag

What properties are available for a certain widget is determined by the widget class, not by the user interface description. The description controls their values, though. Property values are used to customize widgets and carry data, both of which fall into the realm of the user interface description.

It should be noted that any property has a default value. Specifying a value in the description is optional. This allows the description to be more compact in some places and also allows new properties to be added later without breaking existing descriptions. Description writers should take readability into account, however. Especially when it comes to widgets with several alternative modes, it can be useful to explicitly specify the mode nonetheless.

A property's value is set using the `property` tag. The tag must be a child of the `widget` tag it should apply to. Setting a property requires two pieces of information (besides the widget association expressed through nesting, that is): the name of the property and some kind of value specification.

The name of the property is conveniently specified using the attribute `name`. For the value tags were chosen over attributes because they provide greater flexibility, especially for complex values like lists. As will be seen in later chapters, this also provides a starting point for higher-level features.

Normal data values are encoded using the common XML encoding described in section 4.3 on page 35. This encoding uses the tags `string`, `int`, `float`, `enum`, `boolean`, `list`, `map`, and `key`. In addition, properties can be initialized with an object reference to another widget. This uses the `widgetref` tag described in the next section.

5.4.5 The widgetref Tag

The `widgetref` tag is an additional value tag. In the interface description, it can be used anywhere where standard value tags, e.g. `string`, would be allowed. It creates a value of the ObjectReference type, pointing to a certain widget in the description. This can be useful for instance to create connections within the widget tree that are beyond the pure tree structure.

The target widget is identified by its widget name, given in the `ref` attribute.

Listing 5.1: Widget Name Referencing Example

```
<widgetref ref="mainmenu" />
...
<widget class="MenuBar" id="mainmenu"> ... </widget>
```

5.4.6 The variable Tag

Properties and variables are very similar in nature. They store a value of a certain type, have a name, and live in the context of a widget. Thus their XML representations are very similar. The main difference is that variables only exist when explicitly declared, while properties exist of their own accord since they are defined by the widget class. This means that the description must also give the data type of the variable, since it is not known in advance.

Variables are declared using the `variable` tag. There are two required attributes: `name` giving the name of the variable, and `type` giving the data type. The initial value of the variable is given as the content of the tag. The same rules as for the `property` tag apply here.

It should be noted that the `variable` tag is not the only way to create free variables. The system automatically adds a variable named `app` at the root of the widget tree and fills it with an object reference to the root object of the application. Cloning, a semantic concept described in section 8.2 on page 76, also implicitly creates variables.

5.4.7 Extensive Example

To illustrate the tags introduced so far and their relations, here is a more comprehensive example:

Listing 5.2: Description Structure Example

```
<?xml version="1.0"?>
<widget class="Application">
 <property name="title"><string>TestApplication</string></property>
 <property name="mode"><enum>MDI</enum></property>
 <property name="menu"><widgetref ref="mainmenu" /></property>
 <variable name="status" type="string"><string>Ready.</string>
  </variable>
 <widget class="Window" id="about_window">
  <property name="title"><string>About TestApplication</string>
   </property>
  <widget class="HorizLayout">
   <widget class="TextLabel">
    <property name="text"><string>UI Version:</string></property>
   </widget>
   <widget class="TextLabel">
    <property name="text"><string>1.0</string></property>
   </widget>
  </widget>
 </widget>
 <widget class="MenuBar" id="mainmenu">
  <widget class="Menu">
   <property name="text"><string>File</string></property>
   <widget class="MenuItem">
    <property name="text"><string>New</string></property>
   </widget>
  </widget>
 </widget>
</widget>
```

This example is far from complete as it lacks any dynamic elements. It is meant to show how `widget` tags using various classes play together to form the structure of an interface description. Static property initialization is also shown.

5.5 Implementing Widgets

This section talks about the implementation of basic widget functions in the prototype. The widget tree is actually the focal point of the whole architecture. Specialized parts are described in later chapters, after the concepts are presented.

5.5.1 The Class Hierarchy and Qt Peers

Widgets are the largest of the three class hierarchies in the prototype implementation. For convenience, the classes are named after the class names in the abstract widget set description, e.g. `PushButton`. The base class is called `Widget` and contains most of the code to implement the generic functionality.

Directly inheriting the Qt widget classes would create problems, among them conflicts between the properties defined by the abstract widget set and the properties used by Qt's widgets. Instead, each `Widget` subclass creates Qt widgets as necessary and stores the pointers. Since there are abstract widget classes that map to zero, one, or several Qt widgets, support for this peer system is not provided in the base class `Widget`, but in an intermediate class called `RealWidget`. Simple widget classes like `PushButton` inherit directly from `RealWidget`.

Other intermediate classes are provided for functional containers (`ViewWidget`, a subclass of `RealWidget`), for layout containers (`LayoutWidget`), and for menus (`MenuBase`). The latter two are needed because Qt implements layout management and menus not in terms of `QWidget` subclasses, but with separate class hierarchies that spread from `QLayout` and `QMenuData`.

5.5.2 Parsing the Description

The code to parse the XML description is spread over several classes. An object of class `UIDescription` represents the XML description as a whole and coordinates the construction of live widgets. It initiates the process by parsing the XML into a DOM tree, using the XML support present in Qt. `UIDescription` then traverses the tree of `widget` tags and instantiates the requested widget classes.

Each widget instance parses the contents of the `widget` tag on its own. This happens in the `constructFromTag` and `parseActionList` methods of the `Widget` class, which are called by the constructor. Parsing of value tags is further delegated to the `Expression` class. There it also happens in methods called from the constructor. The result of this immedate processing is that the DOM tree can be de-allocated as soon as this first iteration of the tree is complete. Afterwards, the widget tree is represented by `Widget` subclasses and dependent objects, and can be traversed in the usual fashion, exploiting polymorphism.

Such a tree traversal actually happens several times before the widgets finally become active. This is because property initialization can involve cross-references between widgets that cannot be resolved at creation time. Likewise, the instantiation of Qt widgets relies on the completion of property initialization.

5.5.3 Qt Widget Creation

Mapping the abstract widget set to Qt widgets provides some challenges. While the abstract widget set treats every user interface element as a widget whose class derives from the `Widget` base class, Qt has specialized class hierarchies for some element groups. (The term "Qt widgets" is used nevertheless for the sake of simplicity.) This diversity requires a flexible infrastructure for the creation of Qt widgets. Since property initialization must happen before Qt widgets can be instantiated, the implementation can work on a fully initialized tree of `Widget` subclasses.

The base class implements these three methods to facilitate Qt widget creation:

Listing 5.3: QWidget Creation Methods

```
public:
  void createAll();
protected:
  virtual void create();
  virtual void postCreate();
```

The `createAll` method handles tree iteration and cannot be overwritten. Instead, it automatically calls `create` and `postCreate`, which are intended as hooks for subclasses. `create` is called in pre-order and is the normal place to create Qt widgets. This is because children usually need a pointer to the parent's `QWidget` for construction. If a subclass needs to do something to or with its children after creation, it can re-implement `postCreate`, which is called in post-order.

This mechanism is extended by `RealWidget` to further simplify the life of simple subclasses. Its implementation of `create()` obtains a `QWidget` pointer from the parent `Widget` and passes it to the new method `create(QWidget *parentWidget)`. A subclass only needs to re-implement this new method. `RealWidget` also handles other routine task in its version of `create()`, so subclasses should only re-implement it if there is a good reason to.

Chapter 6

Synchronizing Data

"I think Piglet thought of something at the same time. Suddenly."
Pooh in *Winnie-the-Pooh*

6.1 Expressions

Expressions are the foundation for all data access within the interface description. They provide addressing as well as simple calculations.

6.1.1 The Purpose

One purpose of expressions is addressing data. This is a broad task and includes array indexing as well as property access on widgets and application objects. Both of these are real requirements. While properties have a name, they are only accessible in association with their widget or object, which makes it necessary to have a notation that specifies both. Selecting one entity from a list for a detail view is a very common operation in a graphical user interface, and it requires array indexing with a variable.

The second purpose for expressions is less strongly required, but still useful. The "view" part of an application is often responsible for preparing data for presentation. Providing numerical expressions that are evaluated on the GUI side can ease this task for the application. For example, an application may keep some temperature in degrees Centigrade. If the user prefers a readout in degrees Fahrenheit, an expression can convert the value on the fly, without additional code.

6.1.2 Expression Elements

Expressions are composed of atoms and operators. Expression atoms stand for themselves as complete expressions, while operators combine one or two existing expressions to form a new one.

Element	Description	Example
Identifier	References a named entity. Consists of upper case letters, lower case letters, digits, and the underscore, but must not start with a digit.	`app`
Boolean constant	The special identifiers `false` and `true` represent the corresponding boolean values.	`true`
Integer constant	Consists of digits.	`42`
Floating point constant	Consists of at least one digit, a dot, and any number of digits after that. Starting a floating point constant with a dot is not allowed.	`3.14159`
String constant	May contain any character and is delimited by either single quotes (`'`) or double quotes (`"`). The quoting character used must not occur inside the string.	`"Hello"`

Table 6.1: Expression Atoms

Op.	Scheme	Description	Order
`( )`	`(expr)`	Explicit precedence.	nesting
`.`	`obj . id`	Object property access.	left to right
`[ ]`	`list[int]`	List/array element access.	left to right, nesting
`{ }`	`map{string}`	Map element access.	left to right, nesting
`!`	`! bool`	Logical negation.	right to left
`*`	`num * num`	Numerical multiplication.	left to right
`/`	`num / num`	Numerical division.	left to right
`%`	`int % int`	Division modulus.	left to right
`+`	`string + string`	String concatenation.	left to right
`+`	`num + num`	Numerical addition.	left to right
`-`	`num - num`	Numerical subtraction.	left to right
`&&`	`bool && bool`	Logical 'and' operation.	left to right
`||`	`bool || bool`	Logical 'or' operation.	left to right

Table 6.2: Expression Operators

The choice of expression atoms is straightforward. Constants of the basic data types are provided for, as well as identifiers, giving access to widgets and variables. Table 6.1 lists the kinds of atom with a lexical definition and an example each.

Choosing operators is more difficult. Property access and list indexing are needed to fulfill the requirements, but there is a wide range of numerical and logical operators to choose from. Since it is possible to add more operators later in a backward-compatible way, it is not critical to be comprehensive right from the start. The choice was made to provide just some basic numeric operations. Adding more features like functions or more sophisticated operators would allow more application code to be replaced with declarative descriptions, but that wouldn't provide a significant benefit for this work and its limited prototype.

Table 6.2 shows the selected operators, grouped and sorted by descending priority. The names in the scheme column represent expression parts that evaluate to a certain type. `id` represents an identifier, `num` either a integer or a floating point number, and `expr` can be any kind of expression. Dynamic typing and casting happens as required.

6.2 Automatic Synchronization

This work aims at simplifying common tasks in graphical user interface programming by offloading them to a description language. One such task is tracking changes in the application state and presenting them on screen by adjusting widgets.

6.2.1 The Concept

Change Propagation is traditionally done by procedural widget manipulation code that is placed anywhere in the application where the state changes. Since the situation is slightly different in each such place, there is a desire to exploit these differences (e.g. knowing that a certain value cannot occur in one place of the code) to tailor the widget manipulation code to the situation. This ultimately leads to code duplication and bad maintainability.

The Model–View–Controller pattern tries to improve application design in this regard by separating state-keeping and interface presentation. But often there still is a rich interface of notifications between the model and the view or controller component that is responsible for tracking changes for the user interface.

This work tries to take this one step further. Instead of specifying what should be done in response to certain events, the developer describes the static relationship between application state and interface display. The task of detecting relevant changes and adjusting widgets accordingly is thus offloaded to the system and happens automatically.

Such a high-level specification requires a generic data access interface. The abstract object model (see Chapter 4) provides that interface, including asynchronous change notifications. Expressions are used to allow complex addressing as well as on-the-fly transformation of data for presentation.

6.2.2 Technical Matters

Data synchronization can be unidirectional or bidirectional. Since the application is regarded to be authoritative, one-way synchronization always means copying data from the application to the user interface. It also means that even in a two-way synchronization the initial value is taken from the application.

A one-way synchronization requires one expression, while a two-way synchronization may require up to three expressions. This is a result of expressions being GUI-side entities and the problem that not all expressions are assignable.

To demonstrate this, we'll have another look at the Centigrade vs. Fahrenheit example. To transfer the value from the application to the interface (dubbed "reading" for simplicity) it is sufficient to evaluate the expression `regulator.temp * 9/5 + 32`. But this expression cannot be used as-is for transferring a changed value back to the application ("writing"). That requires knowledge of an application property to write to (i.e. addressing only) and knowledge of a value to write. In this case, the value would be computed using `(value - 32) * 5/9` and the destination would be given as `regulator.temp`.

Of course, there are simpler cases where data is simply moved around unmodified between an application property and a widget property. Here one expression is sufficient even for two-way synchronization. This can be seen as a degradation of the three-expression model where the "read-from" and "write-to" expressions are identical and the "write-transform" expression defaults to the identity function.

6.3 XML Notation

Data synchronization introduces two new tags, one as a generic wrapper for expressions and one for automatic data synchronization. These are not the only tags that use expressions, but they are the only generic ones that can be introduced at this point.

6.3.1 The expression Tag

The `expression` tag is a generic wrapper for an expression. It can be used anywhere a value is required, i.e. in exchange for one of the value tags like `string`. This expressly includes the insides of `list` and `map` tags.

The expression string is given as the content of the tag. This follows the markup spirit of the value tags.

The `expression` tag represents a passive entity. The expression it wraps is only evaluated when asked to and does not track changes to its value.

Listing 6.1: Basic Expression Example

```
<expression>percent_field.text</expression>
```

6.3.2 The sync Tag

The `sync` tag is used to denote automatic data synchronization. It can occur inside a `property` or `variable` tag, replacing the initial value that is otherwise given there. In particular, it is not possible to mix the `sync` tag with the `list` tag, as is possible with the `expression` tag. The `sync` tag controls the variable or property as a whole, giving it both an initial value and ongoing adjustments.

Both one-way and two-way synchronization are written using the `sync` tag. They differ in which of the three possible expressions are given in the tag. A one-way synchronization only has a "read-from" expression, while a two-way synchronization must also have a "write-to" expression and may have a "write-transform" expression.

Since there may be several expressions, they are all given as attributes. For flexibility there are four possible attributes that can be combined to set the three possible expressions. The `from` attribute sets the read-from expression. The `to` attribute sets the write-to expression. If both are identical, the `with` attribute can be used instead to set the read-from and write-to expressions at once. Finally, a write-transform expression can be given using the `expression` attribute. It uses the special name `value` to refer to the current value of the synchronized widget property or variable. When omitted, the write-transform expression defaults to the identity function.

Listing 6.2: Automatic Synchronization Examples

```
<sync from="app.version" />
<sync with="app.prefs.magnification" />
<sync from="regulator.temp * 9/5 + 32"
  to="regulator.temp" expression="(value - 32) * 5/9" />
```

6.4 Parsing Expressions

Expressions are used throughout the system for various purposes. In addition to parsing and evaluation, they must track changes to their value and notify the owner every time a potential change occurs. All of this is implemented in the class `Expression` and its private helper classes. `Expression` has a rich interface, including several constructors, evaluation and assignment methods, and a Qt signal for change notifications. The following sections cover the various operations an `Expression` goes through in its life cycle.

6.4.1 Creation and Parsing

The prototype implementation treats all value tags as an expression, not just actual `expression` tags or embedded expressions. The benefit is twofold. First, this arrangement allows an `expression` (or a `widgetref` for that matter) to occur inside `list` tags and similar constructs. Second, it greatly simplifies implementation because there is just one kind of object to handle, the only exception being automatic synchronization for properties and variables.

As a result, the `Expression` class has three constructors. The first one takes a string and is intended for expressions given as attributes. The second one takes a single XML DOM node, suitable for instance for the contents of a `property` tag. The third one takes a list of XML DOM nodes and constructs an expression that evaluates to an appropriate list value. This is intended for special tags that take a list of values, e.g. parameters for a method call.

All three constructors parse their arguments immediately to form a private tree of expression nodes, subclasses of `ExprNode`. Node creation for value tags is straightforward. It uses the subclasses `ConstantExprNode`, `ListExprNode`, `MapExprNode`, and `WidgetRefExprNode`.

Expression strings is less easy and requires proper parsing. Since expressions have a very simple syntax, no standard lexer and parser tools are used. Instead, a custom parser directly generates node objects and iteratively resolves operator precedence. Parsing happens in two phases. The first phase scans the expression string for tokens, building a flat list of node objects. Actually, the first phase also resolves parentheses nesting using a stack, so the result is better considered a broad tree. In any case, the second phase iteratively refines this tree by looking for the lowest-precedence operator and splitting the containing list at that position. This process is illustrated in Figure 6.1, using the expression `5+3*7+offset` as an example.

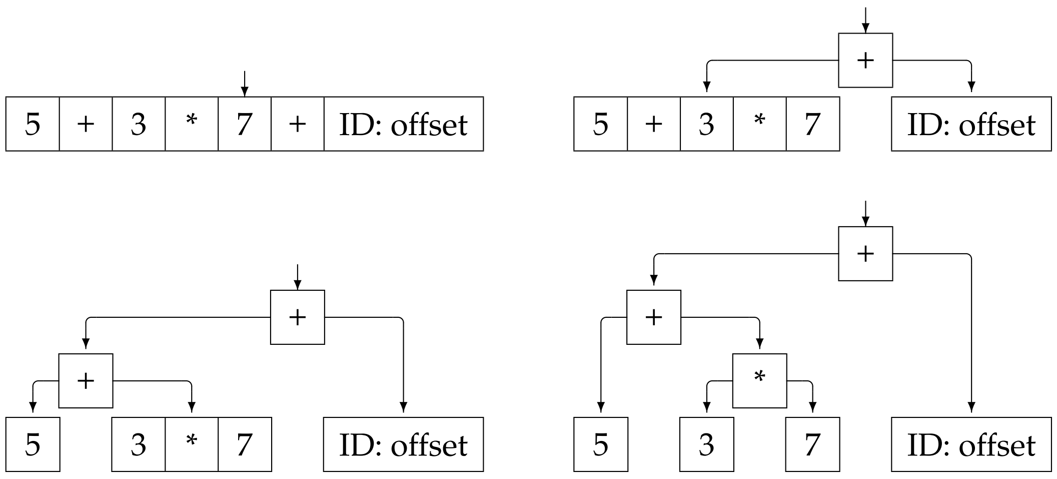


Figure 6.1: Expression Parsing, Phase Two

6.4.2 Identifier Binding

Before an expression can be evaluated, it must resolve any identifiers it might contain; this process is called binding. It cannot happen at construction time. An `Expression` object is constructed as soon as the expression is encountered while traversing the interface description. A widget name referenced by the expression may not be known at that time if that widget hasn't been seen yet. Therefore, expressions must be bound in a second phase. This extends to all objects that keep expressions around, so it is almost universal.

Since expressions try to be as generic as possible, they do not include the machinery to look up identifiers. Instead, they rely on an external object that implements the `Context` interface[1]. That interface provides two lookup methods with slightly different semantics. Their signatures are shown in Listing 6.3.

The first method, `resolveIdentifier`, is used to resolve identifiers that occur in expression strings, while `resolveWidgetName` is used to resolve `widgetref` tags.

[1]The term "interface" is used here in the sense coined by Java. In C++ terms, `Context` is an abstract mix-in class. It does not exploit the possibilities of multiple inheritance, however, so the Java term is more appropriate to its role.

Listing 6.3: Identifier Resolution Methods

```
public:
  QObject * resolveIdentifier(const QString &name);
  Widget * resolveWidgetName(const QString &name,
    bool cloneOkay = false);
```

The second parameter of `resolveWidgetName` is used in other places; `Expression` ignores it since it only cares about translating a name string to a `QObject` pointer. The functional difference between the two methods is that `resolveWidgetName` ignores variables and only looks at the widget namespace.

It should be noted that this lookup happens *only* at binding time. The resulting pointers are stored in the expression nodes for later use. It is also interesting to note that there is special treatment for `Variable` and `Synchronizer` objects, since they directly represent a value, while widgets and application objects are compound entities that have properties.

6.4.3 Evaluation and Assignment

Once an expression is bound, it is ready to be evaluated. Calling `evaluate()` traverses the private tree of expression nodes and finally returns a `QVariant` with the calculated value. Actually, the interface of `ExprNode` closely mirrors that of the front-end `Expression` itself. Each `ExprNode` subclass is responsible for responding to `evaluate()` in the appropriate way. A `ConstantExprNode` will just return its stored value, while a `BinaryOpExprNode` will evaluate its left and right sides and compute a value based on the operands and its operation.

The `Expression` class also supports assignment. Assigning a new value is only supported by expressions that represent an actual data storage location. Most of the time this means variables and object properties. Assigning to an element of a list or map is also supported. It is implementing by retrieving the list or map value from storage, changing the requested element, and finally writing back (i.e. assigning) the complete list or map.

6.4.4 Watching and Notification

Any `Expression` watches its value for changes, regardless of a possible lack of observers. This choice was not made for simplicity, but because of the properties of remote communication. Watching for changes requires telling the communication subsystem which properties of which objects the interface will be interested in in the future, allowing these to be fetched asynchronously and cached for instant access.

During the call to `evaluate()`, a mark-and-sweep algorithm is used to determine the properties to be watched. While traversing the expression node tree, every property access is recorded and added to the internal data structures of the `Expression`. If a new object or property is encountered, it it set up for watching. At the end of the evaluation, any properties that were not accessed in this run are disconnected. This seemingly verbose approach is required because for instance a changing list index can change the object being accessed.

While there is no explicit method to enable or disable watching, the implementation requires that `evaluate()` is called at least once to do the initial setup using the algorithm described above. After that, the `Expression` is "on guard" and ensures that it stays up to date on its own. To be notified of possible changes, the listener object connects itself to the `changed()` signal of the expression.

6.5 Implementing Synchronization

Automatic data synchronization as denoted by the `sync` tag is implemented in the `Synchronizer` class. Since synchronization is based on three expressions, all that is left to do is some logic and connectivity.

When a `Synchronizer` is constructed, it is given two pieces of information: the data object to synchronize (a widget property or a variable) and the expressions as extracted from the `sync` tag. The write-to and write-transform expressions may be empty; the presence of a write-to expression enables two-way synchronization.

In the binding phase all present expressions are bound and watching is enabled on the read-from and write-transform expressions. The write-to expression does not need to be watched; it is only used as the receiver for assignments. Finally, the read-from expression is evaluated to get an initial value for the property or variable.

After initialization, three events can happen at any time:

- The read-from expression signals a change. In response, it is evaluated and the result written to the property or variable.
- The property or variable changes. This is detected using the general property change notification of the object model. The `Synchronizer` responds by extracting the new value from the notification, applying the write-transform expression if present, and assigning the resulting value to the write-to expression.
- The write-transform expression changes. This can only happen if the expression references another property, but it still requires that a new value is calculated and assigned to the write-to expression.

The handling of the write-transform expression requires some more explanation. Its purpose is transforming values entered in the user interface on their way to the application. This means it is used as a function with one argument, the new value as read from the user interface. This argument is provided in the special variable `value`. To implement this, `Synchronizer` redirects the expression's identifier lookup. It implements the `Context` interface and passes itself to the expression in the binding phase. A request for the special name `value` is answered with the `Synchronizer` object itself. All other requests are passed on to the owning `Widget`. The expression recognizes that is has received a `Synchronizer` and immediately accesses the `value` property of that object. This differs from the standard treatment for application objects and widgets, which are passed on as a value of type ObjectPtr.

Chapter 7

Events and Actions

"They're funny things, Accidents. You never have them till you're having them."
Eeyore in *The House at Pooh Corner*

7.1 The Event-Driven GUI

The purpose of a graphical user interface is to react to user input like mouse movements and key presses in a meaningful way. The canonical way for such user input to enter the application's realm is as asynchronous events.

On a fundamental level, physical actions by the user cause events like "mouse moved to coordinates ..." and "key ... was pressed". If GUIs were monolithic and applications would run one at a time, that would probably be all. But with window systems and component-based toolkits arise the needs of both higher-level events and event distribution.

Event distribution concerns itself with the question of where to send an event. A common problem is locating the widget under the mouse cursor, or tracking the widget that currently receives keyboard events. Event distribution also means finding an object that knows how to deal with an event, especially a high-level event. This usually involves traversing the view hierarchy or an additional command hierarchy that includes invisible application objects.

High-level events emerge from the interpretation of physical user actions by the toolkit or window system. It is convenient and customary to let them share the distribution system with low-level events. Many systems also use this same infrastructure for redrawing management and asynchronous notifications from other sources like timers or sockets, but that's not relevant for this work.

What makes high-level events relevant are the new requirements and attributes they introduce. For one, they are always associated with a certain widget[1]. They also tend to have less parameters than low-level events, often none at all, not counting the location (widget association) and type (what happened). Naturally there are more distinct types of high-level events than low-level ones.

How does all this fit into the abstract descriptive approach of SUIT? First of all, low-level events are completely hidden. They are handled by the toolkit or, if that isn't sufficient, by the system implementation. The same applies to redraw (or "expose") events; the application does not want or need to be concerned with these. That leaves the high-level events, each with a specific meaning and a context.

Based on this analysis the following event model was derived. An event is identified by a string name and carries no further parameters. It happens "at" a certain widget and may be handled there by handlers that match the name. If it is not handled, an event may be propagated upwards in the view hierarchy (towards the root), but this must be enabled explicitly.

This simple design has two weak spots: parameters and propagation. The case where an event should carry an argument must be handled in another way on an individual basis. Here widget properties offer themselves, leading to a model where the

[1]Application-global events can be associated with a special Application widget.

event only serves as an interrupt to trigger re-examination of the widget state. As for propagation methods—additional, more sophisticated methods can be implemented within the bounds of the system (and actually are).

7.2 Sources for Events

The main source for events has already been discussed. Low-level events generated from physical user actions get interpreted by widgets and are translated into high-level events. These enter the event handling system of the semantic user interface. The names of these events are set by the widgets that fire them. They can be widget-specific, but there are also some generic names like `action` (used among others by push buttons and menu items) or `close` (used by windows and the application as a whole).

Most applications also have the need to fire events to notify the interface of a new situation. Some of that need is already covered by automatic data synchronization. However, this cannot cover each and every aspect, so there must be a way for the application to fire generic events. Since the application has no knowledge of the view hierarchy, firing events must happen in cooperation between the application and some entity on the interface side. The only suitable messaging transport for this purpose are properties and their change events. In summary, custom events can be fired by watching an application property for changes.

Finally, the event handling infrastructure provides a way to fire more events. This is intended to provide the developer with a flexible way to propagate or rename events as required.

7.3 Event Actions

In traditional applications events are handled by calling a piece of application code. The presented system also provides for this, but since the application cannot directly influence the interface through API calls, there must be additional means to react to events within the descriptive interface. This is realized by providing a range of *actions* that can be combined to form event *handlers*.

Actions can be very generic (e.g. setting a property), but also very specific (like showing a window). Conditionals are also provided for. This section continues with an overview of the generic actions. Specialized actions are introduced in later chapters in context.

- Setting application properties. This is an alternative to automatic data synchronization when data needs to be sent back to the application for processing. Since data access happens via a "set" method, this can also cause further action and could be used as a trigger.

- Setting widget properties or variables. This allows the interface to be adjusted in reaction to the event. For example, a group of widgets can be disabled if a certain choice is made in another dialog. The actual possibilities are only limited by what a widget allows to be controlled through properties.
- Calling a method of an application object. This gives the application the chance to react to the event in any way it wants. However, its reactions must be conducted back to the interface using application object properties.
- Sending another event to a widget. Both the event name and the destination widget can be determined by the developer. With this action, events can be translated or renamed and forwarded to other widgets or central locations for handling. Dispatching an event to the currently focused widget or one of its parents is achieved by the special widget name `focus`; this is especially useful for menu items.
- Focusing a widget. Focusing is a special interface adjustment that cannot easily be expressed through properties. A focus action affects not just the targeted widget, but also its parent and ancestors. The effects vary with the classes of the affected widgets. One possibility is moving keyboard focus to the widget, others are bringing a window to the front and switching pages in a tab dialog.
- Conditionals allow a handler to execute different sets of actions based on the value of an expression. This can be combined with a property-watching event source to watch only for specific values.

Like all other constructs, an event handler is declared in the context of a widget. It is only activated if this particular widget receives the event the handler was declared for. Handling an event that occurs in another widget usually requires a dispatching handler in that other widget to re-send the event to the proper place, as widget-generated events are never propagated in the view hierarchy by default.

7.4 Describing Events and Actions

This chapter introduces the XML tags used to describe event generation and handling.

7.4.1 Event Sources: The watch Tag

Events that are generated by widgets in response to user action exist of their own right. Their generation does not need to be reflected in the description language, only their handling. Custom events generated by property watching must be set up explicitly, though.

The `watch` tag defines a property watch that locally fires an event. It must be given as a child of a `widget` tag. Actually, not a property is watched, but an arbitrary expression, which may include multiple application properties or even widget properties. The expression is specified using the required `expression` attribute. The name of the event to be fired is given in the required `event` attribute. The event is fired locally (i.e. only for the widget containing the `watch` tag) and is not propagated automatically. For finer control (e.g. to send an event when a property takes a certain value), a local event handler can be used in combination with conditional action tags and the `sendevent` tag.

7.4.2 Event Handling: The action Tag

An event handler is introduced by the `action` tag. That tag must be the child of a `widget` tag and may contain any number of actual action tags, some of which are defined below. The name of the event that should trigger the handler is given in the required `event` attribute to the `action` tag.

7.4.3 Action Tags

These tags can occur inside an `action` tag to denote the actions to take when the event handler is called.

The sendevent Tag

This action tag is used to send an event (not necessarily the same one) to other widgets. The required `event` attribute gives the name of the event to send. The required attribute `to` specifies the destination. The destination can be the name of a widget, or a special value. The special value *focus* sends the event to the currently focused widget (useful for menu commands). The special value *this* designates the widget where the event handler is declared. The optional attribute `propagate` enables or disables propagation of the event, i.e. if it is not handled in the target object, the search for a handler continues with the parent, recursively, until the root of the view hierarchy has been reached or the event was handled. Propagation is enabled by default with the `sendevent` tag.

The assign Tag

The `assign` tag can assign values to an lvalue expression, i.e. any widget or application property, or a free variable. The required attribute `to` gives the destination for the assignment (an lvalue expression). The value to assign is given as the content of the tag, using value tags and expression tags as required (i.e. `string` and friends are allowed as well as `widgetref` and `expression`).

The methodcall Tag

The `methodcall` tag is used to call methods of an application object, optionally passing parameters. The call happens asynchronously; no value is returned. There are two required attributes, `object` and `method`. The `object` attribute contains an expression that must evaluate to an object reference pointing into the application. The `method` attribute is a string giving the name of the method.

The tag may contain any number of value or expression tags, like the `list` tag. These are evaluated and passed to the method as parameters, in the order specified.

The focus Tag

The `focus` tag denotes a focusing action. The required attribute `ref` gives the name of the widget to focus. The special name `this` is recognized, but not `focus`, since it doesn't make sense for this kind of action.

The switch, case, match and else Tag Consortium

This group of tags provides conditional action execution, based on the value of an expression. At the top is a `switch` tag, giving the expression to evaluate in its required `expression` attribute. The `switch` tag can contain any number of `case` and at most one `else` tag. A `case` tag must contain a `match` tag, which in turn contains any number of value specifications, including expressions as usual. If the value of the switch expression matches any of the values given in the `match` tag, the action tags inside the `case` tag are executed. The actions in the `else` tag are executed if no `case` tag matches.

7.5 Implementation Aspects

7.5.1 Event Handling Infrastructure

Since events are completely generic, most of the infrastructure can be implemented in the base class, `Widget`. Listing 7.1 shows the two methods used for event handling.

The `postEvent` method can be called from anywhere to inject an event into the system. If required, automatic propagation can be enabled. Other than that, only the name of the event is needed, since events have no further arguments. When called, Listing 7.1: Event Handling Methods

```
public:
  void postEvent(const QString &eventName,
    bool propagate = false);
protected:
  virtual bool eventDefault(const QString &eventName);
```

`postEvent` will look at the table of event handlers generated from `action` tags. This table is also maintained by the `Widget` class.

If no event handler is found for the event, the `eventDefault` method is invoked to allow default processing to take place. This method serves as a hook for subclasses that have such needs. For example, a `Window` widget uses this hook to close itself if the `close` event is not handled. If an event was handled by `eventDefault`, it must return `true` to indicate this.

If all attempts to handle the event fail and propagation is disabled, the event is dropped on the floor without further ado.

7.5.2 Action Classes

The various actions to be executed when an event handler is hit are stored as objects using a small class hierarchy. This allows actions to be treated as black boxes once they have been constructed from the XML description. The base class is called `Action`. It provides the usual construction and binding facilities, plus the one all-important method `execute()`. Since actions may need to know where in the view hierarchy they are executed, each action is provided with a pointer to its associated widget, stored in the member variable `context`.

Most action classes are derived directly from `Action`. Another base class is provided for those places where an action consists of a list of member actions that are to be executed in turn. This class is called `ActionList` and is used to create event handlers in the first place, and for the conditional switch-case-else construct.

7.5.3 The Focus Problem

One particular feature turns out to be more difficult to implement than expected: sending an event to the currently focused widget. Fair enough, Qt provides a method to find the currently focused `QWidget`. But to send an event in our own widget tree, this `QWidget` must be mapped back to the owning `Widget`.

In the prototype, this is solved by manually maintaining a mapping table. Each `QWidget` is registered in a global dictionary as it is created. The `RealWidget` class tries to reduce the burden by providing registration and unregistration of the `qw` member, unless `create()` is reimplemented by the subclass.

The table is maintained and used by the following methods in class `Widget`:

Listing 7.2: QWidget Registration Methods

```
protected:
  void registerQWidget(QWidget *qw);
  void unregisterQWidget(QWidget *qw);
public:
  static Widget * getFocusWidget();
```

The `registerQWidget` adds the given `QWidget` to the lookup table, pointing to the respective `this Widget`. As the name implies, `unregisterQWidget` removes the entry. The complete lookup process is handled by `getFocusWidget`. It first uses `QApplication::focusWidget()` to get the focused `QWidget`, if any, and then tries to find a `Widget` for it in the lookup table. If no entry is found, it will retry the parent `QWidget`, until a match is found or the root has been reached. If the match cannot be made or Qt reports that no widget is focused, a null pointer is returned. `SendEventAction` generates an error message in this case.

Chapter 8

Window Handling

"I've found somebody just like me. I thought I was the only one of them."
Tigger in *The House at Pooh Corner*

8.1 Introduction

This chapter concerns itself with top-level windows, their special requirements, and how these can be met by a description language.

The contents of windows are relatively fixed. They either do not change over time, or the change can be handled by special widgets like tabs. This does not hold true for the windows themselves, though. They are created and destroyed (or shown and hidden) in response to user actions and application requirements. Often there are several windows with an identical widget layout, but different data. The main use for this is in applications that work on several documents at once, but it also applies to objects of other kinds, e.g. a dialog for editing a financial record in a bookkeeping application.

These requirements are implemented with several constructs in the proposed architecture. There are special actions for opening and closing windows as well as a trigger concept to give the application control over opening and closing windows if required. The generic concept of cloning allows several instances of a part of the widget tree to exist simultaneously, each with its own application data association.

The following sections each focus on one of these concepts, detailing design, language representation and implementation. Cloning is covered first, because it also affects the other concepts (and the whole architecture for that matter).

8.2 Cloning

Cloning of windows allows several instances of an application object class to be presented to the user simultaneously.

8.2.1 The Concept

The idea of cloning is simple: a part of the interface widget tree is duplicated and goes on to live a life of its own. Since the new instance has its own local data storage independent of other instances, it can be used to represent and manipulate a separate application object. This is very similar to the idea of object-oriented programming, where classes define behavior and the application creates the required number of instances to hold and manipulate data.

To make cloning more useful, this work adds some convenience features to the basic idea. Each clone instance is given a special variable to hold a reference to the application object it represents. Furthermore a cloning template can be associated with an application property containing a list of objects. That property is watched for changes and clones are automatically created and destroyed to match the list of objects as it changes.

Cloning has a wide impact on the architecture. Contrary to what was postulated before, a widget name no longer uniquely addresses a widget in all situations. In effect, cloning creates a border for name accessibility in the widget tree. While a widget inside a cloned section of the tree can reference widgets in other sections along the path to the root, the reverse way is blocked. Some accessibility can be regained by using a pair of a widget name and an application object, but this only works where cloning is not applied recursively (which otherwise is perfectly legal and useful). These considerations do not apply to free variables, since they are only made available to direct children anyway.

8.2.2 Notation

A widget is marked as cloneable by adding a `clone` tag to its content. The `clone` tag contains further information in the form of three attributes. The `mode` attribute is required and specifies either `manual` or `auto` cloning, using these keywords. If automatic cloning is used, the attribute `from` must contain an expression that evaluates to a list of object references. This expression will then be watched and used to automatically create and destroy clone instances. The last attribute is mandatory and is called `variable`. It gives the name of a free variable to create. The effect is identical to the tag `<variable type="object" name="`*`name`*`"/>`, but with the addition that the value is set implicitly by the cloning machinery. In this respect, there is no difference between automatic and manual mode, since there is always an application object associated with a clone instance.

Listing 8.1: Automatic Cloning Example

```
<widget class="Window">
  <clone mode="auto" from="app.documents" variable="doc" />
  <property name="title"><sync from="doc.title" /></property>
  ...
</widget>
```

8.2.3 Implementation

Cloning has far-reaching effects on all parts of the implementation. Any widget and any object that lives in the context of a widget must be able to create a copy of itself while maintaining the proper links between objects. Unfortunately C++ can't help with this, so it must be implemented manually.

All three class hierarchies—widgets, actions, and expression nodes—are affected by this. C++ provides no way to generically invoke constructors, so each subclass implements a virtual method that uses the correct constructor and returns the newly created object. This method is always called `clone` to reduce confusion. It takes the parameters necessary to maintain the links between objects, usually some kind of "parent" pointer. The actual work of copying the object's data and any dependent objects is done by the constructors. Most classes have two constructors, one for construction from XML and one for cloning.

Classes that are not part of one of the class hierarchies usually also have a separate cloning constructor, but no `clone` method, since their class and by extension the constructor are always known. `Synchronizer` is an example for this.

Cloning happens at a very early time in the life cycle of a widget. This is a consequence of the template–instance model used—the model actually assumes that instances are created from the XML description. The implementation instead creates clone instances from widget objects that were created from XML, but not initialized further. This has the welcome side effect of severely reducing the effort of actual cloning. For instance, references to other widgets are kept as name strings in the template and are resolved into object pointers only after cloning has taken place. This reduces the number of pointers that need adjustment during cloning.

With cloning taken into account, the life cycle of a widget is as follows:

- Creation. The XML description is parsed, the `Widget` subclass and dependent objects are created and arranged in a tree. Expressions are also completely parsed in this phase, but all identifiers and widget names are stored only as strings for now.
- Cloning. Copies of the still-inactive objects are created, copying most data as strings. The tree structure is preserved in the copy by adjusting linking pointers accordingly.
- Binding. Identifiers are resolved into object pointers. Initialization expressions are evaluated and the results assigned to properties and variables. Watching is set up.
- Creation. In another tree traversal Qt widgets are created, initialized, and arranged.
- Normal Life. What now happens is up to the application. It is given a chance to initialize the interface, e.g. by opening some windows. Note that a widget may be hidden or shown together with its enclosing window multiple times during normal operation.
- Destruction. While a normal widget is not deleted until the application terminates, a clone instance can be deleted at runtime. Both the Qt widgets and the abstract widgets are destructed.

The life cycle portrayed here applies to individual widgets, not the description as a whole. In particular, clone instances are not initialized together with normal widgets, and that must be enforced in the tree traversals in `bind()` and `createAll()`.

Special dispensation is needed for automatic cloning, since it is based on a watched expression.

The widget namespace is partitioned by cloning, meaning that it cannot be stored in a single, global place. Instead it is stored at the root widget of each tree partition, i.e. at the root of the tree and in any widget that contains a `clone` tag. The namespace table is copied along with other data, with the added difficulty that the pointers it contains must be adjusted in the process. It turns out that it is easier to rebuild the table from scratch as cloning progresses, i.e. each cloned widget re-registers its own name in the namespace of the newly cloned tree partition. This task is taken on by the cloning constructor of the `Widget` class.

While cloning can in theory be applied to any kind of widget, it is most often used on windows. For convenience, cloning a `Window` or a similar widget is coupled with opening and closing, as will be detailed in the next section.

8.3 Opening and Closing

Opening and closing is a feature unique to windows and often happens as a direct reaction to a user action, like a click on a push button. Thus, one approach at handling window visibility is based on action tags for event handlers.

8.3.1 Design Considerations

There is a strong interaction between window visibility and cloning. A non-cloneable window is merely hidden from view when closed and may reappear later. For a clone instance however closing means destruction, since it is no longer useful. Likewise, the creation of the clone is coupled with window opening. The result is twofold: First, the action tags for opening and closing can cause cloning actions and must be able to identify a specific clone instance. Second, automatic cloning causes automatic opening and closing if applied to a window.

8.3.2 The openwindow Tag

The `openwindow` tag is an action tag, i.e. it can occur inside an `action`, `case`, or `else` tag. It has the effect of showing the designated window, or bringing it to the front of the window stack if already visible.

The required attribute `ref` gives the name of the window widget to be opened. For cloneable windows, the additional attribute `with` gives an expression that must evaluate to an object reference. This object reference is used to identify a clone; a new clone is created if no clone exists for the given object reference.

8.3.3 The closewindow Tag

The `closewindow` tag is also an action tag. It causes a window to be hidden from view. If the window is a clone, it is also destroyed. `closewindow` takes the same attributes as `openwindow`: the `ref` attribute gives a widget name and `with` gives an additional expression to identify a specific clone using its associated object. However, `closewindow` is more flexible in finding its target window. This is because `closewindow` can be used from inside the affected window. The `ref` attribute can be omitted, in which case the innermost enclosing window of the context becomes the target. The `with` attribute serves to uniquely identify a target and can be left out if a widget name is sufficient—in the current context, that is.

8.3.4 Implementation

The implementation of window opening and closing centers around the common base class `WindowBase`. It provides the following public methods:

Listing 8.2: Methods for Opening and Closing

```
public:
  void openWindow();
  void closeWindow();
```

These methods are called by the cloning methods in class `Widget` as well as by the classes implementing the action tags, `OpenWindowAction` and `CloseWindowAction`. In all cases, the target object is checked using Qt's runtime type information before being casted. Destruction of clones happens automatically in the implementation of `closeWindow`.

8.4 Triggers

Triggers offer an alternative for opening and closing of windows that puts control in the hands of the application itself.

8.4.1 When Events Just Won't Do

Event actions are appropriate for many cases, but unfortunately not for all. Some applications are not completely event-driven, but require user input in the middle of a longer operation. It would be possible to rewrite these applications, but this means saving the state of the computation and resuming it at the right point later, after the desired event has arrived. This often results in increased coding effort as well as reduced maintainability, and still leaves the problem of notifying the user interface of the application's data requests.

To illustrate this, consider a graphics application that lets the user save images in different file formats. It can infer the format from the file extension the user enters in the file selection dialog. However, some formats require more information, a compression ratio for example. The decision which information is needed is made by the application after the file selection dialog was closed, and processing must resume once the user provides the required information.

Triggers defuse such situations by providing a well-defined flow of control between the application and the user interface, built on the existing infrastructure. When the application realizes more user input is needed to complete the current processing task, it activates the trigger and the current processing thread is suspended. The activation causes a designated window to be shown in the user interface. The window is responsible for getting the required input from the user and storing it into application properties. Closing the window causes the trigger to return to its inactive state and resumes the application's processing thread.

This signaling is realized using an application property of the Boolean type. The trigger is activated by setting the property to `true`. The user interface watches the property and thus receives a change notification indicating the activation. When the dialog is finished, the user interface sets the application property to `false` again. The application side watches for this change and takes the appropriate actions.

This flow of control could be implemented with existing means—property watches, event handlers, window actions. Triggers are provided to reduce the coding effort and to improve maintainability at the same time.

A trigger takes exclusive control of a window's visibility. As a result, at most one trigger can be attached to a window, and triggers and cloning are mutually exclusive.

8.4.2 XML Description

To set up a trigger, the user interface engine needs to know which application property is used for signaling and which window widget should be opened and closed in response.

The widget association is declared through nesting, as usual. The `trigger` tag must be a child of the `widget` tag it should apply to. The application property is given as an expression in the required attribute `on`.

Listing 8.3: Trigger Specification Example

```
<widget class="Dialog">
  <property name="title">
    <string>JPEG Compression Settings</string></property>
  <trigger on="doc.jpeg_trigger" />
  ...
</widget>
```

8.4.3 Realization in the Prototype

Implementing triggers affects both the user interface and the application support code, as both must work together to ensure the desired flow of control.

On the user interface side, the `trigger` tag is translated into an object of class `Trigger`. The `Trigger` sets up the expression for watching in the binding phase. When the expression changes to `true`, the `Trigger` calls the `openWindow` method in its associated widget—provided it inherits from `WindowBase`, that is. `WindowBase` in turn notifies the `Trigger` when the window is closed through `closeWindow`. The `Trigger` notifies the application by assigning the value `false` to its expression.

On the application side, a boolean property is required. It must be provided by the application developer. This may seem inconvenient, but any help given by the system would still need to be integrated into the application object in question, and thus cannot reduce coding effort significantly. After all, the boolean variable must still be accessible as a property, directly or indirectly.

For actual trigger handling, the developer has the choice between custom code and a convenience function provided by the application support library. The following describes the interface and working of the convenience function. Custom code must provide the same basic behavior, but may choose its own way when it comes to suspending and resuming processing.

The `suit_trigger` function needs to know what property to use and takes two parameters for this: a pointer to a `QObject` and a string naming the property. The function sets the property to `true`, then arranges for it to be watched and puts the current thread to sleep. The thread is woken up again after the property has reverted to `false`, `suit_trigger` returns, and processing resumes normally.

Suspending arbitrary application threads usually means asking for trouble. The prototype avoids potential unresponsiveness with a pool of worker threads. When an application method is called in response to a user action, one thread is taken from the pool to run it. This ensures that the user interface and communication engine keep going even if the application method takes a long time to complete.

The issue of application data integrity remains. Unless the application signals that it is thread-safe, a global application lock will be used to protect data integrity. Before `suit_trigger` suspends the current thread it releases that lock temporarily. The application developer is responsible for ensuring data integrity before calling `suit_trigger`.

Chapter 9

Remote Communication

"No. That wasn't me. Perhaps it was
somebody else."
"Well, thank him for me when you see him."
Pooh and Eeyore in *Winnie-the-Pooh*

9.1 Design Considerations

The topic of this chapter is the communication protocol that allows the graphical user interface and the application code to run on different machines.

9.1.1 The Task

The abstract object model already provides a solid foundation for a generic protocol. What remains to be done is designing the message exchanges to replace various local function calls and their encoding on the wire. Also, a model for session initiation must be chosen.

The design of the protocol is driven by several needs. Going from local method calls to remote messages introduces the possibility for significant delays. The protocol and its implementation infrastructure must try to reduce the impact this has on response time and execution efficiency. The amount of data sent over the wire is also significant and must be kept low, as that can improve performance in low-bandwidth scenarios. These goals are inherent to any distributed system. The system designed in this thesis has a very broad scope and aims to be both platform and language agnostic. This creates the requirement to leverage open standards. Keeping the protocol flexible and extensible is also important for an open system.

9.1.2 Protocol Functionality

Before talking about technical aspects like message encoding and framing, the actual functionality required from the protocol must be discussed. The abstract object model provides a good deal of functionality requirements: reading and writing object properties, change notifications, and method calls. But the protocol needs to do more than that, if it is to replace the local function call interface usually provided by a GUI API.

The first aspect is initialization. The user interface part is handled by generic code, so it must be told about the application. In particular it must acquire a copy of the XML description, but also a reference to the root object of the application, which directly or indirectly references all other application objects. The application code itself is passive most of the time and knows its own self, so it has no special requirements. The retrieval of information from the application is controlled solely by the interface description.

While most states and reactions of the application can be conveyed as property values or property changes, error reporting is a notable exception. Of course it would be preferable to anticipate certain errors and to provide a recovery path for both the application and the user, but this cannot be done for each and every error that might occur. Still, the user needs to be informed of such unexpected errors that simply stop the current operation. (And of course, the application must ensure that it always leaves itself in a sane state, but that is a different matter.) Most application frameworks provide a simple function to present an alert box or to write an entry into a log. So does the proposed system, with the difference that the text must be transmitted in a special protocol message. It is left to the interface server implementation to deal with the message; both an alert box and a (visible) log are acceptable.

And finally, there is the question of file access. Since the interface may be displayed on a different machine than the application runs on, it is not inherently clear which set of files is to be accessed. An ideal solution would offer the user (and the application) transparent access to the files on both machines. While some scenarios may already provide file-sharing between the affected machines through other mechanisms, a self-sufficient graphical user interface system would have to include file-sharing as a part of the protocol.

Since the scope of this thesis is limited, it was decided to limit it to file access on the application's machine. This choice eliminates the need for full file-sharing, but still requires protocol exchanges to allow browsing of remote files by means of a file selection dialog. `file:` URLs are used instead of plain file path names to allow for later extension of the mechanism.

9.1.3 Why BEEP?

The Blocks Extensible Exchange Protocol, also known as BEEP and standardized by RFCs 3080 [32] and 3081 [33], was chosen as a base for the custom protocol. BEEP is an application protocol framework, designed to relieve the protocol designer from common, recurring tasks. It provides message framing, multiple virtual channels on one connection, messages for session and channel management, and standard profiles for authentication and transport security. These common functions are not only pre-designed, but also pre-implemented in software libraries, thus further reducing the effort required to implement a new protocol.

Several alternatives were considered for the protocol. A full-custom protocol, possibly using a binary format, was ruled out because of the extra effort this would entail in comparison with a standards-based solution. This includes both the design for easy decoding and extensibility, and the actual implementation. In the real world, several independent implementations of the protocol would be needed, further increasing the desire for easy implementation.

RPC-like methods were also ruled out. In particular, SOAP [3] and XML-RPC [37] were evaluated. The two are quite similar as they trace back to a common draft. They implement a remote procedure call encapsulated as XML and transported over HTTP. For the system proposed here that model has a series of disadvantages:

- Both the RPC model and the HTTP transport are stateless and loosely coupled. This contradicts the strong coupling and persistent connection between application and user interface required in this field.

- While there is a comprehensive data model, there is no comprehensive object model, just methods for addressing, based on the styles usual for HTTP (URI) and RPC.
- Communication is inherently one-way. Since both sides in our model must be able to initiate an exchange, this would require two channels.
- For small messages, HTTP has a high protocol overhead.
- Authentication as provided by HTTP is per-message, not per-connection.

In contrast, BEEP is connection-oriented, provides per-connection authentication, and allows both peers to take an active role by initiating message exchanges. An appropriate object model can be built into the custom message protocol that lives on top of BEEP.

9.1.4 Why XML Messages?

BEEP gives the protocol designer freedom over what encoding to use for the messages exchanged. For this work, a simple XML representation was chosen. This was a natural choice as XML is already used in this work as well as in the BEEP base protocols. But in addition to reusing the existing data value encoding, XML also has clear advantages over a binary protocol. XML is human readable and a range of parsers are readily available. Processing is independent of platform byte order and other differences. It was felt that these advantages make up for the data volume that could be saved by a compact binary representation.

An even simpler text-based, but line-oriented encoding in the style of SMTP [19] or FTP [29] makes it hard to encode structured data like nested lists, so it was not considered.

9.1.5 Connection Initiation

There are several possible models how application and user interface find each other and initiate the connection. The main choice here is which side should wait for incoming connections. Both options have their merits. Luckily, BEEP makes this choice independent of the actual channel protocol. Once a transport connection is established, it becomes unimportant which of the peers established it. It is actually possible to create a solution that supports both ways simultaneously with little overhead.

For the prototype a model mirroring that of the X Window System [42] was chosen. A generic display server listens for connections on a well-known TCP port number. An application connects to the server at startup, transmits its user interface description, and waits for things to develop from there. In order to allow for multiple display servers on one machine, possibly run by different users, a display number is used to tell them apart. The display number is used to calculate the port number, starting from 9400[1]. The host name and display number can be provided to the application on the command line or through an environment variable. All of this is handled by the application-side library.

The other option would have been to let the application listen for connections. This is equally feasible, since the application is passive and keeps all the persistent state. The generic interface display program would contact it as a kind of remote control. If multiple connections are allowed, this opens the way for multi-user collaboration, thin-client application server environments, and even an alternative to web applications currently delivered as HTML and JavaScript. Exploring these possibilities is left to a future work.

If application and user interface run on the same host, it may be preferable to run the protocol over Unix domain sockets instead of TCP sockets for performance reasons. This is not implemented in the prototype, and no standard conventions are defined here. The prototype instead supports a direct linking approach where the user interface server is linked into the application as a shared library. This performs even better since there is no protocol overhead, but it doesn't work with arbitrary implementation languages without further measures.

9.1.6 Object References

Within the application or the user interface, object references can be stored and passed around as pointers. This doesn't work across the "wire", however. The canonical solution is to assign a unique identifier to each object and use that to represent to it. In this work integer numbers are used as identifiers. They are dubbed "object ids" and are maintained automatically by the application-side glue code.

All of this only applies to application objects. It is a fundamental idea of this work that the application is passive and doesn't know the details of the user interface. Therefore references to objects in the UI implementation cannot and must not be transferred across the wire.

9.2 Message Exchanges

This section develops the actual message exchanges of the protocol, including their XML encoding.

[1]The port range 9400 through 9499 is currently unassigned by the Internet Assigned Numbers Authority (IANA) and was chosen for the prototype without official allocation. If the presented system is to enter wide use, an official port number allocation must be pursued.

9.2.1 Introduction

BEEP places no restrictions on the content of messages, but it enforces a response-reply scheme within each exchange. The reply to a message (MSG) can be either positive (RPY) or negative (ERR). BEEP also allows for multi-part responses (ANS and NUL), but these are not used in this protocol. A default content scheme for ERR messages is defined by BEEP and used unmodified in this protocol. It provides for a three-digit error code accompanied by a plain text message.

BEEP treats the request message payload as opaque, so it is necessary to tell apart different kinds of message. Since XML is used for all messages, this is easily achieved by using a different root element for each kind of message.

9.2.2 Representing Values

Values are encoded in the same way as in the description language (see section 4.3 on page 35). This includes the tags `string`, `int`, `float`, `enum`, `boolean`, `list`, `map` and `key`. It does *not* include `expression` and similar evaluation constructs. Those are to be evaluated by the interface server; only the results are transmitted to the application.

Object references that occur in a data value are written using the object id and the `objectid` tag. The object id is given as the content of the tag, in the same encoding used for the `int` tag.

9.2.3 The Initial Exchange

The initial exchange must be the first request after opening the channel. In it, the application sends the XML interface description and the object id of the root application object to the user interface server. The user interface server parses the XML, constructs its internal state, sets the implicit `app` variable to the root object and sends an empty positive reply (unless an error occurred, of course).

The request root tag is `initialize`. It has no attributes. It must contain two more tags, `interface` and `rootobject`. The content of the `interface` tag is the interface description, properly escaped to prevent parsing by the message XML parser. The `rootobject` tag contains the object id of the application's root object.

9.2.4 Property Exchanges

Several exchanges are used to access the application's properties. The interface server can read properties, set properties, and ask for notifications for a property. The application can send change notifications to the server.

Reading and Watching

To read a property and to enable watching, the interface server sends a request with the `getproperty` tag. The required attribute `object` gives the object id of the application object in question. The required attribute `property` gives the name of the property to access. If the optional property `watch` is present, it gives the desired state of change observation as a boolean value. If the property is absent, the state of observation is unchanged.

The reply contains a value tag giving the current value of the property. If an error occurs, an error reply is sent instead.

Writing

To write to a property, the interface server sends a request with the `setproperty` tag. The required attribute `object` gives the object id of the application object in question. The required attribute `property` gives the name of the property to access. The value to be assigned is given as a value tag inside the `setproperty` tag.

The application responds with an empty positive reply or an error reply.

Change Notification

If a watched property changes value, the application sends a change notification to the interface server. The notification message uses the `propertychanged` tag. The required attribute `object` gives the object id of the application object in question. The required attribute `property` gives the name of the property that changed. The new value of the property is given as a value tag contained in the `propertychanged` tag.

The interface server responds with an empty positive reply or an error reply.

9.2.5 The Method Call Exchange

This exchange is initiated by the interface server when it is instructed to call a method in the application. The root tag of the request is `callmethod`. The required attribute `object` gives the object id of the application object in question. The required attribute `method` gives the name of the method to be called. The tag may contain any number of value tags that should be passed as parameters to the method.

The application responds with an empty positive reply or an error reply.

9.2.6 The Alert Exchange

The alert exchange is initiated by the application to display a message as an alert box or in a similar way. The message is displayed asynchronously. If feedback is required, other means must be used.

The request root tag is `alert`. The optional attribute `object` gives an object id with which the message is associated. This can be used by the interface server to associate the alert box with a certain window that represents that object. The message to display is given as the content of the tag.

The interface server responds with an empty positive reply. As mentioned, this reply only acknowledges the arrival of the message and does not signal that the user has read it. Negative replies should not be generated for an alert request.

9.2.7 The Directory Listing Exchange

The directory listing exchange is used by the interface server to access directory listings on the application host. This is necessary to let the user choose a file on the application host in an interactive file selection dialog.

The request root tag is `listdirectory`. The optional attribute `path` gives the full path name of the directory to list. If absent, the application assumes a default directory, e.g. the user's home directory or the current working directory.

The response is non-empty if positive and contains the listing of the directory formatted using XML. The root tag is `directory`. The required attribute `path` gives the full path name of the directory actually listed. The `directory` tag contains value tags that give the directory listing as a list of maps. Each map represents one directory entry. The following list defines the possible keys in such a map, of which `name` and `type` are required.

- `name` – A string giving the name of the entry, including any extensions[2].
- `type` – A string that specifies the kind of the entry. Values are `file` for a regular file, `dir` for directories, and `special` for other kinds of directory entries, e.g. Unix devices.
- `size` – An integer that gives the size of files in bytes.
- `symlink` – A boolean value. If present and true, the entry is a symbolic link (for Unix systems) or a similar reference to another entry in the file system.

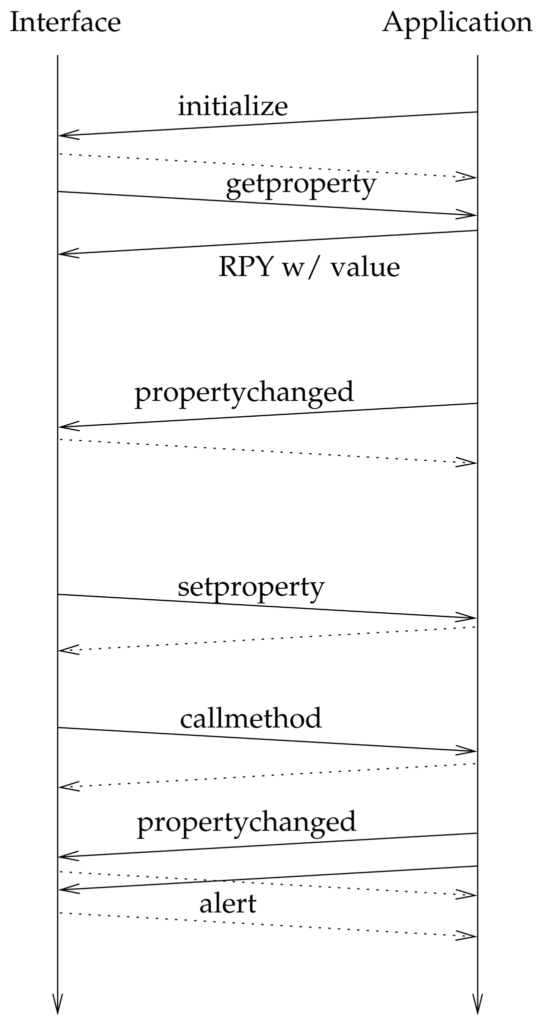
More keys may be defined in the future to transmit further data about directory entries, e.g. the access permissions.

[2]This is only mentioned because some platforms don't treat file name extensions (e.g. `.pdf`) as part of the name proper. However, these systems still use the full name for unique identification of the file.

The "`.`" and "`..`" entries must be omitted from the listing as their meaning and handling is system-dependent. If required, they can be inferred from context.

9.2.8 The Protocol Illustrated

Figure 9.1 illustrates how the various exchanges work together. The left side shows a sequence diagram, the right side details the content of the messages. The dotted lines in the diagram indicate empty positive replies required by the BEEP framework.



```
<initialize>
 <rootobject>1</rootobject>
 <interface>
...

<getproperty object="1"
 property="status" watch="true" />

<string>Initializing</string>

<propertychanged object="1"
 property="status">
 <string>Ready</string>
</propertychanged>

<setproperty object="1"
 property="limit">
 <int>7000</int>
</setproperty>

<callmethod object="1"
 method="calculate"/>

<propertychanged object="1"
 property="status">
 <string>Processing</string>
</propertychanged>

<alert>Out of memory</alert>
```

Figure 9.1: Protocol Exchanges Illustrated

After initialization, the interface starts watching the `status` property of the application root object. The reply to the `getproperty` message contains the current value. Later on, two change notifications are sent for this property. After some user input, the interface transfers a parameter value into an application property, then causes a method to be called. The application responds asynchronously, first with a property change notification and shortly thereafter with an error report.

9.3 Of Stubs and Glue

Now that the protocol has been defined, the implementation in the prototype can be discussed.

9.3.1 The Architecture

Implementing the protocol results in two pieces of code: an extension for the user interface code, making it a server, and a client library for applications to link against. Both contain a protocol engine that encodes and decodes protocol messages, controls transmission, and dispatches received messages after decoding.

The client library decodes messages and translates them into application property and method accesses. It is also responsible for assigning and tracking object ids, and for conveying property change notifications. Since all of this is invisible to the application code, the term glue code is appropriate.

The server part requires more interfacing code. The user interface engine actively accesses application objects and expects them to be available as real, live `QObjects`. That is because the machinery uses `QObject` pointers without discrimination between widgets and application objects. Thus it is the task of the protocol implementation to create and maintain stub objects that look as close to the real thing as possible.

9.3.2 The Protocol Engine

The protocol implementation is based on the beepcore-c library [2]. It implements all parts of the BEEP framework that are not specific to the actual application protocol, including session setup, channel management, and frame header parsing. The protocol engine merely needs to implement a "profile" to be used on one BEEP channel.

beepcore-c is a pure C library, and profiles are implemented as a set of callback functions. In order to enable code reuse between the interface (server) and application (client) side, the implementation elevates this callback interface into C++ with a wrapper class, `BeepProfile`. Some common handling is implemented directly in the C callbacks, but all required hooks are realized as virtual methods of that class.

The profile subclasses `ClientProfile` and `ServerProfile` are responsible for parsing completely received messages. Once a message has been decoded, its content is forwarded to an instance of `ClientConnection` or `ServerConnection`, respectively. These connection classes coordinate remote communication by mediating between the main code and the profile. This involves dispatching in both directions as well as keeping track of the object id mapping.

The connection classes are also responsible for setting up the TCP connection. The beepcore-c library expects to be passed an open socket file descriptor. While there is convenience code to set up TCP sockets in the library, it was not used because the listener doesn't fork for incoming connections. It is necessary to have one process per connection because Qt is built on the assumption of one process per application and makes deliberate use of global data. Custom socket set-up code also has the advantage of flexibility. For instance, the current code already supports IPv6 transparently, and it could be easily extended to support local Unix domain sockets or a SOCKS proxy.

Threading is another point of conflict between beepcore-c and Qt. Qt supports threading, but the widgets and the event system are not thread-safe. Processing of input events and creation of new widgets must happen in a designated main thread. In contrast, beepcore-c makes massive use of threading and in fact always delivers incoming messages in separate worker threads. Mutexes are used to protect data structures as necessary. Serialization is not sufficient for processing of property change notifications, though, because of Qt's main thread requirements. These are secured by injecting property change notifications into the main thread in the form of custom events. Since this is a common problem, Qt provides adequate support for this kind of dispatching.

9.3.3 Stub Objects

Stub objects are used to make remote application objects look like local ones. Due to the distinct properties of remote communication, this masking cannot be complete, and some special care is still necessary when working with remote objects.

Watching a property of a local object is as easy as connecting to the proper signal. The watched object never knows it is being watched. The same approach does not work for remote objects. Change notifications are only conveyed across the connection if explicitly requested. This makes it necessary for watchers to register with the watched object.

Method invocation is also best handled by a special interface. Faking Qt's class metadata and providing shadow methods is not only more difficult, but also performs worse due to the additional conversions that would take place.

This special interface for remote objects is declared in the `RemoteObject` class as follows:

Listing 9.1: Interface for Remote Objects

```
public:
  void startWatch(const char *propertyName);
  void stopWatch(const char *propertyName);
  void callMethod(const char *methodName,
    const QValueList<QVariant> &parameters);
```

Code that may access remote objects must first check the class of the object it accesses. If it inherits `RemoteObject`, that interface must be used. For method invocation this is quite straightforward: `MethodcallAction::execute` simply uses `callMethod` instead of the `invokeSlot` function. In the case of property access `startWatch` and `stopWatch` are called in addition to connecting and disconnecting the `propertyChange` signal.

`RemoteObject` is an abstract class that merely defines the registration interface. Actual stub objects use its subclass `AppObject`. This separation reduces the dependencies between the user interface core and the communication code. Each instance of `AppObject` keeps a list of watched properties and their cached values. The list also includes a reference count for each property, since a single property may be watched by several objects in the user interface.

Caching the values of properties has the advantage of instant local access. Unfortunately, the validity of cached values can only be guaranteed for properties that are being watched, i.e. for properties whose cached value is updated as changes occur because the application sends change notifications. There is no valid stored value for properties that are not watched, and for properties that just started being watched and no initial value has been received from the application yet. Read requests for such properties can be handled either by waiting for a value to arrive or by instantly returning an invalid or outdated value. The second way may seem bad at first, but it makes some sense when combined with watching. While the invalid value may cause short-term inconsistencies, those will be corrected eventually once a value is received from the application and propagated through the change-notification system. The prototype uses this second approach in the hope that it provides better responsiveness.

9.3.4 Remote File Selection

In order to browse files accessible to the application, directory listings must be transported by the protocol and integrated into the Qt file selection dialog.

Qt makes integration into the file selection dialog easy by providing infrastructure for remote access based on URLs. For remote browsing within the presented system the `suitfile:` URL scheme is used. It has the same rules as the `file:` scheme. Support for the new scheme is provided by the class `SuitFile`, a subclass of `QNetworkProtocol`. Qt allows for all kinds of operations, but the prototype only implements directory listings. Once a file is selected, its URL gets transmitted to the application for actual access. On the way, `suitfile:` URLs are translated into `file:` URLs, making them understandable by the application.

When a `SuitFile` instance receives a directory listing request, it hands it on to the `ServerConnection` object. The results are delivered asynchronously to a method of `SuitFile`. As with property change notifications, care must be taken to make this happen in the Qt main thread. `SuitFile` decodes the listing, which is transmitted as a list-of-maps data value, and generates the information objects required by Qt's network protocol infrastructure.

On the application side, directory listing requests are automatically answered by the protocol engine. It uses Qt classes to access local directory listings and extracts the relevant pieces of information for transmission.

Chapter 10

Results

"Is that what we were looking for?"
"Yes."
"Oh! Well, anyhow – it didn't rain."
Eeyore and Pooh in *Winnie-the-Pooh*

10.1 Choosing a Porting Example

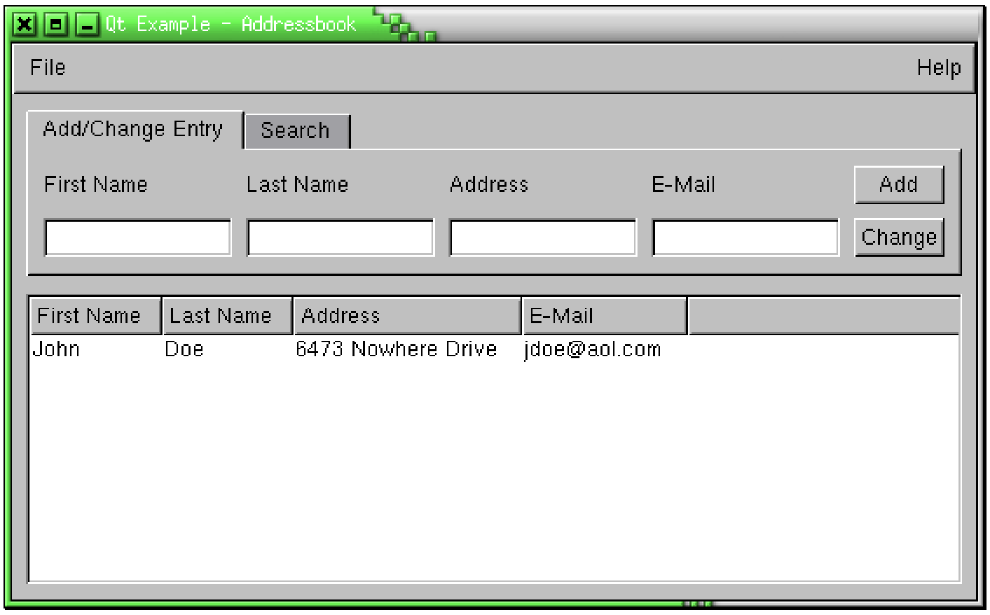
Now that the system has been described at length, it is time to evaluate what has been achieved. To give an idea, a Qt application was ported to the presented system.

The application to be ported was chosen to match the state of the prototype implementation and the time available for porting. This unfortunately means a very simple application that uses a limited set of widget classes. The Addressbook example program distributed with Qt meets these criteria well, since it still contains a complex widget (an item list with multiple columns) and allows data synchronization to be demonstrated.

The Addressbook application uses a single window. Figure 10.1 shows a screenshot of the ported version. The current list of addresses is shown in the table at the bottom. The dialog elements at the top can be used to add more entries, change existing entries, and to search for entries using substrings. Search results are highlighted in the address list. The menu can be used to save and load the address list to files.

10.2 The Porting Experience

The code of the original Addressbook application suffers from its own simplicity. There is no clean separation between data and interface. In fact, it is implemented in terms of subclassed Qt widgets, and even exploits the `QListView` widget for its main data storage.



First Name	Last Name	Address	E-Mail
John	Doe	6473 Nowhere Drive	jdoe@aol.com

Figure 10.1: Screenshot of Addressbook

As a result, the ported application shares no code with the original—it was reimplemented from scratch. It defines two classes, `BookApp` and `Entry`. A list of `Entry` objects is kept in a list property of the single `BookApp` object. As required by the abstract object model, these classes have no knowledge of the user interface and merely expose their properties and some methods.

When rewriting an ordinary application, separating the model and the user interface can create a duplication of effort. In the case of the Addressbook application, two copies of the address list must be maintained: one in the model objects and one in the list widget. This is not an issue with the system presented in this thesis, however. The list widget is maintained by generic code provided as part of the system. So, the application code itself still only maintains one copy of the list.

10.3 Performance Measurements

An attempt was made to measure the performance of the system, using the original and the ported Addressbook application. Due to the interactive nature of a graphical user interface and the lack of good testing tools for X11, that attempt was only partially successful.

10.3.1 Experiment Set-Up

A simple automated test driver creates a reproducable test parcours. It is capable of simulating mouse and keyboard events read from a command file. The test parcours consists of entering two address entries by typing into the text fields and clicking the "Add" button. Unfortunately, further actions are prohibited by the simple nature of the test driver.

Each version of Addressbook (Qt and SUIT) was measured in a "local" and a "remote" scenario, provided by two Linux PCs on a 100 MBit non-switched LAN. The scenarios are illustrated in Figure 10.2. The solid boxes represent processes and code entities. The dashed boxes represent hosts. Socket connections for inter-process communication are shown as lines below the processes.

10.3.2 Time Measurements

The first set of measurements are time benchmarks. Time is measured through instrumentation added to the customized version of Qt. Every time some text is drawn, the text is written to `stdout` along with a time stamp. This output is captured into a file for analysis. The location of time measurement is indicated in Figure 10.2 by a clock icon. Since the X11 protocol uses asynchronous pipelining, network latency is hidden

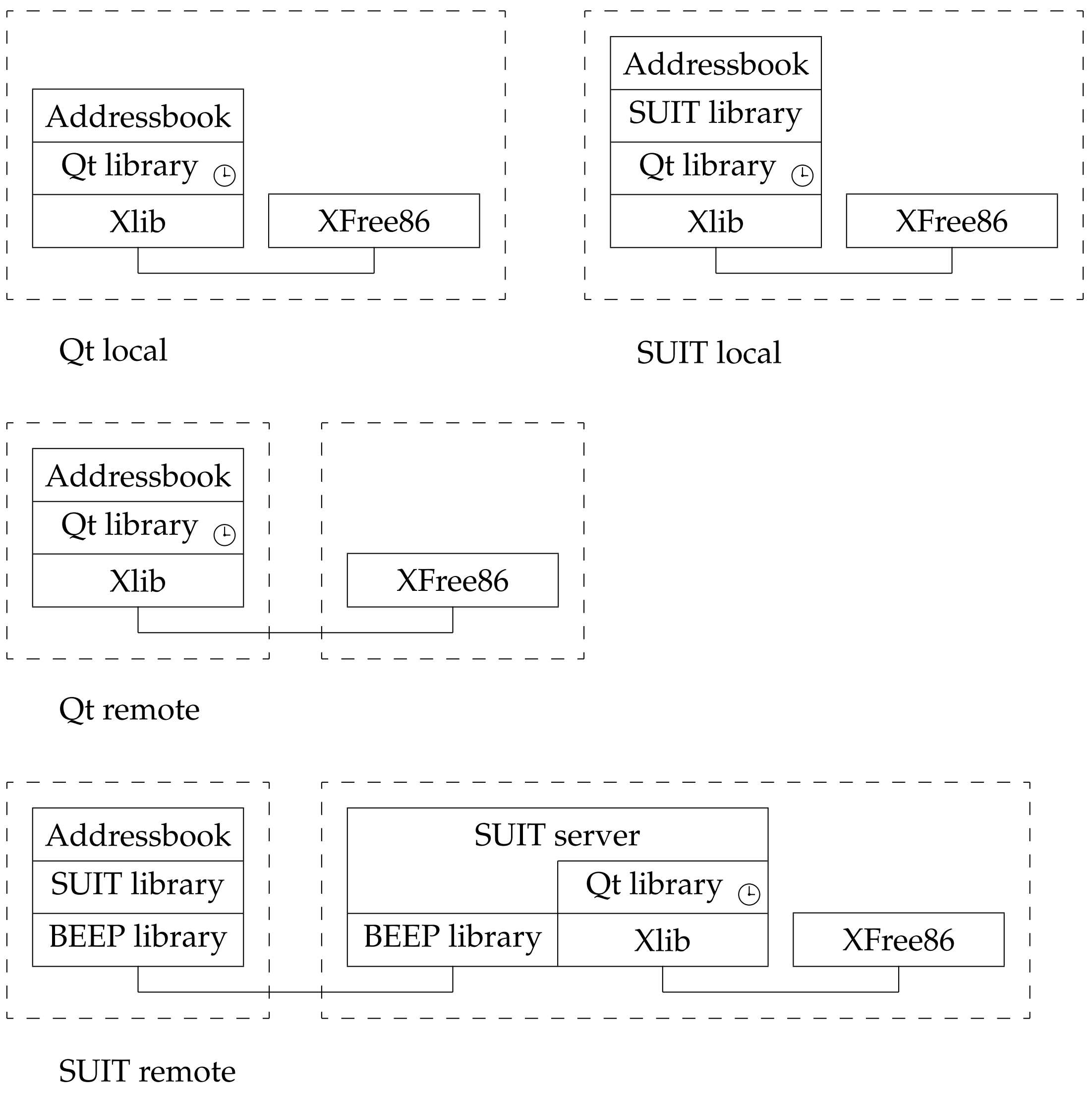


Figure 10.2: Measurement Scenarios

Scenario	Run 1	Run 2	Run 3	Mean
Qt local	574 ms	429 ms	561 ms	517 ms
SUIT local	423 ms	504 ms	467 ms	463 ms
Qt remote	352 ms	395 ms	381 ms	376 ms
SUIT remote	1907 ms	1794 ms	2720 ms	2103 ms

Table 10.1: Measured Times

from measurement in the “Qt remote” scenario, but not in the “SUIT remote” scenario. Thus these two cannot be compared in a meaningful way.

Table 10.1 lists the total time to complete the test parcours as determined from the time stamps in the test protocols. There are three runs for each scenario, with the geometric mean calculated from those.

10.3.3 Data Volume Measurements

The presented system claims to generate less network traffic during remote operation than X11 does. This was verified using the same test parcours as above. Data traffic on the host-to-host TCP connection was captured into files using the `tcpflow` tool [36]. Capturing was active only while the test parcours was run, not during application startup and termination.

Scenario	Run 1	Run 2	Run 3	Mean
Qt remote	92864 bytes	113924 bytes	103648 bytes	103119 bytes
SUIT remote	2773 bytes	2773 bytes	2773 bytes	2773 bytes

Table 10.2: Measured Data Volume

Table 10.2 lists the total amount of data captured during the testing runs, with an average computed from three consecutive runs. It is interesting to note that in the case of Qt more than 85 percent of the total traffic went from the application to the X11 server, while traffic was almost symmetric for SUIT.

10.3.4 Code Size

The total size of application source code is also of interest. Both versions of Addressbook consist of five C++ source files. The Qt version has a total of 597 lines of source code. The SUIT version measures in at 372 lines of source code, plus 330 lines of XML description.

10.3.5 Discussion

The results from the time measurements are interesting. In local mode, the presented system shows no loss in performance over a pure Qt solution. This is surprising, since the presented system adds one layer of abstraction with a new code boundary that must be crossed.

Remote mode is harder to put into perspective. For Qt, the scores are bogus—they were measured on the wrong host, leaving network communications out of the equation. The performance gain over the “Qt local” scenario can be attributed to load sharing between the two involved hosts. The presented system on the other hand has performance problems that are immediately obvious when watching the application. The likely source are threading problems within the beepcore-c library. That library is still under development and erratically exhibits similar delays and even deadlocks with an included trivial test program.

The data volume measurement is easier to interpret. In this simple scenario, not sending individual drawing commands across the wire improves efficiency by more than an order of magnitude. This even holds true when accounting for the Low Bandwidth X Extension (LBX), which can typically reduce X11 data volume by 50

The decrease in code size is less drastic, but still shows the postulated trend. With the XML description taken into account, size has actually increased. However, it can be argued that such a description is less complex than C++ code, making line counts inappropriate for comparison.

Chapter 11

Conclusion and Future Work

"Ha Ha! Amusing in a quiet way, but not really helpful."
Eeyore in *The House at Pooh Corner*

11.1 The Goals Revisited

The following sections look again at each goal to see what was actually achieved in this thesis.

11.1.1 Platform Independence

The promise of platform independence is perhaps the most obvious benefit of the presented system. Instead of re-writing the application for each platform, the application and the user interface description are written once and can be used on any supported platform.

The prototype implementation does not achieve this kind of platform independence. It is based on the Qt Free Edition and runs on Linux and Mac OS X, with the help of the X Window System. To support other platforms, the system must be explicitly ported or re-implemented.

However, the prototype is not the only result of this work. It is based on an abstract design, created with platform independence in mind. I expect that this design makes it easy to create additional implementations for other platforms. The abstract description of widgets using just nesting and properties has shown that it is capable of describing both simple and complex widgets. This claim is backed by the table classes used in the Addressbook example.

It should also be noted that once the system is ported, it can run any application built for it. This is a significant improvement over porting each application separately. This thesis does not address the issue of a platform-independent runtime system for applications. Applications using platform-specific features, e.g. BSD-style sockets, still require porting to run on other platforms. The effort is greatly reduced by exempting the user interface part from porting, though. If porting of the application code is considered too expensive, there is still the possibility for remote execution. Server-only implementations can open up new platforms for deployment when combined with remote application execution.

11.1.2 Language Independence

The same argument as for platform independence applies to language independence. The prototype uses C++ and Qt's object model. It is based on an abstract object model and a well-defined communications protocol, though, which I expect will make it easy to provide additional implementations. The real usefulness of the system cannot show until several implementations exist, so proving it is left to future work.

The separation facilitated by the protocol makes it possible to provide client-only implementations. For example, an implementation of the system for MS Windows will likely use C or C++ for the user interface server, but may coexist with a protocol implementation and language binding for Ada.

Providing an additional language binding for the abstract object model is much less effort than bridging a whole GUI toolkit API. This is due to the generic nature of the object model. Existing object models and property systems can be mapped to the abstract model if they are present in the target language. Mapping is supported by a set of fundamental data types that are designed to be independent of both programming language and machine encodings.

11.1.3 Decomposition of Application and Interface

The decomposition into application core logic and graphical user interface was done using a description language with semantic elements. It turns out there is a trade-off when drawing the line between the two parts as a result. Some tasks, from both the "view" and the "controller" domains, are so complex they cannot be built into a framework. They require application-specific code, and the critical question is where to place this functionality. Putting it on the user interface side breaks the ideal of a description without code, while putting the functionality on the application side breaks the ideal of interface-independence.

This thesis puts the problematic code on the application side, keeping with the ideal of a pure description language. Still, the application side has no direct access to the user interface and its structure. This works out because the description can always dispatch events and data for processing by the application code.

This problem aside, the port of the Addressbook example demonstrates how user interface and data model can be separated completely, with no code dedicated to the user interface. This ideal may be missed for more complex applications, but Section 3.3 demonstrates how these cases can be handled.

11.1.4 Networking

The prototype provides optional network capabilities, allowing application and user interface to execute on different systems. This was already possible with X11, but the proposed system has the potential to appeal to a wider range of developers through the use of the native toolkit on each platform. The advantage over custom-coded client/server systems lies in a reduction of coding effort, since the place of custom clients is taken by the generic interface server.

The appeal of the system critically depends on the availability of implementations for major platforms and programming languages. The use of industry standards in the protocol—BEEP and XML—makes that easier, since readily available libraries can be used to reduce implementation effort.

Efficiency as compared to X11 is good. This is considered a result of the higher level of abstraction. Where X11 repeatedly sends drawing commands, SUIT sends the data that needs to be displayed, once. Still, the efficiency of the protocol is affected by the choice of the base technologies. Both message framing and message contents are encoded as text. A binary encoding might improve efficiency and in extension performance.

The protocol is generic by purpose and thus requires only a small number of message types. Should there be grounds for redesigning the protocol (e.g. to improve coding efficiency), this is feasible without affecting the rest of the system.

11.1.5 General Applicability

In addition to the explicitly state goals this thesis also aims at a system that can be used for any application. This is of course a very broad goal, and it is hard to demonstrate that it has been achieved.

At the small scale, the system has been shown to work. The Addressbook application, albeit simple, was ported completely and with little visual and functional difference to the original.

It is largely unknown how well the system scales to larger applications. The small set of widgets supported by the prototype is insufficient to support casual applications. The widget set must be extended significantly before the suitability of the system can be demonstrated by porting such an application.

Not all kinds of applications lend themselves equally to a platform-independent toolkit. For example, professional pre-press applications require access to a platform's color matching system to accurately display images on screen. Enterprise information systems on the other hand are usually based on forms, and work well with a platform-independent GUI system.

Another drawback of a platform-independent toolkit as implemented for this thesis is the lack of support for custom or customized widgets. Applications tend to have special needs, and these are often realized by custom widgets. The only way to improve the situation is to provide more widget classes and to add new features to existing classes. But there will always be applications with needs that are not served by a pre-defined widget set.

11.1.6 Treatment of Legacy Applications

The relation to legacy applications is not listed with the original goals, but always an interesting question for new GUI systems. The most important point here is that SUIT is intended to integrate into existing environments, not replace them. It does not provide its own widget set, nor a custom look and feel.

How hard it is to port an existing application to the presented system is hard to judge in advance. A part of the task is to remove interface-driving code without leaving holes. As seen in the Addressbook example in Section 10.2, this can mean an almost complete re-write of the application. Applications that use Model–View–Controller or a similar design pattern will be easier to port, though. Automatic mapping tools can create a starting point for a SUIT description from an existing interface description. This is especially easy if the original description also uses XML (e.g. Qt Designer) and transformation engines like XSLT can be used.

11.2 Future Extensions

This section gives an overview of possibilities for future extension of the system and for further research.

11.2.1 Platform Support

Alternate implementations for platforms other than Qt/X11 are required to provide actual platform independence in the real world. The presented system gives alternate implementations many degrees of freedom—programming language, widget inheritance, execution parallelism, and data storage can all be chosen to fit the requirements of the native toolkit. Together with the use of the industry standards XML and BEEP, this makes re-implementations easy. In particular, it is expected that the presented system can be ported to a new platform in less time than a traditional API-based toolkit could be ported.

11.2.2 Language Support

The prototype supports C++ using Qt's extended object model. Support for other languages as well as for other C++-based object models is desirable. Since interface server and application environment are separate systems, language support can be extended independently. Effort required to support new languages is low, since only the abstract object model must be bridged, not an extensive GUI toolkit API.

Java is of special interest here, since a pure-Java implementation of the application environment can run on a wide range of computing platforms. A Java implementation could be based on the JavaBeans component system [17].

11.2.3 Widget Set Extensions

The current implementation only serves as a proof-of-concept and thus supports only a very limited range of widget classes. Some of the classes listed in Sections 5.3.3

through 5.3.6 were not implemented due to time constraints. Implementing these would help, but in order to make the system suitable for casual applications, even more widget classes are required. Palette windows, dockable toolbars, date and time controls, and sliders are commonplace in desktop PC user interfaces.

Since the system uses widgets from the native toolkit, adding support for a new widget boils down to mapping the abstract widget class definition to the toolkit at hand. This is quite easy in those cases where there is a one-to-one correspondence and the SUIT implementation provides adequate infrastructure. For example, the prototype supports the `PushButton` class with about 120 lines of code. More complex widgets or features may be harder to map.

11.2.4 Free-Form Drawing and 3D

Many applications could do with free-form drawing surfaces to display graphics, ranging from a business data chart to scientific visualizations of data sets. Such drawing surfaces can be integrated into the system as a widget class, but may need protocol extensions to convey the actual drawing instructions.

Drawing surfaces can be based on a number of standard technologies. Of these, OpenGL is especially promising. It supports animated graphics in both 2D and 3D, and has established itself as a cross-platform standard. OpenGL is a procedural API, so mapping it to the SUIT model is not straightforward.

There are two fundamental ways to integrate OpenGL into the system. They differ in where actual rendering happens: on the application host or on the interface server host. Which of these is preferable depends on the available resources—hardware acceleration and network bandwidth.

If rendering is done on the application host, only the finished image is transmitted for display. The interface server doesn't need to know OpenGL was used to render the image, actually. The application uses the system's OpenGL implementation to render into an off-screen buffer, then retrieves the contents of that buffer and transmits the image. It depends on the OpenGL implementation whether it is possible to get hardware-accelerated rendering without an associated on-screen graphics context.

In a very simple implementation, the image data can be transmitted as a property value, without changing the protocol. However, there are more efficient ways to transmit the image. A binary encoding that sends just the differences between images and uses on-the-fly compression can increase throuhput significantly. A separate BEEP channel can be used to set these messages apart from the standard, XML-encoded messages.

Rendering the image on the interface server host requires that OpenGL commands are transmitted across the wire. This is more involved than local off-screen rendering as it means creating stubs for all OpenGL API functions. The X11 GLX extension may provide a good starting point for both OpenGL call interception and wire encoding. Again, a separate BEEP channel may be used to avoid interference with the main protocol.

11.2.5 Semantic Concepts

The description language could be extended to encompass more features found in graphical user interfaces. These include modal dialogs and building reusable dialog components from groups of widgets. The latter mechanism will also make it easier to adapt a given user interfaces to low-resolution devices (e.g. a PalmOS PDA) without rewriting the interface description from scratch.

11.3 Retrospective

This thesis tried to combine established techniques to create a new system for developing graphical user interfaces. Each of these techniques taken by itself isn't new, but it seems the consistent combination hasn't been attempted before. The results show that it is indeed possible to retain most of the advantages of the individual aspects. At the same time, a new flexibility in application deployment is gained from the combination of platform independence and transparent networking.

Building a comprehensive GUI system is a daunting task. This has affected both the scope of the thesis as a whole and the quality of the implementation. Several ideas that were part of the original vision of the system were dropped, for example handling of modal dialogs. The widget set was scaled back and is only partially implemented. The prototype also neglects security and error handling.

But scaling back has payed off. The core concepts were implemented and demonstrated to work. Of course, the developed prototype is not a finished product, and there is still some way to go before the system is ready for widespread use.

It is unrealistic to expect existing desktop application software to be ported to the presented system. But it may prove to be a viable alternative to platform-specific GUI APIs for writing new programs, especially programs where use of platform-specific features is less important than development time and flexible deployment. Business applications, in-house tools and scientific simulations come to mind as examples. The system may also have a future as a rapid prototyping platform.

Appendix A

A DTD for the Description Language

The following XML DTD can be used for syntactic validation of user interface descriptions.

```
<!--
  suit.dtd
  Document type definition for interface descriptions.
-->


<!-- shortcut entities -->

<!ENTITY % value        "string | int | float | boolean | enum |
                         list | map | widgetref | expression" >
<!ENTITY % datatype     "string | int | float | boolean | enum |
                         list | map | object" >
<!ENTITY % action       "sendevent | assign | methodcall |
                         openwindow | closewindow | focus |
                         switch" >


<!-- main building blocks -->

<!ELEMENT widget        (widget | property | variable | clone |
                         trigger | action | watch)* >
<!ATTLIST widget
   class        CDATA   #REQUIRED
   id           CDATA   #IMPLIED
>

<!ELEMENT property      ((%value;) | sync) >
<!ATTLIST property
   name         CDATA   #REQUIRED
>
```

```
<!ELEMENT variable      ((%value;) | sync) >
<!ATTLIST variable
   name         CDATA   #REQUIRED
   type         (%datatype;)  #REQUIRED
>


<!-- value types and lookalikes -->

<!ELEMENT string        (#PCDATA) >
<!ELEMENT int           (#PCDATA) >
<!ELEMENT float         (#PCDATA) >
<!ELEMENT boolean       (#PCDATA) >
<!ELEMENT enum          (#PCDATA) >

<!ELEMENT list          (%value;)* >
<!ATTLIST list
   type         (%datatype;)  #IMPLIED
>

<!ELEMENT map           (key, (%value;))* >
<!ELEMENT key           (#PCDATA) >

<!ELEMENT widgetref     EMPTY>
<!ATTLIST widgetref
   ref          CDATA   #REQUIRED
>

<!ELEMENT expression    (#PCDATA) >

<!ELEMENT sync          EMPTY>
<!ATTLIST sync
   from         CDATA   #IMPLIED
   with         CDATA   #IMPLIED
   to           CDATA   #IMPLIED
   expression   CDATA   #IMPLIED
>


<!-- window tags -->

<!ELEMENT clone         EMPTY>
<!ATTLIST clone
   mode         (auto|manual)  #REQUIRED
   from         CDATA   #IMPLIED
   variable     CDATA   #REQUIRED
>

<!ELEMENT trigger       EMPTY>
<!ATTLIST trigger
    on          CDATA   #REQUIRED
>


<!-- event generation and handling -->

<!ELEMENT action        (%action;)* >
<!ATTLIST action
   event        CDATA   #REQUIRED
>

<!ELEMENT watch         EMPTY>
<!ATTLIST watch
   expression   CDATA   #REQUIRED
   event        CDATA   #REQUIRED
>


<!-- actions -->

<!ELEMENT sendevent     EMPTY>
<!ATTLIST sendevent
   event        CDATA   #REQUIRED
   to           CDATA   #REQUIRED
   propagate    (true|false)  "true"
>

<!ELEMENT assign        (%value;) >
<!ATTLIST assign
   to           CDATA   #REQUIRED
>

<!ELEMENT methodcall    (%value;)* >
<!ATTLIST methodcall
   object       CDATA   #REQUIRED
   method       CDATA   #REQUIRED
>

<!ELEMENT openwindow    EMPTY>
<!ATTLIST openwindow
   ref          CDATA   #REQUIRED
   with         CDATA   #IMPLIED
>

<!ELEMENT closewindow   EMPTY>
<!ATTLIST closewindow
   ref          CDATA   #IMPLIED
   with         CDATA   #IMPLIED
>

<!ELEMENT focus         EMPTY>
<!ATTLIST focus
   ref          CDATA   #REQUIRED
>

<!ELEMENT switch        (case*, else?) >
<!ATTLIST switch
   expression   CDATA   #REQUIRED
>

<!ELEMENT case          (match, (%action;)* ) >

<!ELEMENT match         (%value;)* >

<!ELEMENT else          (%action;)* >


<!-- EOF -->
```

Bibliography

[1] Abrams, Marc (editor): *User Interface Markup Language Draft Specification*. Harmonia Inc., January 2000.
http://www.uiml.org/specs/uiml2/

[2] The *beepcore-c* communication library.
http://beepcore-c.sourceforge.net/

[3] Box, Don et al.: *Simple Object Access Protocol (SOAP) 1.1*. W3C Note, May 2000.
http://www.w3.org/TR/SOAP

[4] Bray, Tim; Paoli, Jean; Sperberg-McQueen, C. M.; Maler, Eve: *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation, October 2000.
http://www.w3.org/TR/REC-xml

[5] The *Cocoa* application environment. Apple Computer, Inc.
http://developer.apple.com/cocoa/

[6] *Cocoa Developer Documentation*. Apple Computer, Inc.
http://developer.apple.com/techpubs/macosx/Cocoa/CocoaTopics.html

[7] AppleScript concepts in the *Scriptable Applications* topic of the *Cocoa Developer Documentation*. Apple Computer, Inc.
http://developer.apple.com/techpubs/macosx/Cocoa/TasksAndConcepts/ProgrammingTopics/Scriptability/index.html

[8] The *FOX* graphical user interface toolkit.
http://www.fox-toolkit.org/

[9] The *GLADE* user interface builder.
http://glade.gnome.org/

[10] The *GNOME* desktop environment.
http://www.gnome.org/

[11] Göschka, K. M.; Smeikal, R.: *Interaction Markup Language – An Open Interface for Device Independent Interaction with E-Commerce Applications*. Proceedings of the 34th Hawaii International Conference on System Sciences, Volume 7, 2001.

http://www.computer.org/proceedings/hicss/0981/volume%207/09817074abs.htm

[12] The *GTK+* graphical user interface toolkit.
http://www.gtk.org/

[13] Hamilton, Graham (editor): *The JavaBeans Specification*. Sun Microsystems, Inc., August 1997.
http://java.sun.com/products/javabeans/docs/spec.html

[14] Hodes, T. D.; Katz, R. H.: *A Document-based Framework for Internet Application Control*. 2nd USENIX Symposium on Internet Technologies & Systems (USITS'99), Boulder, CO, October 1999.
http://www.usenix.org/publications/library/proceedings/usits99/hodes.html

[15] Hyatt, Dave (editor): *The XPToolkit Architecture*. The Mozilla Organization.
http://www.mozilla.org/xpfe/xptoolkit/

[16] The *Java AWT* graphical user interface toolkit. Sun Microsystems, Inc.
http://java.sun.com/products/jdk/awt/

[17] The *JavaBeans* component architecture. Sun Microsystems, Inc.
http://java.sun.com/products/javabeans/

[18] The *Java Foundation Classes* and the *Swing* graphical user interface toolkit. Sun Microsystems, Inc.
http://java.sun.com/products/jfc/

[19] Klensin, J. (editor): *Simple Mail Transfer Protocol*. IETF RFC 2821, April 2001.
http://www.ietf.org/rfc/rfc2821.txt

[20] Krasner, G.; Pope, S.: *A Cookbook for Using Model–View–Controller User Interface Paradigm in Smalltalk-80*. Journal of Object-Oriented Programming, 1(3): 26–49, August/September 1988.

[21] Le Hors, Arnaud et al.: *Document Object Model (DOM) Level 2 Core Specification*. W3C Recommendation, November 2000.
http://www.w3.org/TR/DOM-Level-2-Core

[22] The *Mash* streaming media toolkit. The Open Mash Consortium.
http://www.openmash.org/

[23] The *Mash* software, version 5.2. The Open Mash Consortium.
http://www.openmash.org/developers/downloads.html

[24] Milne, A. A.: *Winnie-the-Pooh*. Egmont Books, London, 2000. First published in 1926.

[25] Milne, A. A.: *The House at Pooh Corner*. Egmont Books, London, 2000. First published in 1928.

[26] The *Mozilla* web browser project. The Mozilla Organization.
http://www.mozilla.org/

[27] The *Open Motif* graphical user interface toolkit. The Open Group.
http://www.opengroup.org/openmotif/

[28] Phanouriou, Constantinos: *UIML: A Device-Independent User Interface Markup Language*. PhD thesis, 2000.
http://scholar.lib.vt.edu/theses/available/etd-08122000-19510051/

[29] Postel, J.; Reynolds, J.: *File Transfer Protocol (FTP)*. IETF RFC 959, October 1985.
http://www.ietf.org/rfc/rfc0959.txt

[30] The *Qt* application toolkit. Trolltech AS.
http://www.trolltech.com/products/qt/

[31] *Qt Object Model* overview in the Qt 3.0 reference documentation. Trolltech AS.
http://doc.trolltech.com/3.0/object.html

[32] Rose, M.: *The Blocks Extensible Exchange Protocol Core*. IETF RFC 3080, March 2001.
http://www.ietf.org/rfc/rfc3080.txt

[33] Rose, M.: *Mapping the BEEP Core onto TCP*. IETF RFC 3081, March 2001.
http://www.ietf.org/rfc/rfc3081.txt

[34] The *Simple API for XML (SAX)* project.
http://www.saxproject.org/

[35] Stöttner, Harald: *A Plattform-Independent User Interface Description Language*. Technical Report 16, Johannes Kepler University Linz, Austria, 2001.
http://www.ssw.uni-linz.ac.at/Research/Reports/Report16/Report16.html

[36] The *tcpflow* network traffic recording tool.
http://www.circlemud.org/~jelson/software/tcpflow/

[37] Winer, Dave: *XML-RPC Specification*.
http://www.xml-rpc.org/spec

[38] Wong, Alexander Ya-li; Seltzer, Margo: *Operating System Support for Multi-User, Remote, Graphical Interaction*. Proceedings of the 2000 USENIX Annual Technical Conference, pp. 183–197.
http://www.usenix.org/publications/library/proceedings/usenix2000/general/wong.html

[39] The *wxWindows* graphical user interface toolkit.
http://www.wxwindows.org/

[40] The *XPCOM* component architecture. The Mozilla Organization.
http://www.mozilla.org/projects/xpcom/

[41] The *X Window System*. The Open Group.
http://www.x.org/

[42] *X Window System Specifications*. The Open Group.
ftp://ftp.x.org/pub/R6.6/xc/doc/hardcopy/

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfaßt und nur die angegebenen Quellen benutzt zu haben.

____

(Christoph Pfisterer)