Multi-threaded User Interfaces in Java

Ph.D. Thesis

Dipl.-Math. Elmar Ludwig

University of Osnabrück
Department of Computer Science

May 2006

Contents

List of Figures

Chapter 1

Introduction

One of the most promising aspects of the Java programming language has always been the fact that it became easy to create portable graphical applications with it. Though the first release was not yet fully usable in this regard (mainly due to inflexible event handling and portability issues) and suffered from a very limited set of user interface components, the situation improved dramatically with the release of the Java Development Kit (JDK) 1.2 and especially the *Swing* toolkit.

Today's most widely used graphical user interface (GUI) toolkit for Java remains the *Swing* API, which originated at Netscape as part of the *Java Foundation Classes* project and was added to the official Java distribution with JDK 1.2. It helped to overcome many of the limitations and problems present in the formerly used *Abstract Windowing Toolkit* (AWT) implementation (this was the only graphics toolkit available until then). In this respect, the release of JDK 1.2 and the Swing toolkit that came with it can generally be considered a success, and made GUI programming in Java a lot easier to manage.

But there is another aspect of user interface programming that is often overlooked: There is a growing trend during the past years to no longer accept – even temporarily – unresponsive user interfaces. In the first years of the *World Wide Web*, people were happy when a browser rendered a requested web page at all. Consequently, the first web browsers (quite naturally) choose the easy way of loading a page completely before displaying any content to the user. Networks were slower than today and users were used to the fact that network operations take time.

This situation changed dramatically, however, with the first release of the *Netscape Navigator* application. It featured what is known today as "incremental rendering": Updating the application window to continously reflect the amount of data already received from the server. It looked almost revolutionary at the time, and improved the user's perception of the application's responsiveness dramatically. Netscape was an instant success. Now, more than a decade later, users often ex-

pect operations that take time to not block the program's user interface, even for applications that have nothing to do with networking.

Today, incremental display has become more and more commonplace: all web browsers have it (ever since the days of Netscape), many image viewers, file managers, even some text editors do incremental loading of long documents. But to do this, there needs to be some way to run the task that loads or generates the content to display in parallel with the program's user interface. They should not block each other without a good reason to do so. And the only available answer to this problem in Java is: multi-threading.

1.1 Overview

This thesis will look the question of how this requirement for improved application responsiveness is reflected in the general structure of an application, how well the common user interface toolkits for Java can deal with this, and what could be done to improve on this.

The old Abstract Windowing Toolkit, which has been part of Java since the early days of JDK 1.0, in fact initially supported thread-safe access to GUI components. However, the newer Swing toolkit does not, and the same is true for most of the other graphics toolkits that emerged for Java over time (the currently most widely used third-party toolkit is the *Standard Widget Toolkit* that was developed as part of the Eclipse IDE project).

Notably, support for flexible multi-threaded programming is sorely lacking in the widely used user interface libraries for Java. In particular, the rather natural idea of assigning the responsibility for managing logically separate parts of a program's user interface to separate threads is not easy to tackle, if at all. As a result, the program's responsiveness depends directly on the amount of effort invested by the programmer to work around this.

Just imagine working today with a web browser that blocks all forms of user interaction with any of its windows while a new web page is being loaded in a completely unrelated window. Working with such a program would be nearly unbearable, yet even today's user interface toolkits for Java are still designed with such a usage model in mind. If a program is supposed to actually be responsive while long-running tasks are performed "in the background", a lot of extra care and effort needs to be invested on the side of the programmer.

Some of these problems are inherent in the architecture of these toolkits (like using a single dedicated event thread that should *never* be blocked), others can be worked around with some extra effort by the programmer, e.g. by routing any

updates to graphical components through the event thread. All of this will be discussed in more detail later on.

1.2 Objectives

This thesis will present a different approach to access GUI components from Java in a way that is inherently thread-safe, easily portable and yet very efficient due to the fact that user interface controls can be implemented in native C or C++ code. It is based on the idea of separating the actual implementation of the controls from the Java based application programming interface (API) for accessing these controls and using a message-based model for communicating between user threads and the GUI components themselves.

This implies both encapsulating method calls directed to the controls (commonly referred to as *requests*) as well as encapsulating the flow of events. The communication between the different threads is realized by using buffered channels (inter-thread message passing mechanisms), a concept that originated in C.A.R. Hoare's famous paper on *Communicating Sequential Processes* [12] (later extended by Roscoe in [22]) and has already been successfully applied for example in the *Alef* programming language [28] on the *Plan 9* operating system as well as in *Limbo* for the *Inferno* operating system (as described in [13] and [6]).

This nicely solves the problem of multi-threaded access by implicitly serializing these messages (which is generally required by the underlying native GUI library used by Java), allowing concurrent access to the visual components from any thread. By passing events from the user interface to the application via channels as well, events can be forwarded between threads and there is no longer any notion of a single event handler thread. Each thread can independently listen to events from controls assigned to this thread.

Moreover, this approach allows a separation between the Java program and the user interface library into two separate processes, effectively resulting in a toolkit "server" process. If the messages can be transported across a network, the server component can even be run on a completely different system, e.g. running the application itself on a Windows machine and displaying the user interface on an Apple Macintosh. This separation also makes it possible to easily replace the underlying GUI toolkit – without recompiling the application – by simply running it with a different toolkit server component for the program.[1]

To demonstrate that this approach works and to evaluate the performance impact, this thesis also includes a prototype implementation based on the GTK toolkit

[1] Though it cannot be changed while the program is running.

library, which is a native GUI library implemented in the C programming language and part of the Gnome desktop project. This is discussed in more detail in chapter 5. While this prototype implementation is mostly based on the GTK toolkit, the choice is really arbitrary: The general approach is suitable for other toolkit libraries as well. As a (somewhat extreme) example, an alternative backend is also included that is built in Java itself, based on the Swing components.

1.3 Structure of the Thesis

The compact summary given here provides a short overview of the chapters that follow:

The next chapter starts with an introduction to the general concepts of event handling in graphical user interfaces, focused primarily on Java, but largely applicable to other programming languages as well. It explains the terminology used throughout the later parts of the thesis, describes what sets graphical applications apart from traditional command line programs and presents the problems that arise from this, with a special focus on aspects related to multi-threading.

Chapter 3 presents the broader context for the work described in this thesis: A survey and critical assessment of many different existing graphical toolkit libraries and their approach to (or in some cases, their failure to approach) the issues brought up in chapter 2. Again, it focuses on the various toolkit libraries designed for the Java platform, since all of these have to deal with similar requirements and premises. However, toolkits developed for other systems like Microsoft Windows and Lucent's Inferno system have also been included.

Following this, chapter 4 contains an analysis of the issues presented in chapter 2 and discusses possible ways to improve upon the limitations seemingly inherent in the way that the traditional graphical component libraries organize the event handling process. Here, a message-based approach to event handling and request processing in general is proposed and it is shown how this can solve the issues raised.

The next chapter describes in detail the prototype implementation of the approach presented in chapter 4. It also discusses implementation considerations and their consequences where appropriate (most of the time there simply is not just "one true way" to solve a problem).

This is followed in chapter 6 by a critical evaluation of the message-based approach itself, its applicability in a larger software system, the requirements it imposes on the underlying implementation and the limitations and tradeoffs involved, especially concerning performance. A performance comparison for a simplified test

case is included here as well, though not every aspect can be measured objectively, in particular when areas of usability like perceived responsiveness or ease of implementation are considered.

Chapter 7 is a collection of some ideas for possible further improvements, both upon the design of the message-based approach and especially the implementation of the prototype presented in chapter 5. In this context, it discusses different ways to further enhance the performance and usability of the prototype.

The thesis ends with a concise summary of the important contributions and a short conclusion and list of acknowledgements in chapter 8.

Chapter 2

Background and Motivation

This chapter starts by giving some background and context for the work presented in this thesis, focusing in particular on the topics of user interface toolkit design, general concepts for event handling mechanisms and threading issues in graphical applications. Many of the concepts introduced here will be touched upon in more detail in the following chapters, but it is necessary to get a basic understanding first. Although – as already stated in the introduction – the primary interest is in Java based toolkits, most of what appears in this chapter is in fact relevant to graphical toolkits in general, independent of the particular programming language and underlying platform involved.

2.1 Concepts of User Interface Toolkits

The following sections provide an overview of the terminology and the fundamental concepts used across all user interface toolkits, namely the idea of components, events and event handling. A general understanding of object-oriented programming concepts is assumed here (interfaces, classes, objects, methods, inheritance).

2.1.1 Graphical Components

To facilitate code reuse, it has become commonplace to use standard component libraries to construct a program's user interface. These libraries are often provided as part of the desktop environment (e.g. Microsoft Windows, BeOS, Unix/X11) or a particular programming language runtime, as is the case with in Java or Tcl/Tk. Sometimes, an application also contains specialized interface components created for use in just that application, but this is becoming less common nowadays, mostly

because more and more functionality is integrated into the toolkits in the form of ready to use components.

These reusable, basic building blocks that make up any user interface are called *components*. Some other terms for these are also in use on specific platforms: The Microsoft Windows API calls these objects *controls* to indicate that the user utilizes them to interact with the application. On the X Window System the term *widget* is in common use (it is said to be derived from "window gadget"), mainly due to historical reasons[1]. All of these refer to the same principle: A configurable user interface element designed for some kind of interaction with the user. In this thesis, the term "component" will be used throughout, since this name is universally understood, and it is also the name commonly found in Java.

It is generally considered a bad idea to invent one's own GUI components when similar ones already exist in a readily available library on the platform. This is to enforce a common look and feel across applications as well as a common behaviour across the desktop: There are so-called "style guides" that define how user interface components should behave. To make the components are widely usable as possible, it is desirable that they are highly configurable, both in visual appearance (including content, text and background colours, text layout, images etc.) and their behaviour on user interaction. This refers to what happens when clicking, dragging or editing the component.

Figure 2.1 on the following page shows a typical collection of user interface component classes and their relationship in the class hierarchy. This example actually shows just a few selected classes taken from the *GTK toolkit* library (described in detail in [3]) for illustration. The basic component class of this library is `GtkWidget`, the base class `GObject` visible at the top of the class diagram is the (non-graphical) root class of the GTK object system.

Among the component classes there is a special class of components that acts as a container for other components – both visually and conceptually, thus enabling many components to share the visual space allocated to the container they live in. These are generally known as *panels* or *containers* in Java, though the terminology may vary in other toolkits. Since a container can hold other containers as well, containers are used to tie the individual components of the user interface together in the form of a *component hierarchy*. As a result, the structure of the user interface of a program can be described as a number of component trees, one for each application window.

[1] Xt, the first toolkit on the X Window System back in 1987, used this term

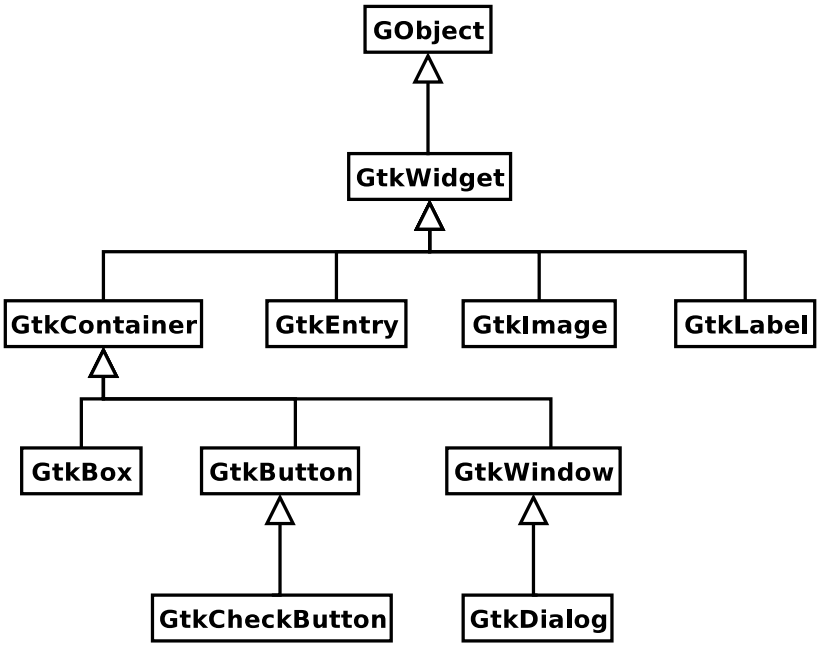

Figure 2.1: Excerpt from the GTK toolkit class hierarchy

2.1.2 Windows and Dialogs

A window is actually just a special case of a graphical container: One that has been assigned a specific area of the physical screen for display. It is therefore always necessary for a graphical application to have at least on window (or maybe one of its more specialized subclasses) to hold the other user interface components. A component generally needs to be inside a window's component hierarchy to be visible on the screen[2].

Windows are also special in that they cannot be placed inside other windows, which is why objects like windows and dialogs are referred to as "top-level" containers in Java: Each window represents the root object of its own component hierarchy. Furthermore, these hierarchies are disjoint: Any given component cannot be present in two windows at the same time.

Again, a dialog is a special kind of window that is designed to be short-lived and is used to either gather information from the user or to provide additional information about the contents of one of the application's main windows. Dialogs are typically attached to another window to indicate this relationship and will then for example be minimized together with the window they belong to.

[2]Early toolkits for the X window system had the concept of each component itself also being a window, so isolated components were indeed possible, but this idea has not caught on and is no longer found in newer toolkit libraries.

Due to the short-lived nature of a dialog, it is sometimes used as a so-called *modal* dialog, which means that all user interaction with the program is focused on and restricted to this dialog while the dialog is on the screen. Modal dialogs are often used when the program cannot usefully proceed without further information from the user, for example when opening a document in a word processor it is necessary to select a file before anything useful can be done in the document window. A thorough discussion of the usability aspects of modal dialogs is beyond the scope of this thesis, however. More about that topic can be found in [10].

2.1.3 Events

Traditional command line programs are mostly input driven: They read data from a file, socket or other input source(s), process this data in some way and produce output to a console, file, or other destination. This is all fine when there is only one possible source of input at a time, like when reading text from a file or from a user typing at the console, but this model is not applicable to the way user interaction works in a visual environment:

Here, the user has many different ways of interacting with the program via the components described in the previous sections. Such components can range from simple buttons and menus to complex objects like tree views of structured data, and there is most of the time very little restriction on what the user may do next. As a consequence, the program effectively has to "monitor" all possible sources of user interaction at the same time. This transforms the fundamental structure of a program into a model where it merely *reacts* to what the user does.

Any such kind of interaction by the user with the program is called an *event*. When looking at the program from this perspective, practically all of the actions which the program can perform are implemented in the form of *event handlers*, that is code fragments which are executed in response to an event that occurred. The most common examples of events are: A key is pressed or released on the keyboard, the mouse has been moved, one of the mouse buttons has been pressed (possibly on a component) or even some combination of these like dragging the mouse with a pressed button for scrolling. Note that not all events are user-initiated, however: A program may choose to generate events itself, which is sometimes used for simple inter-process communication among different applications on the same screen, and software timers can be used to automatically generate events at regular intervals.

At the level of the user interface toolkits, events are generally represented as individual objects, where the different kinds of events carry different information. There is a broad distinction between two categories of events:

- "primitive events"

Such events generally reflect some kind of physical action by the user like key presses or mouse movements. They merely tell that an action has occurred and carry no semantic information about the actual intention of the user, e.g. whether a click was designed to select a piece of text or initiate some other action. So the same kind of primitive event can be interpreted differently for different components.

- "semantic events"

  A component can decide to associate a particular meaning with a specific sequence of events. For example, a list box may interpret a click onto an item as a "selection" event, while or a button will recognize the sequence of mouse click and mouse release (while the mouse pointer is still on the button) as an "action" event. Such events that are generated by a component itself are called *semantic events*. In fact, most of the events interesting to an application programmer fall into this category.

2.1.4 Event Handlers

As described above, events are essentially an asynchronous way for the user interface components to signal to the application that something noteworthy has occurred. Deciding what (if anything) to do in response to this information is then the job of the application, and the parts of the program that do this are called the *event handlers*. On a conceptual level, an event handler is simply an application of the *Observer* design pattern described in [7].

Thus, a typical graphical application consists of two parts:

- A main program that creates and sets up the initial state of the user interface and binds the program's event handlers to the appropriate components in the interface. The last action done by the main program is typically to start the main event loop (if this is explicit in the code), at which point the user is free to interact with the program.
- A set of event handlers, which are invoked at the appropriate time in response to user actions. This is by far the largest part of the application, since nearly all actions that a program will do fall into this category.

Event handlers generally do not have result values and the order in which multiple event handlers are called that are registered for the same event type on the same component should not be relied on. The point where there are the greatest differences between different user interface toolkits is the design decision of which type signatures to use for event handlers. As is discussed in [7], the *Observer* pattern can generally be implemented in a *push* vs. a *pull* model: In the *push* model the observed object (the "subject") sends all its relevant state to the observer upon a

notification, regardless of whether this information is actually needed or not. In the *pull* model the observer only gets the reference to the observed object and has to request ("pull") the specific information it needs from the subject.

Both of these strategies have their advantages and disadvantages, of course, and both are in real use. Java and GTK are examples of the first strategy, which is visible from the fact that the type signatures of the event handlers are always very specific to the kind of event they are intended to handle: A handler for mouse events will get the current mouse position for example, whereas a handler for list selection events will be passed the index of the selected item in the list (which is actually state information forwarded from the list model). On the other hand, Mac OS X is an example of the second strategy, where the only information passed to any kind of event handler is a reference to the sending object (the source of the event). If further information is required, it needs to be queried on demand.

Using a more specific type signature for the event handler allows all the state of the sending object to be transmitted easily, but at the cost of making the handler less reusable in other places of the code. Imagine for example the case of closing a dialog window in response to either pressing a button inside the dialog, closing the window via the window manager (a window manager handles the window placement and decoration on the desktop and normally provides a close button as part of the window border) or pressing the *Escape* key on the keyboard. In this simple example, three different kinds of events are involved: A mouse event, a window event and a key event.

When using a push model with specific parameters for each event handler (like it is done in Java for example), it is impossible to write a single handler for all of these because of type conflicts: Three different methods containing essentially the same code would have to implemented. To avoid this code duplication, one can of course move the code to close the dialog to a separate method and simply call this from each of the handlers, but the basic problem remains. So, one has to find a balance between too many or too specific type signatures for event handlers and the minimal amount (one argument, either the event object itself or the sending object).

Finally, there is yet another aspect to consider: If the application is designed according to the *Model-View-Controller* pattern, most of the state information relevant to the event handler (the controller part in MVC) should not be known to the GUI component that is sending the event anyway and instead needs to be queried from the model part.

2.1.5 Main Loops

So, what is the application actually doing while it is waiting for the user to trigger the next event? To understand this, a more detailed look at the event handling process is needed:

Events are generated at the user interface level, either through the windowing system or sometimes by the components themselves, and are then placed inside the application's *event queue*, a toolkit specific data structure that contains a list of all the events that still need to be handled by the program. At the core of each application resides a central piece of code – the application's *main event loop* – whose sole purpose is to monitor this event queue for any new, unhandled events. Its job is to classify these events if appropriate and to dispatches each event in turn to the corresponding event handler(s) registered for this event type. Due to the fact that this code is fairly generic, many user interface toolkits choose to either hide this event loop completely from the user (Java does this for example) or they include a complete implementation as part of the toolkit library. Most MS Windows and Unix/X11 toolkits do this, as well as the Cocoa Application Kit library on Mac OS X[3].

It is important to remember that since events are always processed sequentially by the main loop and handling an event can take an unpredictable amount of time, a number of still unhandled events can pile up in the event queue. This should be avoided however, because it leads to the effect that the user interface "lags behind" in terms of user perception. So the rule of thumb is that event handlers should always finish quickly in order to not delay the processing of further events.

Some toolkits also allow the programmer to hook pieces of code into the main loop, which are then executed whenever there are currently no events to process. The typical structure of such a main loop is illustrated using pseudo-code notation in figure 2.2 on the next page. The function `DispatchEvent()` would be responsible here for identifying which component a particular event originated from, fetching the list of registered event handlers for the given type of event from the component and calling each handler found in this way in turn.

2.2 Threads and Synchronization

Threads in the context of application programming refer to independent flows of control inside one process. Like multiple processes than can run simultaneously on one computer, multiple threads can run in parallel inside one instance of the Java virtual machine. Each thread can execute its own code, but all threads share the

[3] The Mac OS X API uses the term "Run Loop" for this concept.

```
EventQueue queue;
Event event;

while (1) {
    while (IsEventPending(queue)) {
        event = GetNextEvent(queue);
        DispatchEvent(event);
    }
}
```

Figure 2.2: Typical structure of a main loop (pseudo code)

same process memory, that is they can access data from and share data with other threads in the same process. The idea behind this is that independent operations can often be performed in parallel (which can also be more efficient on the proper hardware), and the general program structure may be simpler.

The Java programming language has had the concept of threads and the corresponding synchronization primitives integrated as a core language feature right from the first public release. However, Java is portable to a wide range of platforms and devices, not all of which can support native threading, where native here means "provided by the operating system that the Java virtual machine runs upon" (this is also known as *preemptive threading*). If there is no thread support at this level, the Java virtual machine will provide its own simulated thread support (also known as *cooperative threading* or "green threads" in Java). This is mostly transparent to the application, apart from the fact that in the latter case, thread scheduling only takes place during a few, selected operations like *yield*, *wait* and *synchronized*. Another restriction of *cooperative threading* needs to be considered when using the Java Native Interface [14]: If one thread executes a native method, all other threads will be blocked until the native method returns.

It is important to realize that working with multiple threads on common data without proper synchronization is asking for trouble: If a data object is examined by one thread while another thread is modifying it at the same time, the information seen by the examining thread is non-deterministic and therefore undefined. Similar problems ensue when multiple threads simultaneously try to modify the same data object. The only general solution to this problem is to serialize the access to the data (i.e. prevent concurrent access), and this is done using *thread synchronization*: Defining checkpoints in the code that are used to delay execution of a thread until a specific condition is true.

Thread synchronization in Java is handled typically by using the so-called *Monitors*[4] provided as part of the language syntax: A monitor is an arbitrary Java

[4] These are also known as *Locks* to emphasize the mutual exclusion behaviour.

object controlling access to a critical region of code that may only be executed by one thread at a time. Various other synchronization primitives fulfilling the same purpose can be emulated in Java using Monitors, notably *Semaphores* (originally described by Edsger Dijkstra in [5]) and *Channels* as proposed in CSP [12], which are used extensively in the approach described in this thesis.

2.3 Threads in User Interfaces

When working with GUI toolkits for different programming languages, it becomes apparent that most try to avoid dealing with multi-threading in the first place. Even in case of programming languages like Java that intrinsically support the notion of threads (or parallel processing in general, for that matter) it is still relatively uncommon to find a design for a user interface toolkit that supports a decent degree of multi-threading. So one has to ask: Why is this? What is it that makes this problem hard? There are several answers to this question that will be discussed in this section.

- There is no native cross-platform thread API

  The majority of the GUI toolkits available are implemented in C or C++, including the toolkits accessible from other programming languages like Java, Perl, Python or C#. Alas, there still is no cross-platform thread API for C/C++ yet, which makes it difficult to support multi-threading in a portable GUI toolkit. There is now a POSIX thread API, but this is not yet available on Windows[5] and only very recently on Mac OS X, both of which have their own native threading APIs. Furthermore, many of the standard system libraries are not thread-safe, which in turn makes it difficult to develop a truly thread-safe native GUI toolkit.

- Multi-threaded programming is hard

  Any program that involves the use of multiple threads of execution to solve a given task will have to deal with the problems of interaction between the threads and with their access to common data. Since the flow of execution is no longer linear, the program behaviour may no longer be deterministic, which may be the cause of subtle bugs or misbehaviour. All programming languages that support multi-threading also provide some kind of synchronization primitives (locks, semaphores, mutexes, channels etc.) that can be used to synchronize the execution of the different threads as described before. This is most often used to control access to data shared between the threads.

[5] There is actually a project that tries to implement POSIX threads on top of the Win32 API.

But applying these mechanisms correctly is hard, and even synchronization examples in text books on the matter get this wrong sometimes. This led to the observation by John Ousterhout (the designer of the *Tk Toolkit* for the Tcl scripting language) that "Threads Are A Bad Idea (for most purposes)" in his 1996 USENIX presentation [21], where he proposes to avoid the use of concurrency as much as possible (in favour of events) and simply try to live with the restrictions placed on the event handling by using a single event loop. He concludes with the remark that "Threads [are] much harder to program than events; for experts only.". Or, to quote two of the designers of the *Swing* toolkit for Java:

> Component developers do not have to have an in-depth understanding of threads programming: Toolkits like ViewPoint and Trestle, in which all components must fully support multi-threaded access, can be difficult to extend, particularly for developers who are not expert at threads programming. Many of the toolkits developed more recently, such as SubArctic and IFC, have designs similar to Swing's.
>
> – Hans Muller and Kathy Walrath in "Threads and Swing" [19]

So it is only logical to want to avoid these pitfalls whenever possible. Many problems that can be solved more elegantly using multiple threads can also be solved using just a single thread if the programming language supports some form of asynchronous I/O, which would allow the program to treat pending input as just another kind of event. Interestingly, this has long been a problem for Java, which did not support the idea of I/O multiplexing (equivalent to the `select()` system call on Unix or `WaitForMultipleObjects()` on Win32) or asynchronous I/O before the inclusion of the *java.nio* package in Java 1.4.

- Multi-threaded programming involves a performance tradeoff

  A graphical component library cannot easily detect whether a program is multi-threaded or not, so if it supports multi-threaded access at all, it has to assume that all method calls have to be correctly synchronized, unless there is a way for the programmer to explicitly indicate the use of threads to the library. As a consequence, this introduces some extra overhead for a thread-safe component library even for programs not using multiple threads at all.

  Designing a GUI toolkit for multi-threading therefore always involves facing this tradeoff between performance and ease of programming. And if the programming language and the system libraries do not already support multi-threading, this tradeoff is decided in favour of performance more often than

not. Again, this has been one of the design decisions that led to the fact that *Swing* does not support concurrent access to its GUI components, yet in this case despite the fact that Java provides a well-designed thread API to the programmer.

2.4 Motivation

The next topic that will be looked at here is the amount of limitations placed on an application by a lack of thread support in the user interface toolkit and what can be improved in this area to deliver a better user experience as well as improved flexibility to the programmer.

Traditionally, the general rule is that an application's event handlers are executed synchronously, which means that no new events are dispatched while an event handler is still running. This makes programming event handlers easier, since there is no need to e.g. make them reentrant. Also, all event handlers will typically be run in the same (dedicated) thread, if the program is multi-threaded at all (this is explained in more detail in the next chapter starting on page 25). As a consequence, any event handler will "block" the program's execution for the time it is running, making the user interface unresponsive during this time. This is not a problem when each event handler does not need to do a lot of work, but this is not always the case and generally cannot be guaranteed. Some actions like loading complex documents – possibly even across a slow network connection – simply will take time. So there has to be some way to keep the user interface responsive while still supporting such longer running event handlers.

Another important point is that many of today's programs are modelled according to the so-called "multi-document architecture" that allows the user to manipulate several different documents (like images, text files, web pages etc.) in separate windows or possibly sub-windows of the same application. A natural assumption to make here would be that user interactions with separate documents in the same application do not interfere with each other, even while performing a long-running operation on one of the program's documents. One example of this would be running a complex filter on an image in an image editing application like Photoshop. The user may want to work on a second document while a filter is running, or even run separate filters at the same time on different documents.

When looking at the above model for event handling it becomes immediately clear that this is not going to work in such a way: If there is only one main loop and all events are processed synchronously, running a filter will block the program completely until this filter run is completed. In particular, the user interface would appear to be "frozen" (i.e. completely unresponsive to the user), affecting all of

the program's windows.

Such behaviour is of course not acceptable to the user, especially given that there would not even be a way to abort the operation because no new event to initiate that could be processed either while the event handler is still running. So in practice, one of two approaches is used to avoid this effect:

- Push long-running actions into idle handlers

  One common way to avoid blocking the event thread is to defer any long-running actions to so called *idle handlers* or use *timers* to execute them completely asynchronously. An idle handler is a function that is automatically called by the main event loop whenever there are currently no events to process (just like an event handler, but without an event). The general idea here is that the time which the program is spending while waiting for user input can be used for performing some "background" tasks, without creating a new thread for this purpose. Supporting such a model requires a slight modification to the standard main event loop:

  – If there are pending user events waiting to be processed or the list of idle handlers is empty, the event handling is performed normally. The corresponding event handlers will be invoked for all pending events.

  – Otherwise, the program will invoke each registered idle handler in turn instead of immediately waiting for the next event to arrive. As no extra threads are involved here, all idle handlers will run in the same thread as the main event loop, and event processing will continue only after all handlers have returned, and an ill-behaved (i.e. long-running) idle handler can block the event loop just like a normal event handler.

  The intended usage pattern for an idle handler therefore is this: Divide the work to be done in response to an event into very small steps and let each call to the idle handler perform a single one of those steps. In this way, the program can stay responsive while still processing the steps one after the other as long as no events arrive. It is quite obvious that in this model the idle handler must have some implicit knowledge of the progress of the operation, and the work must be dividable into "small" (in the sense of program run time) steps in the first place.

  Although idle handlers are a nice idea to avoid blocking the program's user interface for longer periods of time, they fundamentally share many of the same problems as the event handlers, in particular that idle handlers are executed synchronously on the event thread. This means that they must finish each invocation in a reasonably short time frame, otherwise they will block further event processing (along with other idle handlers) just like long-running event handlers would do. And in addition to that, the programmer

has to make sure that no conflicting operations are initiated while the idle handler is still active, like – in the example cited above – running a second filter on an image on which a filter is already running.

- Run the code of the action in a separate thread

  Another way is to run the action performed by the event handler in a new thread started from the event handler, so that (again) the handler invocation finishes quickly and the event thread is ready to process the next pending event. This new thread will run in parallel and obviously will not block the program's user interface, though may require some extra logic inside the application in case that a previous asynchronous operation must complete before a new one is started (like discussed for the filters). As this introduces threading into the user interface, it means that some form of locking needs to be applied inside the application.

  This model allows the work triggered by the event to proceed despite the fact that the event handler has finished. Alas, this is likely to cause yet another problem: If the toolkit is not thread-safe, accessing the user interface components from the background thread is generally not allowed, so while the new thread may calculate some result, it cannot directly display this to the user. In order to work around this restriction, there needs to be a way to pass method calls to the event thread which are then executed from the main loop.

In summary, both of these strategies are not without problems, as they will constrain the programmer in a significant way, like requiring the operation to be chopped up into small chunks of work or going through hoops just for updating a single user interface component. Yet the question remains: what can be done to improve this? The answer to this question is actually not that surprising:

If blocking the event loop is the cause of the responsiveness problems, one could try to decouple the user interface from the event loop, putting the task that is responsible for updating the components into its own thread. Then, blocking the main loop will not have such immediate negative consequences for the application's responsiveness. Putting the main loop into its own thread of course will require that more than one thread is allowed to interact with the GUI components, but doing so has the additional benefit that spawning new threads from event handlers suddenly becomes much less of a problem as well, this may be worthwhile.

Another interesting question in this context is how main loops and threads correlate. As described, a main loop is designed to process an incoming stream of events in a sequential way. Can an application have more than one main loop, and if so, can it benefit from this fact? It is obvious that any given thread can only run one main loop at a time, so it requires the use of multiple threads to use several

main loops concurrently.

One interesting use case here would be to use different main loops for event processing in distinct areas of the application's user interface (like separate windows for example). For this to be feasible, however, it would need to be possible to separate the stream of events into one stream directed at each main loop. So, unless there is some infrastructure in place to tell either the events or the components generating these events which main loop is responsible for handling them, the application will not gain anything. What is needed therefore is a way to tell e.g. one component (or maybe set this on a per window basis) that all events or specific types of events should be directed to a particular main loop.

The general approach presented later on in chapter 4 of this thesis is able to support all of these ideas.

Chapter 3

Related Work

There have been a number of different approaches in the past for making various graphical toolkits accessible from Java, each with its own goals and constraints. This chapter will present the most important of them and evaluate for each one how the event handling is done and in which way support for multiple threads is included (if at all). As already mentioned in the introduction to the previous chapter, the fundamental concepts behind graphical toolkits, event handling and multi-threading are not in any way specific to Java, so this chapter will also look beyond Java and include some non-Java toolkits as well, both for comparison and inspiration. While certainly not all ideas will be applicable to a Java based toolkit, it is nonetheless interesting to see how other systems approach the same fundamental problems. Still, the main focus of this chapter will be toolkits related to Java – either implemented in Java or otherwise accessible from Java – and it will discuss only some of the more widely used non-Java solutions.

The chapter will start with some general observations and then give a short summary of the main features and design goals of each toolkit in turn, followed by a critical evaluation of how well each one can handle the problems detailed in the previous chapter. Note that none of the toolkits described here can support location transparency for the display components (i.e. displaying an application's user interface on a different system from the one that the application runs on), beyond the usual remote display capabilities of the X window system.

3.1 Graphics Toolkits for Java

The available toolkits for Java fall into two general categories: Those that implement one of the standard Java APIs for graphical user interfaces (i.e. are modelled after either AWT or Swing) and those that define their own API, often in order to

make some existing non-Java toolkit available to Java programmers. Toolkits that merely provide alternative implementations of the standard APIs are of course restricted in much the same way as the toolkit library they try to emulate.

3.1.1 Abstract Windowing Toolkit (AWT)

The *Abstract Windowing Toolkit* was the first graphical toolkit for Java as already explained in the introduction to this thesis. Its goal was to provide a standard set of components available across all platforms supported by the Java language itself. The components themselves are implemented in native, platform-specific code and consequently follow the look and feel defined by the platform.

To do this, the AWT toolkit introduced the so-called "peer" class model for user interface components: Every component actually consists of two separate objects, a platform independent object that is implemented in pure Java – this is the one that is accessible through the public API – and a platform dependent part that typically contains mostly native code (i.e. either C or C++) for interaction with the native graphics toolkit. This second part is called the *peer* object.

This separation allows for a simple way to replace the peer classes: Since they are not part of the public API, they can be transparently replaced with a completely different implementation (which needs to follow the same interface, obviously). This strategy is known as the *Bridge* design pattern (see [7]). Each platform has its own specific realization of these peer classes, always based on the "default" toolkit used on that platform. The drawback is, however, that the variety of available component classes is severely limited by the fact that the AWT has to restrict itself to the common subset of the components available on all supported platforms. Also, the features supported by these components are limited to what every platform could support at that time, which is why AWT components are rarely used today. For example, there is a non-editable combo box but no editable variant, because this could not be supported on all platforms.

The situation regarding the thread-safety of the AWT is somewhat ambiguous: The peer classes do partly allow concurrent access by multiple threads, but the AWT public API classes are no longer officially considered thread-safe, as is clearly stated now in Sun's Java tutorial:

> Note: Although this section talks about Swing, the same issues apply to all Components. Specifically, AWT components are not guaranteed to be thread safe.
>
> – excerpt from "How to Use Threads" in the JFC Swing Tutorial [26]

The AWT in fact initially allowed concurrent access to its components and was considered *thread-safe*, but this promise was later dropped by the developers at

Sun because it simply could not be kept due to some ongoing locking problems. An analysis of these problems can be found in the web-log of one of the developers, Graham Hamilton [9].

What remains is that only the dedicated event thread is allowed to interact with the user interface components. Starting with the JDK 1.3 it is also possible to delegate execution of `Runnable` objects to the event thread, so it is possible to update the components from another thread using this technique.

3.1.2 Swing (Java Foundation Classes)

Starting with Java version 1.2 there is a newer API for user interface components in Java: The *Swing* class library, which was originally developed as a part of the larger *Java Foundation Classes* project at Netscape. The major design goals for the Swing toolkit were:

- platform-independent, portable implementation of all components in Java itself (as far as possible), including the rendering of the components themselves
- extensive backwards-compatibility with the AWT components, allowing both Swing components inside AWT containers as well as AWT components inside Swing containers (with some limitations)
- pluggable "look and feel" to emulate (in part) the look and behaviour of the native components of the different supported platforms: X11/Motif, MS Windows, "Metal" look
- *Model-View* separation of many components, allowing for multiple views of the same data
- all components follow the JavaBeans conventions (described in [18])

As a consequence of the first two design goals, however, two new problems have been introduced by the Swing toolkit compared to the AWT:

Firstly, Swing components still feel somewhat sluggish – even on moderately fast hardware today – both when looking at the time to create new windows and user interaction with the components like scrolling, but this problem is bound to disappear over time due to faster computer hardware.

More importantly, the Swing class hierarchy is based on different classes from the AWT, which leads to some unusual inheritance relationships in the Swing API as new classes had to be fitted in there: For example, the `JFrame` class representing top-level windows in Swing is derived from the similar AWT class `Frame` in order to be displayable on the screen in an AWT compatible way. Alas, this leads to the

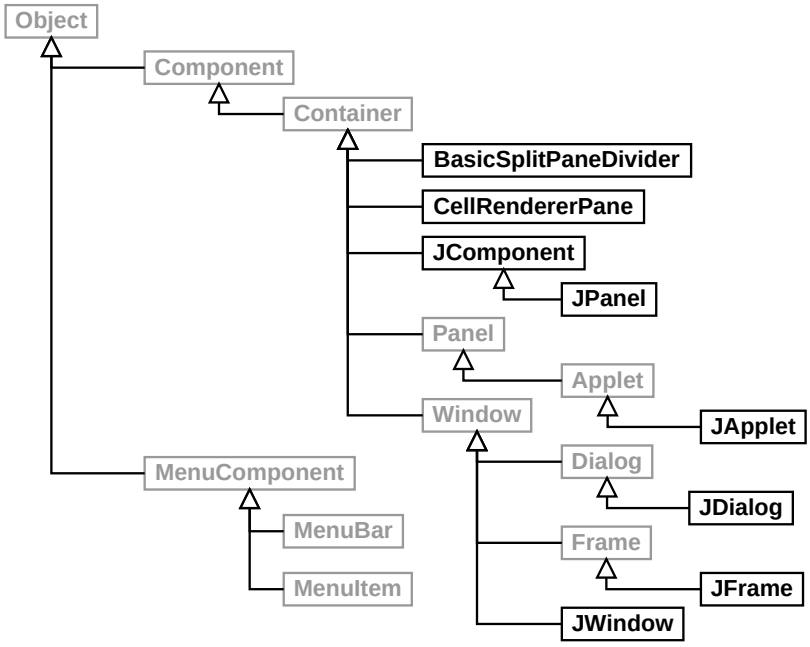

Figure 3.1: Component class hierarchy of AWT/Swing

immediate problem that a `JFrame` has to be used differently from most the other Swing components, which are subclasses of the `JComponent` class instead.

The same can be seen in other places as well, as illustrated in figure 3.1. In particular, there are about seven different Swing container classes unrelated to each other in the class hierarchy. Consequently, a lot of the basic Swing component methods are therefore either not available for these classes or had to be re-implemented for each class.

Interestingly, the separation between the `Component` and `MenuComponent` classes present in the AWT has *not* been carried over into Swing[1]. This was both technically the right thing to do and shows that the designers were willing to break compatibility in some places. Doing so for all components would probably have been beneficial in the long run.

Of course, any toolkit that claims to be "100% pure Java" has not much of a choice here: It has to depend on a few selected (primitive) AWT components to be able to display anything on the screen at all, since even basic operations like opening a window require interactions with the native graphics library on any platform, which is here done by the AWT on behalf of Swing. What can be observed is that obviously consistency was considered less important that compatibility with the AWT. Whether this was the right decision in the long run remains to be seen.

[1] This separation was caused by differences in the event handling for menu components in Java 1.0 that should not have been visible to the programmer to begin with.

As already mentioned in the introduction to the thesis, the Swing toolkit has not been designed to be thread-safe either, and so it also shares the limitation that only the event thread may interact with the user interface components. If any other thread needs to update the interface, it is necessary to push code (`Runnable` objects) into the event thread for execution using the `invokeAndWait()` and `invokeLater()` helper methods of the `java.awt.EventQueue` class. The rationale behind this is explained in [19].

3.1.3 Eclipse Standard Widget Toolkit (SWT)

The *Standard Widget Toolkit* is part of the Eclipse Platform project and provides a common and portable programming interface (API) for user interface components. It includes its own class hierarchy for components (not based on Swing) and has implementations of these for different platforms (Windows, X11/Motif, X11/GTK, Mac OS/Carbon).

This fulfils one of the project's stated main goals: To provide an efficient implementation of user interface components that is portable across different platforms. However, some features may exist only on a specific platform, like embedding of ActiveX controls, which is only possible on Windows. The Eclipse Development Environment (IDE) is the major application based on top of the SWT toolkit.

When looking at the interaction between threads and the GUI components, the rules and restrictions applicable to the underlying toolkit apply to the SWT as well, and it explicitly does *not* try to hide these from the application programmer, nor does it attempt to provide consistency here across the different platforms. As a consequence, portable programs simply cannot rely on thread-safe access to GUI components at all, so the same model already seen for the Swing toolkit is used again here: There is one dedicated thread that is responsible for the complete event handling process and methods are provided to schedule `Runnable` objects for invocation by the event thread either synchronously or asynchronously (i.e. without waiting).

3.1.4 GNU classpath

The goal of the *GNU classpath* project is to create a completely free, GPL licensed implementation of all of the Java class libraries compatible with those provided by Sun Microsystems as part of the Java SDK. Currently, only the Java compiler from the *GNU Compiler Collection* (GCJ) and the *Jikes* Java compiler (initially developed by IBM) are supported, not the Sun Java compiler.

There are two sub-projects within GNU classpath that are relevant in the context of this thesis:

AWT/GTK Peer Classes

This version of the AWT peer classes using the GTK toolkit is done in much the same way as the original AWT peers in Sun's JDK, implementing the same peer interfaces in native code. Therefore there is no fundamental difference on a design level between this peer implementation and the one provided by Sun for X11/Motif and MS Windows.

Just like the AWT peers, the GTK peer classes permit thread-safe access to their methods, using the *thread awareness* of the GTK toolkit: there is a global mutual exclusion lock that is requested at the start of each method, yet it remains unclear whether this translates into a thread-safety for the combination of the peer classes with the platform independent parts of the AWT, the documentation does not say anything definitive on that matter. It would be possible to extend this locking scheme to the Java classes as well using a monitor object.

Because these classes simply re-implement the AWT interface, the same problems already described above regarding long-running event handlers occur here as well. In addition, the GTK peer classes share the same limitations regarding the rather small set of available component classes. The main advantage remains the efficient native implementation of the GUI components.

Swing Implementation on top of GTK Peers

There is also a yet unfinished project to provide a re-implementation of the Swing API on top of the GTK peer classes, analogous to the Swing classes in the JDK. The class hierarchy is – as far as it is complete – exactly modelled after the hierarchy of the original `javax.swing` package, as expected, including all its inconsistencies. Because the original Swing is not thread-safe by design, the re-implementation follows the same idea and is likewise not thread-safe, so all the points mentioned above for the Swing toolkit apply here as well.

There have been no efforts yet to port this part of the classpath project to platforms beyond Unix/X11. But considering that the underlying toolkit is portable enough, it should be possible to do so.

3.1.5 Java-Gnome

The *Java-Gnome* project is part of a larger effort by the developers of the Gnome desktop to provide bindings for a number of programming languages for all of the core Gnome libraries, and Java is just one of the supported languages. Since Gnome is built on the GTK toolkit, the Java-Gnome project also includes a Java API for both the basic GTK library as well as additional functionality provided by the Gnome framework. It does so by creating individual wrapper classes for each of the components found in the GTK toolkit. The actual implementation is completely independent of the AWT and Swing interfaces and uses native GTK components exclusively for its user interface elements. This is implemented in a large collection of Java classes that refer to native methods for performing all GUI operations.

The most important features provided by the GTK part of Java-Gnome are:

- By completely relying on a native toolkit, the component implementation can be very efficient.
- Due to not needing to maintain any form of compatibility with either AWT or Swing, a much bigger set of components is made available than would be possible with the AWT, and the class hierarchy is not suffering from the inconsistencies of the Swing API. In particular, there is a common base class for all graphical components.
- The use of native methods is similar to that used in a peer model, but the fixed choice of the toolkit makes the abstraction provided by the separation between a platform-dependent and -independent class for each component unnecessary. Because the component classes are really just a thin wrapper around the native GTK components, they do not include any locking mechanism, so the same restrictions for multi-threaded programming apply in this case: Only the event thread is allowed to interact with the GUI components at all, and this thread is also executing the event handlers. If an event handler does not return quickly to the main loop, the user interface will become unresponsive during that time.

  As a solution to this problem, the library offers several methods for a custom thread to delegate the invocation of a `Runnable` object to the event thread, just like the methods `invokeAndWait()` and `invokeLater()` are used in the case of AWT and Swing.

In theory, the GTK part of the Java-Gnome project should be usable on other platforms as well, though according to the developers no work has been done in this area.

3.1.6 Qt AWT

This is an (incomplete) implementation of the Java AWT peer classes based on the Qt component library for C++ [4]. Qt has been developed by the Norwegian company Trolltech and is used extensively in the KDE desktop environment for X11 that is commonly found on Unix and Linux workstations. The initial motivation for *Qt AWT* was the desire to run Java applets in KDE's Konqueror web browser without relying on a Java plugin, which did not exists at the time. The implementation is rather unfinished and seems to have been abandoned several years ago in favour of the *Java Applet Server*, which itself was dropped later on once support for the Netscape plugin API was in place, which allows the use of the official Java plugin inside the browser.

The basic design (and the parts of the code that have been done) is comparable to the GTK implementation of the peer classes in the GNU classpath project mentioned before.

3.1.7 Qt Java

What is *Java-Gnome* to the Gnome project is *Qt Java* to the KDE project: A way to build programs using Qt (and KDE) components in Java.

It is designed in essentially the same way as Java-Gnome is done: Each of the Qt component classes is represented by an equivalent Java class with native methods, where each Qt method is simply mapped onto a corresponding Java method. Due to the particular nature of event handling in Qt – for some kinds of events it is necessary to extend the Qt component class and re-implement the corresponding handler – *Qt Java* takes special care to allow subclassing of Qt classes in Java.

Because of the similar general architecture and the fact that Qt like GTK is not thread-safe on the C/C++ level, the same points already mentioned for Java-Gnome also apply in this case: Only one thread has access to the user interface components, the event handling process is designed for single-threaded use and the event thread provides a mechanism to queue `Runnable` objects for invocation by the event thread at a later time. This is the only provided way to update the interface from any thread other than the event thread.

3.1.8 KDE Java Applet Server

The *Java Applet Server* mechanism was created as a different way to support Java applets in the KDE web browser, Konqueror. The applet runs in its own Java virtual machine in a separate process, it is only the window of the applet that

is embedded into the browser window, using standard X11 technology of widget embedding across applications (and is thus not portable to non-X11 platforms).

At first this does not look very much related to the work presented here. But a closer look reveals a specific commonality in a design aspect:

Each time an applet is loaded, the web browser spawns a new process and starts a new instance of the Java virtual machine in it. This process is controlled through a pair of pipes (I/O file descriptors) to pass control commands (like `start()` and `stop()`) to the applet. In the reverse direction, the applet can send commands to the browser to load a new URL or display some text in its status bar. The amount of interaction with the applet is very limited, but the general model of replacing method calls with process communication is clearly visible.

Though the applet server was in a working state, it has become obsolete since support for using the standard Java plugin was added to the KDE web browser.

3.2 Non-Java Graphics Toolkits

The second part of this chapter now presents the approaches taken by some other well-known graphical user interface toolkits towards the handling of multi-threading in general and multi-threaded access to the provided interface components in particular. Comparisons to Java will be drawn where appropriate.

3.2.1 MS Windows Graphics Device Interface

The native Win32 Shell API and especially the *Graphics Device Interface* (GDI) on Microsoft's Windows GUI environment take a very different approach to event handling[2]:

Events in the GDI are internally represented as *window messages* that are dispatched to the event handling procedure associated with the window in which the event occurred, often referred to simply as the *window procedure* or `WndProc()` in the API. Each window can have its own procedure that can handle all the events related to this particular window. Combined with the ability to run several threads in one process, this can be used to easily distribute the application's event handling into different threads (but at most one per window). Each thread can create its own windows, but these windows and all components therein may only be accessed by the thread that created that window.

[2] Actually, the functions are spread across several different libraries in addition to the GDI, but such distinctions do not matter here.

Other than this fixed mapping of one `WndProc()` per device window, there is no other pre-defined mechanism for a finer-grained event distribution. In a similar way to the `invokeAndWait()` method used by the Java-API, it is also possible to (either synchronously or asynchronously) forward such event messages to the thread that owns a particular GUI control by issuing a `SendMessage()` call to the thread responsible for a particular window. So it is in fact possible to have a model with at most one thread dedicated to each window, but additional threads created from event handlers cannot update any user interface components they did not themselves create.

3.2.2 Windows Forms (.NET)

The situation described before is also very similar for the *Windows Forms* library that is used for example by C#, C++ or Visual Basic on the .NET platform. The general rule that applies here is often quoted as:

> Though shalt not operate on a window from other than its creating thread.
>
> – saying known as the "prime directive of Windows programming"

In essence, the components created by each thread belong to the context of that particular thread and must not be accessed from any other thread. To make it easier for the programmer to create invocation constructs for executing code in a particular window's event thread, C# supports the notion of "delegates": references to a method with a specific type signature. This is more or less equivalent to a parametrized `Runnable` implementation with some syntactic sugar applied by the language.

A slightly different approach is planned for the "next generation" Windows graphics API that is currently in development: *Avalon* will support separate threading contexts for each control. This essentially means that while the control implementations themselves are still single threaded, each thread may request access to a specific threading context by calling the corresponding `Context.Enter()` method on the context, followed by a `Context.Exit()` when it is done.[3] This still requires care on the side of the programmer, however: He still needs to remember which controls belong to which thread, so that the appropriate threading contexts can be requested when necessary. On the positive side, this can of course also be a bit more efficient than doing this blindly, since it avoids acquiring any unnecessary locks.

[3] Note that this is basically a locking operation that can block.

3.2.3 Mac OS X Application Kit

The history of the primary user interface toolkit on Mac OS X (also known as the "Cocoa" API to developers) is interesting in that it was actually inherited from the acquisition of NeXT Inc. by Apple in 1999. In large parts, it is the OpenStep programming interface that was developed at NeXT for use on their NeXTSTEP and OPENSTEP operating systems back then.

The Mac OS X Cocoa API consists of several independent libraries commonly referred to as "Kits" on Mac OS X. The most important ones for application programming are the *Foundation Kit* and the *Application Kit*. The *Foundation Kit* provides a basic set of data structures (lists, arrays, hash tables, sets), support for threads and Unicode strings and – more importantly here – also a generic main loop abstraction called `NSRunLoop`. The *Application Kit* contains all the various user interface components like windows, buttons and so forth.

The Foundation Kit allows for a limited amount of multi-threading: It is possible to have multiple main loop instances and run each of these in its own thread in parallel. Alas, the Application Kit that manages the GUI components is not thread-safe in any way. Therefore, an application can contain multiple threads (and could in theory contain even multiple main loops), but only one of these can receive events from the window server and interact with the user interface components provided by the Application Kit. Consequently, the threading model for GUI applications is actually very similar to the one used in Java, with the difference that the main loop is made explicit in the API. The main loop also supports invoking methods asynchronously in the event thread.

3.2.4 BeOS Application Kit

The Be Operating System was an attempt at a small micro-kernel operating system geared towards high multimedia performance. While initially only available in combination with custom hardware, it was later ported to standard Power PC and Intel/AMD PC systems and sold independently. The company behind BeOS ("Be Inc.") is no longer in existence, today. Yet it remains a very interesting experiment with a different architecture for designing application programs:

Graphical components on BeOS are handled by the *BeOS Application Kit* library, a C++ framework containing a wide variety of predefined user interface component classes (a thorough description can be found in [25]). The BeOS Application Kit is one example of the few class libraries that were designed right from the start for use in a multi-threaded environment. A part of the class library is illustrated in figure 3.2 on the next page:

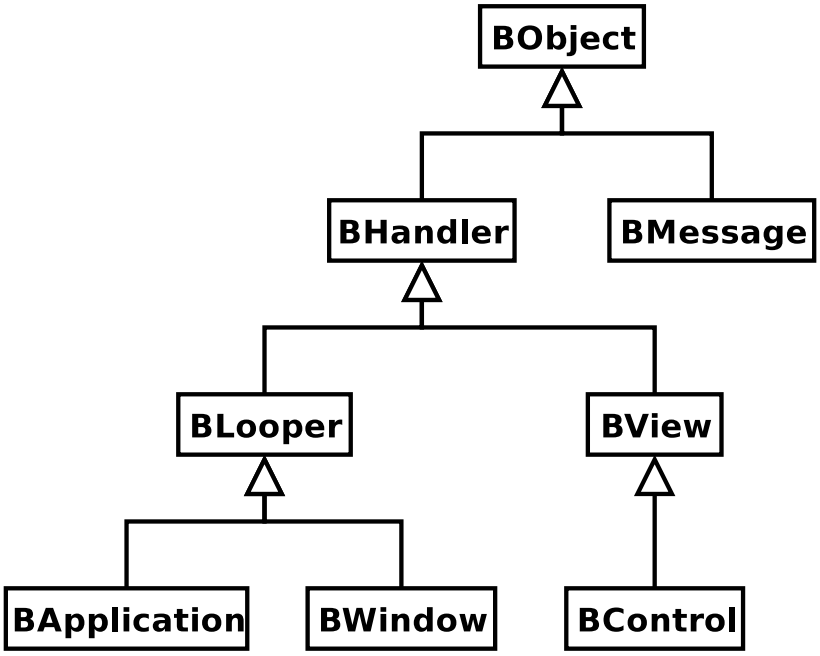

Figure 3.2: Crucial classes in BeOS Application Kit

The central role here plays the `BLooper` class, basically an object-oriented main loop abstraction that is able to receive event messages encapsulated as an object (actually, a `BMessage`). Each instance of a `BLooper` has the ability to run in its own thread, and in fact all subclasses except for the `BApplication` class automatically create a new thread when instantiated. The `BApplication` instance is special in that it executes in the program's main thread (it is typically invoked at the end of the `main()`), so there is no need for a separate thread to be created in this case.

Classes like `BWindow` and `BApplication` are derived from `BLooper`, which means that every application window *always* gets its own main loop that is responsible for handling events originating from the components in that window. This is enforced by the toolkit, so the option of using a single thread and a single main loop simply does not exist here. Event messages can be forwarded between different main loops, even allowing for the use of several main loops in one window if desired. This makes it all very easy to run several parallel event loops inside one application, which is ideally suited for a multi-document application model where each document window is supposed to behave independent of the others.

The Application Kit also automatically handles locking for multi-threaded access to components – apart from some special cases like the clipboard that require cross-application synchronization – so any thread may access user interface components at any time.

```
button .b -text "Hello" -command {puts "Ouch"}
pack .b
```

Figure 3.3: Event handling example in Tcl/Tk

3.2.5 Inferno

Inferno is an operating system developed at the Computing Science Research Center of Bell Labs (now a division of Lucent Technologies) as a successor to the Plan 9 operating system. The current development is in the hands of Vita Nuova. Inferno typically runs as an emulated environment on a host system like Windows or Linux (it can also be used as a stand alone system) and offers an environment to programs comparable to the Java VM, including a virtual machine abstraction, portable code and data types and automatic garbage collection.

The default graphical user interface on Inferno is a simplified variant of the *Tk toolkit* originally designed by John Ousterhout for use with his *Tool Command Language* (Tcl) scripting language [20]. Figure 3.3 above shows a simple example of how to create a button in Tcl/Tk that has the label "Hello" and prints the string "Ouch" to the console when the button is clicked with the mouse. The name of the button component in Tcl is "`.b`" to indicate that it is a direct child of the toplevel window. Component names are actually path names that reflect the widget hierarchy inside the application, starting at the top-level window.

Tcl/Tk uses a standard single event loop based model for event handling and does not support multiple threads. The handler code is attached to a component in the form of Tcl command strings with the `-command` option when creating a widget or the `bind` command later on, and the attached Tcl command string is then executed whenever the corresponding event occurs.

The Inferno system uses a very different approach to event handling, however: Applications on Inferno are typically written in *Limbo*, a C like language running in a virtual machine. All user interface components (except the top-level windows) are created and manipulated by sending Tk commands as strings to the display server thread, which contains a simplified re-implementation of Tcl/Tk that only supports a subset of the basic widgets offered by the standard Tk. In essence, a small Tcl interpreter is embedded into the display server, and the Limbo program can communicate with it using Tk commands. Event handler code can be attached to components in the same way as in Tcl/Tk, but since the subset of Tcl/Tk implemented in the display server does *not* support user defined variables or any control structures, it is not useful for implementing event handlers directly.

Instead, Inferno supports *typed channels* as a general mechanism for synchronous communication between threads running in the virtual machine, and the display server can use these to send event messages to the Limbo application. This can

```
channel := chan of string;
tk->namechan(toplevel, channel, "channel");
tk->cmd(toplevel,
    "button .b -text Hello -command { send channel Ouch }");

expl := <-channel;   # will see Ouch when button pressed
```

Figure 3.4: Event handling example in Limbo/Tk

be seen in the Limbo example in figure 3.4: A channel is created in Limbo and is assigned a name in the display server. It can then be used to send back arbitrary messages to the application, which can wait for events on the channel using the channel *receive* operator "`<-`".

The actual event handler is then done completely in Limbo code, executing Tk commands via the `Tk` object if it wants to update the user interface. It is left for the programmer to decide whether to use different channels for the various GUI components or to pass most (or even all) events through a single channel. Since there is no pre-defined main loop class, the program has to either use multiple threads to handle multiple event channels at the same time, or use the `alt` control structure of Limbo (which is itself taken from the Alef language in Plan 9) to wait for events on any number of channels in a single thread.

Due to the separation between the display server and the Limbo application (Tk commands are sent across a channel to the display server as well) and the synchronization provided by the channels, any number of threads can communicate with the display server.

3.2.6 Toolkits for Ada

The *Ada* programming language is interesting in this context because it is highly regarded for its structured, high-level facility for concurrency. The unit of concurrency is a program entity known as a "task" in Ada. Tasks can generally communicate implicitly via shared data or explicitly via a synchronous control mechanism known as a "rendezvous". If access to shared data is sufficient, a data item can also be defined as a *protected object* in Ada 95, which causes all accesses to this data item to be automatically synchronized.

The core Ada language does not include any features for graphical user interface programming, but several graphical toolkits exist separately. These are usually based directly on a native toolkit on a specific platform, which means that most of them are not portable across different systems. Examples of these include *GtkAda* (based on GTK), *Carbon* (Mac OS) and *x11ada* (X11/Motif).

Sadly, the direct mapping of the platform API to Ada also transfers the limitations of the native user interface toolkit regarding multi-threading to the API visible to an Ada program. For example, the *GtkAda* manual states that:

> Note that Gtk+ under Windows does not interact properly with threads, so the only safe approach under this operating system is to perform all your Gtk+ calls in the same task.
>
> Under other platforms, the Glib library can be used in a task-safe mode by calling `Gdk.Thread.G_Init` and `Gdk.Thread.Init` before making any other Glib/Gdk calls. [...]
>
> When Gdk is initialized to be task-safe, GtkAda is task aware. There is a single global lock that you must acquire with `Gdk.Threads.Enter` before making any Gdk/Gtk call, and which you must release with `Gdk.Threads.Leave` afterwards.
>
> – "Tasking with GtkAda" in the GtkAda User's Guide [1]

The same is true for the other Ada toolkits mentioned above as well: Either there is no support for tasking in the GUI at all, or it is not integrated transparently into the language. So, while Ada itself indeed offers a higher-level model of thread-synchronization than Java, looking at the GUI toolkits for Ada does not seem to turn up any new ideas.

3.3 Summary

When comparing the available toolkits for Java, it becomes immediately apparent that all of them share the same fundamental design principle regarding multi-threading: They use a single event thread and main loop and simply tell the programmer not to touch the user interface components from any other thread. While this is certainly convenient for the toolkit implementation, it does not help to make the life of the application developer easier.

The non-Java toolkits clearly show that there is room for improvement when looking beyond Java: There is indeed a wide variety in the degree of thread support in GUI toolkits available on different platforms, from the essentially single-threaded model of Mac OS X to the fully multi-threaded one used in BeOS. Of special note here is the channel based communication with the display server as used in the Inferno system: This was a source of inspiration for the design of the Java toolkit presented in the following chapter of this thesis.

Chapter 4

Design of a Message-Based Approach

This chapter presents the design of a message-based approach to event handling for Java. It starts out by looking at the reasons behind the limitations apparent in today's graphical toolkits available for Java and discusses some possible solutions to these problems. This part also includes a set of design objectives for the approach presented here.It is followed by a description of the design of a message-based approach to event handling for Java built around a client/server model, and it is shown how this design can actually fulfil the objectives laid out here. The actual prototype implementation of the approach is discussed in detail in the next chapter.

4.1 Arguments against Multi-Threading

As can be seen from the analysis in the previous chapter, the main limitation of the toolkits available for Java today is the lack of support for multi-threading. There are mainly two separate reasons for this:

- The native toolkits provided on most platforms are themselves not thread-safe. In fact, user interface libraries are seldom created from scratch, and often the basic infrastructure they have rely on is old and rarely updated. This leads to the fact that also newly developed native toolkits often cannot guarantee thread-safety even if they wanted to.

  For instance, most of the core X11 libraries date back to the 1980's and have never been designed with thread-safety in mind. As a consequence, most of them still are not considered thread-safe today, so any native toolkit that sits on top of them like GTK simply cannot be completely thread-safe

either. GTK tries to solve this by introducing a form of manual mutual exclusion locking at the GTK interface level, which is neither transparent nor particularly portable. If a platform could already provide a thread-safe native toolkit, it would be easy to make this accessible to Java in a thread-safe way. But that simply is not the case today.

- A single-threaded application design is "good enough" in most cases. While the benefits of multi-threaded programming become apparent for large or networked applications, many smaller programs can live just fine without them, or can be made to fit into the single-threaded model with a bit of extra work. Additionally, the application developers are already familiar with the single threaded, single main loop event handling model. So the question of whether it is worth for toolkit designers to introduce the extra complexity for making the toolkit multi-threaded is worth it, is answered with "no" more often than not.

What can be learned from this? Firstly, that requiring a thread-safe native toolkit is simply not an option. Even if a limited degree of multi-threading is supported, as seen for example in the Win32 API, this is not enough and it is not portable either. So one has to accept that native toolkits currently simply are limited to single-threaded use, and start to design from there.

The second point is that introducing multi-threading into the toolkit requires adding synchronization to the code (for example in the form of locks or monitors), which can be difficult to get right, especially when callbacks from the native code into Java are added to the picture. This can potentially lead to lock ordering problems and cause deadlocks, which were exactly the kinds of problems that finally motivated the Java developers at Sun to drop the idea of thread-safety for the AWT.

So it might be a good idea to take a step back and ask whether to add locking at this level it is right approach in general, given these problems. How about avoiding any kind of explicit locking at the toolkit level in favour of an implicitly synchronizing communication pattern? This is what leads to the idea of modelling the message flow between the application and its user interface instead of looking at individual locking problems. When thinking about talking to the user interface at a higher level in terms of message flow, the problems may be easier to solve. The need for synchronization does not magically vanish, of course, but it is moved to the communication primitives instead, and these are much simpler to verify.

4.2 Design Objectives

This section briefly summarizes the main objectives that the design of the proposed message-based approach tries to achieve:

1. Allow access to the user interface components from arbitrary thread contexts. The restriction that only one selected thread (like the "event thread") may interact with the components is very cumbersome when trying to distribute event handling code into multiple threads, especially when background threads regularly need to update the user interface. I consider this basically a technical limitation of the platform's graphics library that ideally should not affect an application programmer.

2. Support parallel event handling for separate logical components (like windows or independent parts like tabs inside a window) of the application using multiple concurrent event loops. The job of forwarding the events to the correct main loops should be transparent to the programmer.

3. Provide a consistent and clean interface to the application programmer, completely independent of the AWT and Swing component classes. The goal here is to avoid the inconsistencies seen in the Swing API due to the fact that it was based on the AWT. To remain flexible, this interface should be toolkit independent and able to support multiple backend implementations, including fast native toolkits like GTK or Qt or the Win32 Forms API on Windows. If a native toolkit library is available on multiple platforms, the code should be portable with a simple recompile, if possible.

4. Enable an application to display its user interface on a different system from the one is it running on. Although Java is portable to a wide range of systems, it would be nice to provide some form of location transparency at the toolkit level as well. Given that both X11 and solutions like VNC are becoming increasingly popular, this would provide an additional benefit.

4.3 Possible Solutions

The following section will look briefly at each of the objectives listed above in turn to discuss how practical it is to achieve this within the currently available Java toolkits and how a message-base approach will help to solve it:

1. Allowing arbitrary threads to interact with the toolkit components is mostly a thread synchronization problem: Since it simply is not reasonable to require the underlying toolkit to be thread-safe itself (as described in the introduction to this chapter), all access to the toolkit functions has to be synchronized

by the glue between the application and the native toolkit itself. While this could be added to one of the existing toolkits as well, it requires a lot of care and verification.

But there are other problems that cannot be resolved using locking alone: The limitation that on Windows only the thread that created a window may interact with the components therein is a fundamental restriction of the current Win32 API that cannot be avoided. The only way to solve this is to use just a single thread for all interaction with the native toolkit, which is exactly what the message-based approach is all about: It offers a very simple and flexible way around this limitation by using message passing to delegate work to a single toolkit thread.

2. Adding support for multiple event loops with the current Java toolkits is not feasible for two reasons: There is no way to tell a component to deliver events to a particular main loop, which makes is impossible to handle this transparently, and it contradicts the single event thread model, so that any additional event loop would have to delegate its GUI updates to the main event thread, which kind of defeats the purpose of having more event threads in the first place.

   A message-based approach on the other hand is almost ideally suited for this: Since events are distributed in the form of messages anyway, it does not make any difference to which main loop they are dispatched for handling. The only requirement is that this information must be provided when attaching an event handler to a component.

3. Offering a consistent interface to the programmer is actually not much of a problem when one is willing to sacrifice compatibility with the AWT. All of the Java toolkits except for Swing do that already, but given how Swing did this wrong in my opinion, it may nonetheless be worthwhile to add this as one of the goals. A message-based approach can do this as well, although it does not have any special advantage here.

   Supporting native toolkits is equally possible as demonstrated by e.g. the Abstract Windowing Toolkit or the Standard Widget Toolkit of the Eclipse project. The way this is done in both APIs is to provide a set of platform independent component classes that internally delegate the platform specific operations to native peer classes using the Java Native Interface. Using the message-based approach for this is conceptually not very different, actually: Both add a level of indirection that permits redirecting a method call to a platform specific implementation.

4. Supporting location transparency in the toolkit is close to impossible for any of the existing Java toolkits without redesigning them first to also use a

message-based model similar to the one described here. This is quite natural, since being able to support this has never been a design goal for any of them.

Of course it is possible to use existing solutions instead like X11 remote displays, which is not portable to either Windows or Mac OS, or one of the remote desktop protocols like VNC or Windows Terminal Server, which however do not work at the application level and can consume a lot of network bandwidth.

The message-based approach offers an easy way to support location transparency here: The design of an architecture that uses message passing as the basic form of component interaction makes it suddenly fairly simple to move some of these components into a separate process, or possibly even to distribute these components onto different systems by using a TCP socket for exchanging the messages instead of channels inside the Java VM.

4.4 A Message-Based Approach

The approach presented in this thesis is built upon the general idea of applying the concept of *messages* flowing between objects to the field of user interface programming in Java. In particular, it incorporates the idea of using a separate display server component as found in the Inferno system and X11 and permits concurrent event handling using multiple main loops inspired by the BeOS Application Kit. An earlier version of this design proposal has already been published before in [15].

The basic idea is to resolve the conflict between the application that wants to be multi-threaded and the toolkit library that cannot support this by logically separating the two from each other, allowing the application to run independently from the toolkit library. This approach satisfies both sides: The application is no longer affected by the restrictions imposed by the toolkit, and the toolkit can run in its own single-threaded world that it is used to, in particular no locking has to be applied there. The glue used between the two is a communication framework built around the idea of message passing:

All operations on components by the application's threads are turned into messages sent to the toolkit, and the flow of events generated by the toolkit is transmitted to the application in the same way in the form of event messages. This works because the message passing implicitly forces a serialization of the requests and events, so that it is possible to add the message handling to the toolkit without the need to make it multi-threaded, whereas the same is not possible using normal method calls. To make this proposed separation transparent to the application, proxy objects are introduced in the user interface component API that perform all request handling on behalf of the application.

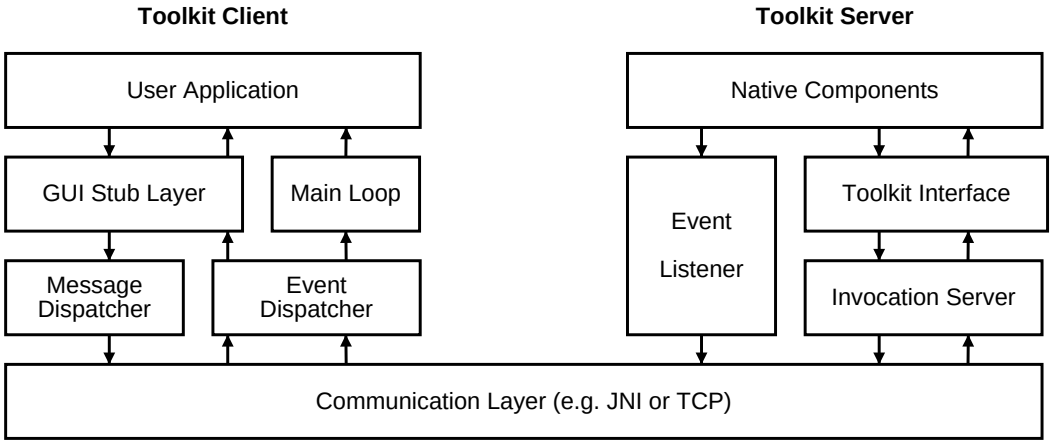

Figure 4.1: General overview of the system architecture

By passing events from the user interface to the application via messages as well, event messages can be easily dispatched to any number of main loops inside the application: There is no longer any need to restrict the user to just one event handler thread. Each thread can independently listen to events from controls assigned to that thread. Beyond allowing the application to become multi-threaded, using messages for the communication between the application and the toolkit has some additional advantages as well: The separation introduced in this way allows the application to become independent of the toolkit, so that the toolkit implementation is easy to replace. Furthermore, there is no reason to limit the communication to a single process: The toolkit can also be run inside its own process, effectively resulting in a kind of toolkit server process, which can even be moved onto a different system if desired.

The transport mechanism that is used for communication between the application's threads and the toolkit varies depending on how far the separation between the two should go. If both run inside the same process, *channels* can be used as the primary means of thread communication. The concept of channels used here is taken from C.A.R. Hoare's *Communicating Sequential Processes* [12] as explained in section 4.5 on the next page. If the toolkit is running in its own process, some form of inter-process communication has to be used instead, like pipes or network sockets.

A general overview of the system architecture and its components is shown in the diagram in figure 4.1. Each of the boxes represents a logical component that may correspond to a class or a set of classes in an actual implementation. These components can be divided into two groups, forming a *Toolkit Client* (often simply called the "frontend") and the *Toolkit Server* (the "backend"). The term "Native Components" has been used in the diagram on the right to refer to the actual components inside the backend's toolkit library. This does not necessarily imply

that these need to be implemented in native C code, that is just one possibility. One example of components that are not implemented in C is shown in the Swing backend in the next chapter.

It should become immediately apparent that the architecture presented here is generally very similar to a distributed object model like CORBA or Java *Remote Method Invocation* (RMI), and this is not a coincidence: The goal of a distributed object system is similar to the approach presented here in that it offers a transparent way to access objects located on a remote server to the program. What looks like a plain local method call to the application is turned into a message behind the scenes that is sent across the network and executed in a server process. References to objects in the server that the client holds are in fact just message forwarding proxies for any operation that the server object supports. Yet even though the overall architecture has similarities with RMI, it is different from just providing RMI wrapper classes for (e.g.) AWT components, which would not (on its own) solve any of the problems regarding multi-threading.

The following sections will provide a short walk-through of the individual components shown in diagram 4.1 to describe their functions and their responsibilities within the overall architecture, starting with the basic communication pattern used in this approach: channels.

4.5 Channels

All communication between the threads on the Java side is realized with channels. Channels is a concept originally invented by C.A.R Hoare as a general means for process synchronization in his work on *Communicating Sequential Processes* [19]. CSP is an abstract mathematical notation for describing the interactions between processes. This was later picked up by Phil Winterbottom, who integrated the idea as a core concept into the programming language *Alef* [28], which he designed at AT&T's Bell Labs for the Plan 9 operating system.

While channels as used in CSP were initially unbuffered[1], Alef introduced buffered channels as a safe and more efficient way to interchange data between threads without blocking the sender on each write operation. Synchronization is handled in Alef with standard unbuffered channels or special lock objects handling mutual exclusion. Both buffered and unbuffered channels are standard data types in Alef and can be stored in variables or passed around as arguments (unlike CSP).

Alef itself never became widely used, but many of its ideas were integrated into the programming language *Limbo* used on the *Inferno* operating system. Limbo was

[1] It is possible to use buffered channels in CSP by inserting an intermediary buffer process

designed by Sean Dorward, Phil Winterbottom and Rob Pike as an integral part of the Inferno system and its primary programming language. Among the ideas inherited from Alef was the concept of channels as a mechanism for synchronization and general data exchange between Inferno threads. Limbo was the first system to use channels to deliver user interface events to an application.

The channel concept can be used in Java as well, both as a means for thread synchronization to model synchronization patterns similar to those in CSP and as a means for data interchange between threads, as used in Alef and Limbo later on. There have in fact been several efforts to use CSP style communication patterns for general multi-threaded Java programming, most notably by Peter Welch and Paul Austin [27] and Gerald Hilderink [11].

One could argue that the pipe streams provided by the standard `java.io` package combined with Java's ability to serialize objects can be used in a way very similar to buffered channels, but using them would require all data that should be transfered to be serializable and would in fact produce copies of the original objects on the reader side (which is not always desirable for passing objects around between threads). Very recently, a similar mechanism has been added to Java in the JDK 1.5 with the `java.util.concurrent` package and its `SynchronousQueue` and `ConcurrentLinkedQueue` classes.

4.6 Walk-through of the Design

This section explains the role of the different components in the overall architecture starting at the GUI stub layer and following step by step the way that a typical method call to a user interface component would take from the application. The only classes that an application will directly interact with are these stub classes (representing the components) and the main loop abstraction that is responsible for delivering events to the application's event handlers as described later.

4.6.1 The GUI Stub Layer

The stub layer forms the virtual component API that is visible to the application code: It is a set of component proxies that look like real components to the application, but in fact merely forward all requests to the actual implementation of the native components in the toolkit server. Nonetheless, this is the interface visible to the toolkit, so it must be designed with care. It should not be specific for one particular toolkit but rather designed similar to the Eclipse SWT, offering an abstract interface for several toolkit implementations. For example, it is necessary to decide here whether to present layout managers as used in the AWT or

Swing interface or to layout components using specialized subclasses of a generic container class that provide particular layout strategies to the applications.

Each method call on such a component proxy – including constructors – is translated into a *request message* representing this method call as an object, which is then passed on to the message dispatcher using a channel. One can distinguish here between *synchronous* and *asynchronous* messages: A synchronous message is used whenever a method needs to deliver a result value or can report an error to the caller. In this case, the method of the component proxy needs to wait for the response to this request (which is again delivered through a channel) from the event dispatcher, and it will return only when the response has been received. On the other hand, operations that do not produce a result value can use asynchronous messages, where the proxy method can return immediately.

Two aspects are noteworthy here: If errors are reported as exceptions in the toolkit server, these need to be transmitted as result values and must be reconstructed (and thrown) by the component proxy, because exceptions cannot be thrown "through" a channel or socket. Secondly, the *event dispatcher* is also responsible for handling reply messages. While this may appear strange at first, the reason behind it will become apparent later on.

4.6.2 The Message Dispatcher

The message dispatcher is the central component on the client side that handles all the interactions of the application with the toolkit. The job of the message dispatcher is to receive the request messages from the individual component stubs using a channel, and to forward these to the invocation server on the toolkit server side using a suitable communication mechanism. It is the responsibility of the message dispatcher to encode the message for transport if required, e.g. a Java object cannot be directly transmitted across a socket. The equivalent process is called *marshalling* in the Java RMI.

An important point here is that the complete interaction between the the component proxies and the toolkit server is encapsulated inside the two dispatcher components (message dispatcher and event dispatcher). As a consequence, these are the only classes in the toolkit client that have to be replaced with another implementation to provide access to a different toolkit server or to experiment with a different communication layer. All the classes visible to the application remain completely unchanged.

4.6.3 The Communication Layer

The communication layer provides the basic message transport mechanism by which the toolkit client and the server communicate. Many different transport mechanisms can be conceived here, ranging from simple local method calls inside the same Java process (possibly using the Java Native Interface) or channels to a completely distributed system involving remote method invocation akin to a model similar to RPC or RMI.

The main role of the communication layer is to pass requests from the message dispatcher to the toolkit server and to send back a response for a request if applicable (not all requests generate result values). Event messages that originate at the toolkit server are transported in the opposite direction in much the same way.

4.6.4 The Invocation Server

The invocation server forms the counterpart to the message dispatcher on the client side: It waits for request messages from the dispatcher. Once a request has been received, it has to be decoded into a format suitable for the toolkit server as appropriate, e.g. a TCP message has to be parsed and translated into native data types. The server also needs to maintain a mapping between the client component proxies and the server side native components in order to find the native object that the message was directed at, and to resolve any component references in the argument list ("unmarshalling" in the terms of the RMI).

The decoded request is then translated back into a method call to the native toolkit interface in the server. For operations that produce a result value, the result returned by the operation is likewise wrapped into a *response message*, encoded for transport and returned to the event dispatcher (not the request dispatcher) on the client side. The role of the event dispatcher in this is described in section 4.7.2 on page 51.

4.6.5 The Toolkit Interface

The part of the overall architecture that handles the final method calls to the native components on the toolkit server side is called the *toolkit interface* in this design. The name is inspired by the fact that this layer has to implement the interface provided by the client component proxies to the application: Each method call performed by the application results in exactly one method call to this toolkit interface layer. Any result values returned from here will be passed back to the application.

Because the API presented to the application and the interfaces provided by the native components are not going to be identical, and in special cases may even be substantially different, the toolkit interface layer plays the role of an adaptor layer between the two APIs. This is necessary because the goal is not to provide the native component API to the application but instead a more abstract interface.

4.6.6 Native Components

As already mentioned in the general overview presented in section 4.4 on page 44, the term "native components" is here just a generic placeholder for the actual component library used in the toolkit server, it is irrelevant for the architecture whether these are actually implemented in native code or not. And even though the toolkit's native components are always referred to as *objects*, this does not imply that the toolkit has to be implemented in an object oriented language at all (in fact, the GTK toolkit is a plain ANSI C library), so that the target object may just become another parameter to the native function call in the end.

As already mentioned before, the API that is used for the user interface components visible to the application (the component proxies) must not necessarily match the API of these native components, but it must be possible to implement all the functionality advertised to the client using the functionality offered by the native components in the server.

4.7 Event Messages

On the previous pages the individual steps that a request goes through on its way from the application to the native component library have been followed. Events flow in the opposite direction: They are generated asynchronously by the native components and are sent to the application in much the same way as request responses: In the form of *event messages*. Like requests, event messages can also carry arguments conveying additional information to the application about the activity that is described by the event. A fairly typical example of event arguments are the current mouse coordinates in the case of mouse events.

The next sections will provide a short overview of the components that are involved in the event dispatching process from the native toolkit to the application.

4.7.1 The Native Event Listener

Whenever the application requests that an event handler is added to a component proxy, the toolkit server responds to this by attaching its own native event listener to the corresponding native component. This event listener on the server side is a generic listener that is suitable for all kinds of events that the toolkit components may generate. Any such event generated thereafter by the component is handled in the event listener by creating a corresponding event message for this event and sending it through the communication layer to the event dispatcher in the toolkit client. And just like the request messages, the event messages have to be properly encoded for transport as well.

In practice, an additional optimization can be applied here: It is in fact only necessary to attach such a listener the first time an event of given type is requested from a component, because event multiplexing can also be done transparently on the client side, avoiding the need to transmit the same event multiple times from the server to the client. This is both more efficient and reduces event latencies.

Although this is not reflected in the diagram, it may also be a good idea to pass the event messages through a method in the invocation server in an effort to isolate the event listener component from any changes in the communication layer, so that only the invocation server needs to be replaced when using a different communication mechanism. Otherwise the event listener needs to be replaced as well in such a situation.

4.7.2 The Event Dispatcher

All messages that arrive across the communication layer at the toolkit client are handled by the event dispatcher. There are two kinds of such messages, response messages corresponding to requests that are still awaiting an outstanding reply in the client and event messages reporting about activity in the user interface. The reason that the event dispatcher is responsible for both of them is quite simple: Since there is only a single connection between the toolkit client and the server, both types of messages have to be transported across the same medium, and therefore there can only be one thread waiting for these messages.

While the semantic difference between these two message types is irrelevant to the communication layer, the event dispatcher needs to be able to distinguish between them to correctly decode and forward them: Response messages are matched to their corresponding request message and directly forwarded to whichever thread sent the original request. Event messages need to be dispatched instead to the main loops on which an event handler for this type of event has been registered

for the source component of the event. This is the place where event multiplexing is performed when multiple handlers have been registered for the same event.

4.7.3 The Main Loop

The main event loop very much plays the general role already described before in chapter 2.1.5 on page 17. It is responsible for sequentially dispatching all events delivered to this loop to the event handler methods inside the application code.

The main difference in the approach presented here compared to AWT or Swing is that the main loop is made explicit in the API, which gives the freedom to the programmer to actually have multiple main loops inside any given application. Each of these loops runs completely independently of the others, sequentially handling the events assigned to it by the event dispatcher. It would even be possible to handle the same event in several main loops in parallel if desired, though no guarantees are made regarding the order in which multiple copies of the same event would be dispatched to the different main loops in such a case.

4.8 Dealing with Modal Windows

There is one case that needs special care when implementing a main loop for the approach presented here: modal application windows. To understand why, it needs to be explained briefly what a modal window is and how it behaves differently from a normal, non modal window. A window in a user interface can generally either be used in a *modal* or *non modal* way, where the latter is the usual case. A *modal* window restricts all interaction of the user with the program to only that particular window, controls in other windows of the application are blocked as long as the modal window is on the screen. This is often used for dialogue windows that require immediate user attention before the program can proceed in any meaningful way. Modal windows are often used for displaying error messages or letting the user choose a file to open as a document in the application.

To the programmer, a modal window also looks a lot different: In the AWT and Swing for example, displaying such a window will cause the corresponding `show()` or `setVisible()` operation to block until the window is removed from the screen again. More generally: A modal window allows the application to open a window and synchronously wait for the window to close. This immediately leads to a very interesting question: If the modal window is shown from an event handler – and this is of course the usual case - this will have the obvious consequence that the event handler cannot return to the main event loop until after the window is closed. But how does the application then process events inside the modal window? As

described before, not returning from an event handler can have very devastating consequences for a normal Java program.

The answer to this question is that for modal windows, the operation that displays the window is a bit "magical": It does not return, but instead starts a new (recursive) main loop. Closing the window automatically terminates this main loop again and returns from the method that displayed the window to the application's event handler. This trick is why event handling for modal windows works at all in a strictly single-threaded event handling model. So how can this behaviour be simulated in a message-based approach? The problem here is that due to the logical separation of the application from the toolkit the application's main loop does not know about the modal window and will return immediately.

There are two ways to solve this problem: The simplest way is to define the `setVisible()` method as a *synchronous* operation, for example by making it return a value. This forces the method call to wait until the corresponding operation in the toolkit server returns. The other option is to start a recursive invocation of the current thread's main loop from the proxy component as well whenever a modal window is about to open. Doing this of course requires that the main loop implementation is able to support recursive invocations.

4.9 Java Applets

The message-based approach can be applied to Java applets as well: An applet is a graphical Java component that is designed for running inside a restricted environment in a user's web browser. The applet code is loaded on demand across the network, obviating the need to first distribute the application to the user. Instead of opening its own application window, an applet shares part of the screen with the browser.

Though it is of course possible to include both the toolkit client and the server into the applet, the more interesting case here is using a network connection between them: This leads to a model where the applet is merely functioning as the user interface layer for a server side application, becoming very similar to what is more widely know as a *web application* today: An server side program that is controlled by a user interface in the client's web browser. In fact, this model is almost exactly what Java application servers like *Jakarta Tomcat* and Apple's *WebObjects* try to provide: The illusion that the web application which is running on the server is transparently connected with its user interface in the clients web browser, using the stateless HTTP protocol. *WebObjects* even includes an interface builder that allows the developer to design an HTML user interface where events like button clicks can be visually connected to event handling methods inside the application.

But an HTML form is very limited in its model of interaction with the user: The interface is completely static, and the server can only update the interface when a submit button is pressed by the user. Furthermore, it always has to replace the interface *completely*, even if it just wants to update a single line of text, which may cost a lot of traffic. This problem can be mitigated to some extend by moving parts of the application into client side JavaScript: Using JavaScript on the client allows for much better interactivity, since it enables the program to handle arbitrary mouse clicks and process keyboard input. For example, menus on web pages are often done completely in JavaScript. The obvious disadvantage is that there is no standard library of "JavaScript enabled" GUI components for use in web pages.

Using the message-based model in combination with a Java applet provides an easy solution here: The user interface can be created from re-usable Java components and the application logic is connected to the interface through a persistent socket connection that is not limited by the HTTP protocol. In particular, events can be sent to the server without issuing new HTTP requests that would lead to a complete re-rendering of the page.

Interestingly, a similar approach can be seen in the new *Ajax* technology [8] used for example in *Google Mail*. Ajax is an abbreviation for *Asynchronous JavaScript And XML* and tries to solve the asynchronously event handling problem by using XML-RPC instead of HTTP to talk to the application logic on the server:

> Instead of loading a web page, at the start of the session, the browser loads an Ajax engine – written in JavaScript and usually tucked away in a hidden frame. This engine is responsible for both rendering the interface the user sees and communicating with the server on the user's behalf. The Ajax engine allows the user's interaction with the application to happen asynchronously – independent of communication with the server. So the user is never staring at a blank browser window and an hourglass icon, waiting around for the server to do something.
>
> – Jesse James Garrett in "A New Approach to Web Applications" [8]

While using Ajax is a definite improvement in this regard, it forces the application developer to move a significant part of the application (the event handling) into the client side code, so instead of building a single application the code is spread across the JavaScript client and the server side logic. Applying the approach presented here allows for an even more flexible way to build web applications without the need to separate the application into client and server code.

4.10 Summary

This chapter has presented a general architecture that is able to fulfil all of the objectives outlined in section 4.2 on page 42:

By using messages instead of method calls as a communication technique, it provides an inherently thread-safe way to access user interface components from application threads even for multi-threaded applications, without putting special demands on the toolkit or introducing locking at the toolkit level. The same mechanism enables the use of multiple concurrent event loops inside an application with transparent event forwarding, enabling the application to for example handle events originating from separate windows in different threads.

The abstraction enforced by the logical separation between the application and the user interface toolkit can be used as well to support a wide range of possible toolkit implementations, including native and non-native toolkits (examples for this are presented in the following chapter). By offering the same consistent interface to the program, these implementations can be easily replaced without the need to recompile the application. This separation also opens the possibility for introducing a remote display capability for Java programs, based on the same architecture of message passing.

Finally, by applying the same message-based approach to Java applets, a new model for creating web applications arises: Implementing a form of server side applets with a client side user interface.

Chapter 5

Prototype Implementation

5.1 Overview

This chapter describes a prototype implementation of the message-based approach presented in the previous chapter to prove that the ideas presented there are actually viable. It is divided into four major parts: The component API visible to an application, the native toolkit server, the Swing toolkit server and the TCP socket based transport implementation for both. This is followed by presenting an example application and finally a section on the handling of applets. But it will start first with a short look at how the channels used for thread communication can be represented in Java.

5.2 Channels

The buffered *channel* implementation in Java is actually quite straightforward: It is based on a linked list as the internal data structure for the object buffer because it is used as a queue, and an array based list cannot efficiently remove the first element of the list[1]. Care must be taken to properly synchronize all operations on the buffer. The `send()` operation is used to send an object across a channel:

```
public class Channel
{
    private LinkedList queue = new LinkedList();

    public void send (Object object)
    {
        synchronized (this) {
```

[1] An even faster alternative for a limited buffer size would be to use a ring buffer.

```
            queue.addLast(object);
            notify();
        }

        synchronized (Channel.class) {
            Channel.class.notify();
        }
    }
```

Sending an object never blocks and notifies a potential thread that is waiting on this channel. It also needs to notify the channel class to be able to support the `alt()` operation described below. The counterpart for receiving a data object from a channel is the `receive()` operation, it will block until data becomes available if the channel's buffer is empty:

```
public synchronized Object receive () throws InterruptedException
{
    while (queue.size() == 0) {
        wait();
    }

    return queue.removeFirst();
}
```

It is also possible to use the `available()` method to check whether any data is currently available on a channel without blocking the reader. It will return the next object from the channel or `null` if the channel is empty. This effectively works like a non-blocking version of `receive()`:

```
// note: unavailable is indistinguishable from null
public synchronized Object available ()
{
    if (queue.size() == 0)
        return null;

    return queue.removeFirst();
}
```

CSP [12] also introduced the ALT compositional construct, which is used to describe that a process depends on a number of channels and waits for any of them to become ready for either reading or writing (the other operations supported by ALT in CSP are not used here). Alef and Limbo offer this in the form of a new control structure in the language itself, which is of course not an option here, so it is done as a class method of the channel class. The `alt()` operation is in fact very similar to the *select* function in POSIX for waiting on multiple sockets or the corresponding *WaitForMultipleObjects* in the Win32 API. The only tricky aspect is that the monitor construct used for thread synchronization in Java is very limited

when it comes to waiting for multiple resources: The only monitor operation for waiting until a condition is signalled by another thread in the core Java API is the `wait()` method, which releases *a single* monitor and waits for a notification. What would be needed here is way to release *all* monitors and wait until *any* of them is notified. The only way to model this in Java is to either use multiple threads (one waiting on each channel) or by using a well-known object like the `Channel` class itself as a central monitor object that is notified whenever *any* channel changes its state to become ready. The latter variant has been chosen for this implementation.

Since the channels dealt with here are buffered, ALT is really only useful for waiting for a list of channels to become ready for reading, because a write to a channel will never block. The `alt()` operation therefore works on an array of channels and will return the next data that is available from any of them:

```
    public static synchronized Object alt (Channel channels[])
        throws IllegalArgumentException, InterruptedException
    {
        if (channels.length == 0) {
            throw new IllegalArgumentException("empty channel list");
        }

        for (;;) {
            int index;
            Object object;

            for (index = 0; index < channels.length; ++index)
                if ((object = channels[index].available()) != null)
                    return object;

            Channel.class.wait();
        }
    }
```

It is actually possible to use this method to build a subclass of `Channel` that encapsulates a group of channels to make them look like a single channel to the caller: A `receive()` operation will return the first available message received from any of the watched channels and the `send()` operation sends an object to all channels in the group and therefore can act as a simple multiplexer for messages:

```
public class AltChannel extends Channel
{
    private Channel channels[];

    public AltChannel (Channel channels[])
    {
        this.channels = channels;
    }

    public void send (Object object)
```

```
    {
        int index;

        for (index = 0; index != channels.length; ++index) {
            channels[index].send(object);
        }
    }

    public Object receive () throws InterruptedException
    {
        return alt(channels);
    }
}
```

5.3 The GUI Library

The implementation of the graphical user interface library visible to the application consists of four components, which are described in the following sections in turn: The event abstraction, the main loop, the request dispatcher and a set of component classes. In theory it would be possible to make the component API mostly compatible with (possibly a subset of) some existing GUI library for Java, but this has been avoided to gain more freedom in choosing which subset of components to implement for the prototype toolkit server.

5.3.1 Events

Events are generally represented as objects that may carry extra state which is relevant to the particular type of event. The basic event type is the `Event` class that only contains the reference to the event source, similar to the standard class *java.util.EventObject* which forms the base class for the AWT and Swing event classes. The only difference here is that the event knows its type name:

```
public class Event
{
    protected final Object source;
    protected final String type;

    public Event (Object source, String type)
    {
        this.source = source;
        this.type = type;
    }

    public Object getSource ()
    {
```

```
        return source;
    }

    public String getType ()
    {
        return type;
    }
}
```

More specific event types typically use subclasses of this class, as for example mouse events. Using a subclass is always required when the event also conveys some state information that cannot be queried from the source object on demand, i.e. that is not inherent state of the component itself. Common examples of this are mouse events (which include the mouse position and button state) and key events (typed key-code, modifier keys).

5.3.2 Event Handlers

The event handling model follows the general strategy described at the beginning of this thesis in chapter 2.1.4 on page 15, with a few notable differences due to the fact that parallel event handling is supported, these are explained below. Event handler methods can either implement the pre-defined `EventListener` interface, which is – as the name implies – a generic interface suitable for all types of events or they can be arbitrary Java methods that simply follow the method signature for event handlers: A `void` return type and a single parameter of type `Event`:

```
public interface EventListener
{
    public void handleEvent (Event event);
}
```

This is different from the model used in the AWT and Swing, where there are different event listener interfaces for each event type, and the method signature includes the concrete event type. Always using the base event type in the method signature has the advantage that is allows for different event types to be handled by the same handler code. The price for this is that it introduces a small burden on the side of the application programmer, who has to cast the event object to the proper type before accessing specific attributes of the event (except the *source*). This also has the effect of deferring event type checking until runtime. Of course, using separate handlers for distinct event types remains the common case.

As a consequence, there is only one set of methods for installing event handlers of any type, and the events types are differentiated using standardized event names. The two main methods for attaching new event handlers to a component are provided by the `Component` class:

```
void addEventHandler (String type, EventListener target);
void addEventHandler (String type, Object target, String method)
    throws NoSuchMethodException;
```

For example, to add an event handler for a button click, one would use:

```
button.addEventHandler("clicked", anEventHandler);
```

The same method can also be used to capture change events of a slider component:

```
slider.addEventHandler("changed", anEventHandler);
```

Attaching an event handler to a component always (often implicitly) associates the given handler with an instance of a main loop, so that when the event occurs, the handler is invoked from the correct thread which is running the corresponding main loop. This can be done either explicitly by specifying a particular main loop when attaching the event handler or it happens implicitly when this information is not provided:

```
public void addEventHandler (String type, EventListener target)
{
    addEventHandler(type, MainLoop.defaultMainLoop(), target);
}
```

As described in the next section, every thread is assigned a *default main loop* when it first tries to register an event handler without explicitly specifying a main loop, which can always be obtained by a call to `MainLoop.defaultMainLoop()`. So, in the common use case of connecting the event handlers from the same thread that will handle these events later on, specifying a main loop can be omitted.

5.3.3 Encapsulation of the Main Loop

The `MainLoop` class plays a very central role in the event dispatching process: All events are forwarded by the *dispatcher* to the main loop responsible for invoking a given event handler method. To be able to take part in this process, each main loop needs its own `Channel` for receiving `DispatchInfo` objects. The channel to use can be specified when creating the main loop, if desired (otherwise a new channel is created for each main loop):

```
public class MainLoop implements Runnable
{
    private Channel channel;                    // event dispatch channel

    public MainLoop ()
    {
        this(new Channel());
    }
```

```
public MainLoop (Channel channel)
{
    this.channel = channel;
}
```

This class also maintains a *default main loop* separately for each thread, stored in a `ThreadLocal` object. `ThreadLocal` is a standard Java class that allows each thread to store data visible only to the calling thread, so that a thread can store its own private data without disturbing the other threads (normally, data is shared between all threads). The current thread's default main loop can always be obtained by a call to the `defaultMainLoop` method.

```
private static ThreadLocal loop = new ThreadLocal() {
    protected Object initialValue() {
        return new MainLoop();
    }
};

public static MainLoop defaultMainLoop ()
{
    return (MainLoop) loop.get();
}

public void setDefaultMainLoop ()
{
    loop.set(this);
}
```

The `run()` method of the main loop itself is almost trivial: It will repeatedly wait for new `DispatchInfo` objects arriving on its channel and dispatch those to the corresponding handler that is stored inside each `DispatchInfo` object. The loop runs until it is halted by a call to its `terminate()` method:

```
private int level;                          // is MainLoop running?

public void run ()                          // start this MainLoop
{
    int current = ++level;

    while (level >= current) {
        try {
            DispatchInfo info = (DispatchInfo) channel.receive();

            info.dispatchEvent();
        } catch (InterruptedException ex) {
            terminate();
        }
    }
}
```

```
public void terminate ()
{
    if (level > 0) {
        --level;
    }
}
```

To properly support recursive main loop invocations, which may be needed by the event handling for modal windows (as described is section 4.8 on page 52), an internal count of the level of recursion is maintained by the main loop object.

5.3.4 The Request Dispatcher

The job of the request dispatcher in this implementation is twofold, as it combines the roles of the request dispatcher and the event dispatcher: It has to forward any requests arriving on its request channel to the toolkit server, and it is responsible for delivering message replies and events to the appropriate threads that are intended to handle them.

The `Dispatcher` class itself is abstract and functions mainly as a factory for the dispatcher singleton object, which is an instance of one of the concrete `Dispatcher` subclasses. The prototype includes several implementations reflecting the different communication patterns (Java Native Interface, Java method call, remote server process). The actual implementation to use in the application can be selected from the command line when starting the program:

- When the system property *gui.toolkit* is defined, it selects the JNI dispatcher and tries to load a particular native toolkit library, in this example the "`toolkit_gtk`" library:

  ```
  $ java -D gui.toolkit=gtk demo.Calculator
  ```

- When the system property *gui.display* is defined, it selects the TCP dispatcher and specifies the location (i.e. host name and port number) of the toolkit server. In this example, the server is running at port 5000 on the host *suleika*:

  ```
  $ java -D gui.display=suleika:5000 demo.Calculator
  ```

- Finally, if neither the toolkit nor the display is set, the Java dispatcher is used (in combination with the Java toolkit server):

  ```
  $ java demo.Calculator
  ```

The `getDispatcher()` class method always returns a reference to the current dispatcher object, which is as explained above an instance of one of the con-

crete `Dispatcher` subclasses appropriate to the application: `NativeDispatcher`, `JavaDispatcher` and `TCPDispatcher`. Should a specific dispatcher class always be required for an application, it is also possible to directly instantiate one of the provided subclasses.

```
public abstract class Dispatcher extends Thread
{
    private static Dispatcher dispatcher;

    public static synchronized Dispatcher getDispatcher ()
    {
        String display = getProperty("gui.display");
        String toolkit = getProperty("gui.toolkit");

        try {
            if (dispatcher != null)
                return dispatcher;
            else if (display != null)
                return new TCPDispatcher(display);
            else if (toolkit != null)
                return new NativeDispatcher(toolkit);
            else
                return new JavaDispatcher();
        } catch (IOException ex) {
            System.err.println(ex);
            System.exit(1);
        }
    }

    protected Dispatcher ()
    {
        synchronized (Dispatcher.class) {
            dispatcher = this;
        }
    }
```

Because applets are running inside a restricted execution environment, trying to obtain the system properties described above in an applet will result in a `SecutityException`, so a simple wrapper for the standard `getProperty()` method is included here that returns `null` values in case a property cannot be accessed (there is really no other way to find out about this without trying). In fact, these properties will never be set in an applet anyway, so this is just fine.

```
    private static String getProperty (String name)
    {
        try {
            return System.getProperty(name);
        } catch (SecurityException ex) {
            return null;
        }
```

```
    }
```

The dispatcher base class also provides a `getChannel()` method to access the channel on which the dispatcher is willing to receive requests once the dispatcher thread is running:

```
    protected Channel channel = new Channel();

    public Channel getChannel ()
    {
        return channel;
    }
```

5.3.5 Request and Event Messages

As one of the major goals of this approach is to represent the flow of both requests and events as messages, the next aspect described is how this is done in the prototype. Since sending a message across a channel is always an asynchronous operation – the `send()` will complete before the result value, if any, is known – request replies require their own messages, so there are in fact three message types: Request, reply and event, called `EventMessage` here to disambiguate this from the `Event` class described in the previous section.

A *request* is comprised of several pieces of information: The channel where the reply should be sent to (if any), the target object (also known as the message receiver), the message name and the request argument list. The be able to represent any form of parameter types, the argument list is represented as an `Object` array here, which is the common way to store such parameter lists in Java (as is seen in the *Java Reflection API* for example). Additionally, each request is assigned a unique sequence number, which is not strictly required, but helps to detect subtle errors like missing or out-of-order replies:

```
public class Request
{
    private static int seqnum;

    private int number;
    private Channel wait;
    private Object target;
    private String message;
    private Object args[];

    public Request (Channel wait, Object target, String message, Object args[])
    {
        synchronized (Request.class) {
            this.number = ++seqnum;
```

```
        }

        this.wait = wait;
        this.target = target;
        this.message = message;
        this.args = args;
    }
```

The `reply()` method of a request is used to send a result value back to the correct thread that is waiting for the result using the channel stored in the request:

```
    public void reply (Object result)
    {
        if (wait != null)
            wait.send(result);
    }
```

The `Reply` class is much simpler, comprised only of the operation result value and the sequence number of the request that this reply belongs to.

```
public class Reply
{
    private int number;
    private Object value;

    public Reply (int number, Object value)
    {
        this.number = number;
        this.value = value;
    }
}
```

Finally, events generated by the user interface need to be represented as messages as well. These are actually somewhat similar to requests in that they can carry an (optional) argument list, but instead of a target and a message they contain information about the event source and the event type:

```
public class EventMessage
{
    private Object source;
    private String type;
    private Object args[];

    public EventMessage (Object source, String type, Object args[])
    {
        this.source = source;
        this.type = type;
        this.args = args;
    }
```

A unique aspect of events is how they are delivered: Instead of having a specific target object, the event source maintains a list of observer objects per type that are notified whenever an event of the given type originates from the component. To model this, the `EventMessage` object has a `dispatch()` method that obtains the list of event handlers from the component and posts the event (encapsulated inside a `DispatchInfo`) to the channels of each the handler's main loops:

```
public void dispatch ()
{
    Component source = (Component) this.source;
    HandlerInfo handlers[] = source.getEventHandlers(type);
    Event event = getEvent();
    int index;

    for (index = 0; index < handlers.length; ++index) {
        HandlerInfo handler = handlers[index];

        handler.getChannel().send(new DispatchInfo(handler, event));
    }
}
```

5.3.6 User Interface Components

The `Component` class is the base class of all user interface classes in the API visible to the programmer. The class itself is *abstract*, so it cannot be instantiated, yet it provides the necessary infrastructure for its subclasses to create new requests and communicate with the request dispatcher. Moreover, it defines a set of basic methods common to all components, in particular the methods to attach and detach event handlers to components.

Remember that all these components are just acting as proxies for the real native components inside the toolkit library, so the first thing this class has to provide is a transparent mapping between these two layers of objects. To achieve this, the `Component` class maintains a map of all components created by the application (using *weak references* for the reasons explained in section 5.4.5 on page 85). Basically, a *key* is defined for each object and this can be used to lookup a particular component later on if necessary.

In the prototype this is simply the unique object identifier of this object in the Java virtual machine as returned by the `toString()` method of Java's `Object` class. But this choice is really arbitrary: Apart from the premise that this string is unique among all created components and stays constant over time, it is nothing more than an opaque token used to refer to this object in the message protocol. One can think of this as "assigning names" to the components by the frontend.

The `getComponent()` method is used to look up a component given its key:

```
public abstract class Component
{
    private static Map map = new HashMap();

    protected static Component getComponent (String key)
    {
        synchronized (map) {
            WeakReference ref = (WeakReference) map.get(key);

            return ref != null ? (Component) ref.get() : null;
        }
    }
```

Due to the fact that this can be used simultaneously by several threads, all access to the map has to be properly synchronized. Whenever a new component is created, it is inserted into the map and a constructor message is send to the dispatcher:

```
    protected Component (Class type)
    {
        synchronized (map) {
            map.put(toString(), new WeakReference(this));
        }

        dispatchVoidMessage(type.getName(), null);
    }
```

This is the first time that the request dispatching process can actually be seen in action: `dispatchVoidMessage()` creates an *asynchronous* request message from a message name – this is always the type name for constructors – and an argument list, which is empty in this case. The target of the message is implicit, as it is always the component itself, and the message is dispatched immediately.

Once all the references to a component disappear, the `finalize()` method removes this entry from the map and tells the toolkit backend that this component is no longer referenced from the application, using the *gui.Component.unref* message:

```
    protected void finalize ()
    {
        synchronized (map) {
            map.remove(toString());
        }

        dispatchVoidMessage("gui.Component.unref", null);
    }
```

All communication with the dispatcher works by message passing using channels,

of course: As already described in section 5.3.4 on page 63, the dispatcher singleton instance has its own channel for receiving requests, so the `Component` class just has to build a new `Request` object and send it to this channel:

```
private static Channel dispatch_channel =
    Dispatcher.getDispatcher().getChannel();

protected void dispatchMessage (String message, Object args[])
{
    Channel chan = (Channel) channel.get();

    dispatch_channel.send(new Request(chan, this, message, args));
}

protected void dispatchVoidMessage (String message, Object args[])
{
    dispatch_channel.send(new Request(null, this, message, args));
}
```

The difference between `dispatchMessage()` and `dispatchVoidMessage()` is that the former creates a *synchronous* request whereas the latter creates an *asynchronous* request. Synchronous requests are used for methods that expect reply values. Since replies are sent as messages via channels as well, the `Component` class provides a channel on which these may be received. And because multiple threads can be waiting for replies simultaneously, there needs to be one such channel per thread, which again is handled in Java by using a `ThreadLocal` object. The `getResult()` method can then be used to obtain the result of the most recently dispatched synchronous message in the current thread:

```
private static ThreadLocal channel = new ThreadLocal() {
    protected Object initialValue() {
        return new Channel();
    }
};

protected Object getResult ()
{
    try {
        return ((Channel) channel.get()).receive();
    } catch (InterruptedException ex) {
        // this should never happen
        throw new InternalError("interrupt in receive()");
    }
}
```

Another important part of the `Component` class is the set of methods used to add and remove event handlers for components. As has been mentioned before, all event types use these methods to maintain their list of event handlers, unlike the

API used by AWT or Swing, where this functionality is scattered across (and often duplicated in) many GUI component classes in an effort to provide type safety. Handlers are organized by type and stored in separate lists created on demand for each event type:

```
private Map handlers = new HashMap();

private synchronized void addEventHandler (String type, HandlerInfo info)
{
    List list = (List) handlers.get(type);

    if (list == null) {
        list = new ArrayList();
        handlers.put(type, list);
    }

    if (list.size() == 0) {
        dispatchVoidMessage("gui.Component.attach", new Object[] {type});
    }

    list.add(info);
}
```

Removing event handlers works much in the same way as adding them:

```
private synchronized void removeEventHandler (String type, HandlerInfo info)
{
    List list = (List) handlers.get(type);

    if (list == null)
        return;

    list.remove(info);

    if (list.size() == 0) {
        dispatchVoidMessage("gui.Component.detach", new Object[] {type});
    }
}
```

The remainder of the methods of the `Component` class just serve as an example here of how the two message dispatching functions shown above are used inside the method stubs of the other component classes to forward method calls as requests to the toolkit server. Each component method uses these dispatching functions to build a new `Request` object that encapsulates the receiving object, the method identifier and the argument list. The request is then immediately sent to the dispatcher's request channel. In this way, the component classes are equivalent to the client side proxy objects of a distributed object system like RMI or CORBA.

All primitive types of course need to be wrapped into their corresponding Java wrapper classes when dealing with messages:

```
public Component getParent ()
{
    dispatchMessage("gui.Component.getParent", new Object[] {});
    return (Component) getResult();
}

public void setEnabled (boolean enabled)
{
    dispatchVoidMessage("gui.Component.setEnabled",
            new Object[] {new Boolean(enabled)});
}

public boolean isEnabled ()
{
    dispatchMessage("gui.Component.isEnabled", new Object[] {});
    return ((Boolean) getResult()).booleanValue();
}
```

In fact, the construction of the request argument lists could be simplified a lot by using the new syntax for variable argument lists for methods which has been introduced in the JDK 1.5, but the code generally tries to stay compatible with older releases of the JDK, so that is not done here.

5.4 Native Toolkit Server

The native toolkit server is an implementation of the server component of the message-based approach in native C code that runs in the same process as the Java interpreter. Accessing native code from Java is done via the *Java Native Interface* [14], which practically forms a bridge between the Java language and C/C++: It allows Java classes to contain methods that are implemented in C or C++ and it provides a C compatible API to access Java classes, objects and methods from native code. In Java, such methods are declared with the `native` keyword and are missing a method body (similar to abstract methods), the implementation is then provided by a shared library module (DLL) that is loaded at runtime into the Java virtual machine.

5.4.1 The Native Dispatcher

The `NativeDispatcher` class is intended for interfacing with a native toolkit implementation that is assumed to be single-threaded. It consists of two threads: The request dispatcher thread (mainly implemented in Java) and a native event

thread, which is handling all interactions with the toolkit library, including invoking native component methods, listening to native events and sending back message replies. Because one cannot assume that multiple threads may interact with the underlying toolkit's components, it is very important here that only the event thread is visible to the toolkit library. This is not unlike the threading model used in the Swing toolkit where the event thread is both responsible for delivering events and updating the user interface by method calls to the components.

When the native dispatcher is created, it tries to load the toolkit library into the Java VM and starts the two dispatcher threads:

```
public class NativeDispatcher extends Dispatcher
{
    private Request request;

    protected NativeDispatcher (String toolkit)
    {
        System.loadLibrary("toolkit_" + toolkit);

        new EventThread().start();
        start();
    }
```

The dispatcher thread maintains a current request object and notifies the event thread using a native method whenever a new request has arrived on its channel. The event thread then wakes up and processes this request, possibly sending back a reply. This process is described in much more detail in the next section where the GTK implementation for the toolkit server is discussed.

```
    private native void notifyRequest ();

    public void run ()
    {
        try {
            for (;;) {
                Request request = (Request) channel.receive();

                synchronized (this) {
                    while (this.request != null)
                        wait();

                    this.request = request;
                    notifyRequest();
                }
            }
        } catch (InterruptedException ex) {
            // terminate loop
        }
    }
```

The event thread is special in that is has a native `run()` method which runs the toolkit library's event loop. The `getRequest()` method is called from the native code to obtain the next request to process from the dispatcher:

```
public class EventThread extends Thread
{
    protected Request getRequest ()
    {
        Request request;

        synchronized (NativeDispatcher.this) {
            request = NativeDispatcher.this.request;

            NativeDispatcher.this.request = null;
            NativeDispatcher.this.notify();
        }

        return request;
    }

    public native void run ();
}
```

5.4.2 GTK Toolkit Server

This toolkit server is based on the native *GTK toolkit* library, which was originally developed for the GIMP image editing application and the Gnome desktop environment to render its graphical components. The GTK toolkit is a general library for creating graphical user interfaces on Unix/X11 platforms, Windows and Mac OS X (the Mac OS X port is still not completely finished). The library itself is implemented in ANSI C, but it provides a flexible, object-oriented architecture that supports many of the features also found in Java, including objects, classes, inheritance and dynamic binding. There is also a smaller accompanying library called *GLib* that provides the basic data types, the framework for the class and object system, memory management functions and a multitude of abstraction APIs enabling platform independent support for input/output, Unicode string handling, multi-threading and synchronization. This library is used for many of the data types in the native toolkit server.

The native thread which is exclusively interacting with the toolkit functions is the event thread that is created by the native dispatcher when the `NativeDispatcher` is created, as explained in the previous section. The event thread's native `run()` method is running the toolkit's main event loop. When the thread is started, it first initializes the GLib and GTK type system and sets up several internal data

structures. The next step is to add the notification pipe file descriptor as an *I/O watch* to the GTK main loop, which ensures that the request notifications from the dispatcher thread can be handled by the main loop. Finally, the GTK main loop is started:

```
JNIEXPORT void JNICALL
Java_gui_NativeDispatcher_00024EventThread_run (JNIEnv *env, jobject obj)
{
    ...
    callback_data_t data;

    g_type_init();
    gtk_init(&argc, &argv);

    if (!initialized)
        init_environment();

    data.env = env;
    data.obj = obj;
    g_io_add_watch(notify_read, G_IO_IN|G_IO_HUP, io_watch, &data);
    ...
    gtk_main();
}
```

What is this notification pipe doing? To understand this, one has to take a more detailed look at the fundamental problem that needs to be solved here:

The native toolkit has its own event loop, implemented in the toolkit library and often not accessible directly to the programmer (though there are some exceptions to this rule, as noted further on). The actual low-level events like mouse movements and keyboard actions are generated by a display server that is separated from the user program itself. Common examples of this are the X11 server on Unix systems, the Graphics Device Interface on Windows and the WindowServer on Mac OS X. Events are then distributed to the program by some form of inter-process communication like sockets or message queues, and the main loop inside the native toolkit library needs to listen to this source of events. How this is done depends on the operating system's facilities that can be used here, for example, a toolkit based on X11 uses the `select()` or `poll()` system call to wait for readable data on its socket connection to the X11 display server. On Windows this is handled similarly, but instead of a socket, a message queue is used together with the Win32 equivalent of `select()`, the `WaitForMultipleObjects()` function.

This is all fine as long as the display server is the only source of input that the toolkit's main loop needs to listen to. But here the native event loop must be able to handle any incoming requests from the communication layer as well, while still listening to the display server at the same time. All of this has to work without the need to introduce separate threads, which as discussed previously is not desirable. This is actually quite easy to do on a conceptual level, because both `select()` and

`WaitForMultipleObjects()` already support listening to multiple input sources, if only the programmer could influence the parameters passed to these functions. Such a problem is not new, in fact. Every GUI program that needs to read data from a possibly slow data source (like a remote network server) would benefit from being able to wait for new data and new events from the user interface at the same time. Therefore, almost all toolkits in use today actually already provide some means to do this, in one of two ways:

The first method is to provide one's own implementation of the main event loop in place of the default one. For this case there are typically some low-level functions provided by the toolkit to make this as easy as possible. For example, with the Xt toolkit on X11 doing so would look like this:

```
fd = ConnectionNumber(display);

if (select(...) > 0)
{
    XEvent event;

    XtNextEvent(event);
    XtProcessEvent(event);
}
```

This method is of course somewhat cumbersome (as well as less likely to be portable) and just shows that the toolkit implementation in question lacks a proper abstraction of the main loop in the first place[2].

The second and more elegant way to solve this is to merely register a new input source with the toolkit's native event loop, when the toolkit's API supports a way to do this. Waiting for any incoming data can then be handled just fine by the native event loop in parallel with waiting for events from the display server.

Either way, this does not yet completely solve the original problem here: The events are delivered to the dispatcher thread using a Java channel object, and such a channel is obviously not a suitable input source for use in a `select()` operation in the toolkit. It is not really clear which operation the `wait()` method used by the `Channel` implementation while waiting for data on a channel maps to at the operating system level, but on Unix it is most likely one of the synchronization primitives provided by the POSIX thread interface. Alas, none of these can be used as parameters in a `select()` operation either, so another way needs to be found. A different approach would be to use the Unix `kill()` system call to send a signal to the corresponding thread, but POSIX mandates that signals cannot be sent to individual threads inside a process, they always affect *all* threads of a

[2] Xt does have this, in fact. The above code is just used to illustrate the general concept here.

process, which is clearly not desirable for what is needed here. Relying on signals are also less efficient and not portable.

A simpler and actually portable way is to just create an anonymous pipe and use this to send the notification from the dispatcher thread, since a pipe is a valid input source that can be registered with the toolkit's main loop. It is not necessary to pass the actual message data itself through the pipe (though this could be done), since it only serves as a means to wake up the toolkit thread, which can then read the data directly from a variable inside the process address space. This is the approach used by the current GTK backend when running the toolkit server in the same process as the application. The latency introduced by the pipe is not measurable compared to the time it takes to execute the actual operation.

There are of course even more approaches that could be used here, depending on what the toolkit supports: A separate thread could be used to listen for incoming requests and queue their handling into the native event loop. This requires that there is already some support for accessing the event loop from another thread or locking access to the event loop. Of course, this thread must *never* access the GUI components itself, otherwise the condition of permitting only single-threaded access would be violated. A scheme like this is used by the Swing toolkit server, in fact, where there is explicit support for inserting user defined events into the event loop (see section 5.5.2 on page 89). Should none of the methods described above be applicable on a certain platform, one could in theory also use a timer executing in the event thread to actively poll for requests from the communication layer. This method will always work, but it would either introduce delays in the overall responsiveness of the system (if the poll frequency is low) or cause unnecessary overhead (if the poll frequency is too high).

This implementation is relying on an anonymous pipe as mentioned above, which becomes apparent when looking at code of the native `notifyRequest()` method of the dispatcher thread:

```
JNIEXPORT void JNICALL
Java_gui_NativeDispatcher_notifyRequest (JNIEnv *env, jobject obj)
{
    GIOStatus status;
    gsize bytes;

    ...
    status = g_io_channel_write_chars(notify_write, "@", 1, &bytes, NULL);
    if (status == G_IO_STATUS_ERROR) {
        perror("g_io_channel_write_chars");
        gtk_exit(1);
    }
}
```

Whenever a new request arrives at the dispatcher for handling by the toolkit,

this method writes a single byte into the notification pipe in order to inform the GTK main loop that there is a new request to process. The pipe is created in the `init_environment()` function that is called as part of the initialization by the `run()` method shown before, disabling any encoding conversion and buffering for the pipe at the GLib level:

```
static void init_environment ()
{
    int notify_pipe[2];

    if (pipe(notify_pipe)) {
        perror("pipe");
        gtk_exit(1);
    }

    notify_read  = g_io_channel_unix_new(notify_pipe[0]);
    notify_write = g_io_channel_unix_new(notify_pipe[1]);

    g_io_channel_set_encoding(notify_read, NULL, NULL);
    g_io_channel_set_buffered(notify_read, 0);
    g_io_channel_set_encoding(notify_write, NULL, NULL);
    g_io_channel_set_buffered(notify_write, 0);
}
```

Whenever data is written to the pipe by the dispatcher thread, the registered *I/O watch* is invoked from the native event thread, which is responsible for fetching the current event from the dispatcher (using JNI functions), decoding it by replacing the Java objects with data types appropriate for use in C and invoking the actual request handling function `handle_request()` in the toolkit server:

```
static gboolean io_watch (GIOChannel *source, GIOCondition cond, gpointer data)
{
    ...
    status = g_io_channel_read_chars(notify_read, &chr, 1, &bytes, NULL);
    if (status == G_IO_STATUS_ERROR) {
        perror("g_io_channel_read_chars");
        gtk_exit(1);
    }

    class = (*env)->GetObjectClass(env, thread);
    mid = (*env)->GetMethodID(env, class, "getRequest", "()Lgui/Request;");
    request = (*env)->CallObjectMethod(env, thread, mid);
    ...
    array = decode_request(env, request);
    handle_request(array, env, request);
    g_value_array_free(array);

    return TRUE;
}
```

The request handler then looks up the corresponding method in the toolkit interface to call for this request. There are two different cases to consider here: Normal method calls, where the target object is already known to the toolkit server, and constructor calls, where a new native component has to be created and associated with the component proxy visible to the toolkit client. For methods that return a value, the result value is wrapped into a Java object and returned using the `reply()` method of the request via JNI:

```
static void handle_request (GValueArray *request, JNIEnv *env, jobject req_obj)
{
    const char *target = g_value_get_string(g_value_array_get_nth(request, 0));
    const char *method = g_value_get_string(g_value_array_get_nth(request, 1));
    gpointer object = lookup_object(target);

    if (object) {
        GValueArray *args =
            g_value_get_boxed(g_value_array_get_nth(request, 2));
        GValue result = {0};
        call_func_t call_func = lookup_call(method);

        if (call_func) {
            call_func(object, args, &result);

            if (G_IS_VALUE(&result)) {
                jclass req_class = (*env)->GetObjectClass(env, req_obj);
                jmethodID mid = (*env)->GetMethodID(env, req_class, "reply",
                                                    "(Ljava/lang/Object;)V");
                jobject value = encode_value(env, &result);

                (*env)->CallVoidMethod(env, req_obj, mid, value);
            }
        }
    } else {
        init_func_t init_func = lookup_init(method);

        if (init_func) {
            object = init_func();
            register_object(object, target);
        }
    }
}
```

In order to find the correct native component corresponding to a proxy object, the toolkit server maintains a two-way mapping between these objects, managed by the `register_object()` and `unregister_object()` functions. To ensure that the native component is not destroyed by GTK while this table still references it, an explicit reference is created:

```
void register_object (gpointer object, const char *_key)
{
```

```
    char *key = g_strdup(_key);

    g_object_ref(object);
    gtk_object_sink(object);

    g_hash_table_insert(map, key, object);
    g_hash_table_insert(rmap, object, key);
}

void unregister_object (gpointer object)
{
    char *key = (char *) lookup_key(object);

    if (key != NULL) {
        g_hash_table_remove(map, key);
        g_hash_table_remove(rmap, object);

        g_object_unref(object);
        g_free(key);
    }
}
```

In addition to handling the requests, the role of the toolkit server is also to forward native events in the form of event messages to the client. This is done using the server's `post_event()` function, which works analogous to sending back reply values to the toolkit client: It converts the event parameters from C data types into Java objects, creates a new `EventMessage` object for the event and dispatches this via the Java Native Interface:

```
void post_event (gpointer source, const char *event, GValueArray *args)
{
    JNIEnv *env = _env;
    jclass class = (*env)->FindClass(env, "gui/EventMessage");
    jmethodID mid = (*env)->GetMethodID(env, class, "<init>",
        "(Ljava/lang/Object;Ljava/lang/String;[Ljava/lang/Object;)V");
    jobjectArray event_args = NULL;
    jobject event_msg;

    if (args != NULL)
        event_args = encode_value_array(env, args);

    event_msg = (*env)->NewObject(env, class, mid,
                                  encode_component(env, source),
                                  encode_string(env, event),
                                  event_args);

    mid = (*env)->GetMethodID(env, class, "dispatch", "()V");
    (*env)->CallVoidMethod(env, event_msg, mid);
}
```

One can think of the encoding and decoding process mentioned before as a general way of converting Java data structures (objects of different types, arrays of objects) into corresponding native data types, which are represented by the generic `GValue` type in the toolkit server. This type is defined by the GLib library and acts as a wrapper for all of the standard C types. A `Request` object for example consists of a target component, a method name and an argument array, all of which are Java objects even at the JNI level, i.e. references to `jobject` data structures inside the Java virtual machine. These are not practical to work with at the C level, so they must be converted to native C data structures first. The first step in the decoding process is to find out what type of value is stored inside a given Java object through the use of the reflection capabilities of the Java Native Interface.

The `decode_value()` function is used to analyze the type of a Java object: It does the equivalent of an `o.getClass().getName()` method call to obtain the object's class name in order to compare this against a list of known class names (like `Integer` or `String`). If none of these match, it is tested whether it is a `Component` subclass or a Java array. Arrays can be decoded recursively in the same manner. Using an unsupported type at this level results in an error.

```
static void decode_value (JNIEnv *env, GValue *value, jobject object)
{
    jclass component_class = (*env)->FindClass(env, "gui/Component");
    jstring jtype = NULL;
    const char *type = NULL;
    int is_array = 0;

    if (object != NULL) {
        jclass class = (*env)->GetObjectClass(env, object);
        jclass class_class = (*env)->GetObjectClass(env, class);
        jmethodID mid = (*env)->GetMethodID(env, class_class, "getName",
                                            "()Ljava/lang/String;");

        jtype = (*env)->CallObjectMethod(env, class, mid);
        type = (*env)->GetStringUTFChars(env, jtype, NULL);
        mid = (*env)->GetMethodID(env, class_class, "isArray", "()Z");
        is_array = (*env)->CallBooleanMethod(env, class, mid);
    }

    if (object == NULL) {
        decode_pointer(env, value, object);
    } else if (strcmp(type, "java.lang.Boolean") == 0) {
        decode_boolean(env, value, object);
    } else if (strcmp(type, "java.lang.Integer") == 0) {
        decode_long(env, value, object);
    } else if (strcmp(type, "java.lang.Double") == 0) {
        decode_double(env, value, object);
    } else if (strcmp(type, "java.lang.String") == 0) {
        decode_string(env, value, object);
    } else if ((*env)->IsInstanceOf(env, object, component_class)) {
```

```
        decode_component(env, value, object);
    } else if (is_array) {
        decode_array(env, value, object);
    } else {
        fprintf(stderr, "invalid type: %s\n", type);
    }

    if (type != NULL)
        (*env)->ReleaseStringUTFChars(env, jtype, type);
}
```

The individual functions used for decoding the different Java data types are all very similar, so only one of them is shown here to illustrate the general idea: Use JNI functions to read the value from the Java object (in this case an `Integer`) and store this inside a `GValue` structure:

```
static void decode_long (JNIEnv *env, GValue *value, jobject object)
{
    jclass class = (*env)->GetObjectClass(env, object);
    jmethodID mid = (*env)->GetMethodID(env, class, "intValue", "()I");
    long val = (*env)->CallIntMethod(env, object, mid);

    g_value_init(value, G_TYPE_LONG);
    g_value_set_long(value, val);
}
```

The process of encoding the native data types into Java objects suitable for passing back to the JNI works just the other way round: `encode_value()` uses the GLib type macros to analyze which data type is stored in a given `GValue` and calls the matching type encoding function:

```
static jobject encode_value (JNIEnv *env, GValue *value)
{
    if (G_VALUE_HOLDS_POINTER(value))
        return NULL;
    else if (G_VALUE_HOLDS(value, G_TYPE_VALUE_ARRAY))
        return encode_value_array(env, g_value_get_boxed(value));
    else if (G_VALUE_HOLDS_STRING(value))
        return encode_string(env, g_value_get_string(value));
    else if (G_VALUE_HOLDS_BOOLEAN(value))
        return encode_boolean(env, g_value_get_boolean(value));
    else if (G_VALUE_HOLDS_LONG(value))
        return encode_long(env, g_value_get_long(value));
    else if (G_VALUE_HOLDS_DOUBLE(value))
        return encode_double(env, g_value_get_double(value));
    else if (G_VALUE_HOLDS_OBJECT(value))
        return encode_component(env, g_value_get_object(value));
    else
        fputs("invalid type in encode_value()\n", stderr);
```

```
    return NULL;
}
```

As before, the encoding of a C `long` value into an `Integer` object is shown as an example here:

```
static jobject encode_long (JNIEnv *env, long value)
{
    jclass class = (*env)->FindClass(env, "java/lang/Integer");
    jmethodID mid = (*env)->GetMethodID(env, class, "<init>", "(I)V");

    return (*env)->NewObject(env, class, mid, (jint) value);
}
```

5.4.3 Native Event Listener

Events in GTK are actually just one application of a more general notification system called *signals*, which is why the functions used for event handling often refer to the term *signal* instead of *event*. Event handling in GTK is done by connecting signal handlers for specific events to a component, just like event listeners are added to a component in the AWT or Swing API in Java. Because C is not an object-oriented programming language, the handlers work in terms of functions, not objects, i.e. a callback function is registered for an event instead of an event listener object. The events themselves are represented by the `GdkEvent` data type in the GTK library.

The general role of the native event listener was explained in the previous chapter: It acts as a generic event listener that is capable of receiving arbitrary event types and handles these by wrapping them into event messages that are sent to the event dispatcher in the toolkit client. In this implementation, the wrapping of events is already handled by the server's `post_event()` function described above, so the event listener just has to forward events to this function. For simple events that do not require any event arguments, the native event handler code can just call the `post_event()` function with the source object and the event type:

```
static void generic_event (GtkWidget *self, gpointer data)
{
    post_event(self, data, NULL);
}
```

Not all event listeners can be that simple, however. If further arguments need to be included into the event message, these have to be copied from the native event

structure into a value array (an array of `GValue` structures) that forms the event message argument list. This is for example done for the mouse motion listener:

```
static gboolean motion_event (GtkWidget *self, GdkEventMotion *event,
                              gpointer data)
{
    GValueArray *args = g_value_array_new(2);
    GValue value = {0};

    g_value_init(&value, G_TYPE_LONG);
    g_value_set_long(&value, event->x);
    g_value_array_append(args, &value);
    g_value_set_long(&value, event->y);
    g_value_array_append(args, &value);
    g_value_unset(&value);

    post_event(self, data, args);
    return FALSE;
}
```

Attaching and detaching event listeners to and from components just maps to the corresponding GTK functions for connecting and disconnecting signal handlers:

```
static void attach_clicked (GObject *self, const char *event)
{
    g_signal_connect(self, "clicked", G_CALLBACK(generic_event),
                     (gpointer) event);
}

static void detach_generic (GObject *self, const char *event)
{
    g_signal_handlers_disconnect_by_func(self,
        G_CALLBACK(generic_event), (gpointer) event);
}
```

To support any number of event types, the event listener contains a table of known types and the corresponding functions for attaching and detaching event handlers of that type on a component. These are stored in a hash table at program startup for faster access and can be looked up by event type name using the `lookup_signal()` function:

```
static signal_map_t signals[] = {
    { "activate",       attach_activate,        detach_generic },
    { "changed",        attach_changed,         detach_generic },
    { "clicked",        attach_clicked,         detach_generic },
    ...
};

void init_signal_map (void)
{
```

```
    int index;

    signal_map = g_hash_table_new(g_str_hash, g_str_equal);

    for (index = 0; index < G_N_ELEMENTS(signals); ++index) {
        const char *event = signals[index].event;

        g_hash_table_insert(signal_map, (char *) event, &signals[index]);
    }
}

signal_map_t *lookup_signal (const char *event)
{
    return g_hash_table_lookup(signal_map, event);
}
```

5.4.4 Native Components

The native component implementation of the toolkit interface that was described in section 4.6.5 on page 49 consists of a rather large collection of individual functions, each corresponding to one method of a component proxy that may be called by the application. It is basically a thin adaptor layer between the abstract GUI component interface presented to the application and the capabilities offered by the native library on which the implementation is based. Using a native toolkit library that follows an object-oriented approach here can make these adaptors a lot easier to implement, but that is in no way a requirement. Most functions used here just map directly to functions of the underlying toolkit in the case of GTK.

To illustrate this point, some of the functions from the implementation of the `Window` component in the toolkit server are presented here just as an example. Creating a new window simply maps to the `gtk_window_new()` function:

```
gpointer gui_window_new (void)
{
    GtkWidget *result = gtk_window_new(GTK_WINDOW_TOPLEVEL);

    return result;
}
```

Note that the function shown above is a constructor function, so it does note take any arguments (this is a convention used throughout the prototype) and always returns a component as the result value. Normal functions get their arguments in the form of a `GValue` array, as can be seen e.g. in the `set_size()` function:

```
void gui_window_set_size (gpointer self, GValueArray *args, GValue *result)
{
    int width = g_value_get_long(&args->values[0]);
```

```
    int height = g_value_get_long(&args->values[1]);

    gtk_window_resize(self, width, height);
}
```

The GTK toolkit also supports component properties very similar to the *Bean properties* used in Java, which can be set using `g_object_set_property()`:

```
void gui_window_set_visible (gpointer self, GValueArray *args, GValue *result)
{
    g_object_set_property(self, "visible", &args->values[0]);
}
```

Returning values is done likewise using a `GValue` structure that can hold arbitrary data types. In this case, it is used to return a C string value owned by the component:

```
void gui_window_get_title (gpointer self, GValueArray *args, GValue *result)
{
    const char *value = gtk_window_get_title(self);

    g_value_init(result, G_TYPE_STRING);
    g_value_set_static_string(result, value);
}
```

5.4.5 Object Ownership and Garbage Collection

Another area that needs to be considered is the problem of object ownership in such a distributed object model. It is almost guaranteed that the Java runtime and the GUI library have different concepts of object ownership, making the question of when an object (and here in particular a graphical component) is no longer in use and can be destroyed by the library and garbage collected by the Java runtime, respectively. The evaluate this, taking a closer look at how object lifetime is handled in different systems is necessary:

A real garbage collected memory management system requires an automatic way to determine when new references to an object are created and when existing references go away. This is generally only possible in higher level languages that do not allow direct (i.e. uncontrolled by the runtime) access to variables and data structures. Java is one example of this, as are in fact most interpreted languages[3]

[3] This includes programming languages that are compiled to byte-code internally like e.g. Perl and Python.

Over time, many different ways to implement garbage collected memory management systems have been proposed, but the most common ones in use today are:

- Reference Counting

  Reference counting inserts a counter (as the name implies) into each managed object that tracks the number of "live" external references to this object. When a new reference is created, for example by inserting the object into a container – in GUI terms this may correspond to adding a button to a window – the reference counter is incremented. Likewise, when an existing reference goes away, the counter is decremented. Once the counter reaches zero, the object is destroyed.

  Such a scheme is fairly straightforward to implement and very efficient, but it has the fundamental limitation that it cannot (by its very design) correctly track object lifetime for circular references: If a set of objects contains a reference cycle, it will never be destroyed because all objects in the cycle keep each other alive. Such a problem needs to be resolved by either using a weak reference to break such a cycle or by using a different mechanism to detect such cases.

- Mark and Sweep

  A mark and sweep collector is based on the idea that each object can be determined to be either reachable or not after traversing all object references in the program. The basic requirements here are that it must be possible to reliably identify a reference (i.e. there must be no way to "hide" a reference by for example storing it inside an integer variable) and there must be a way to iterate efficiently over all starting objects for the mark algorithm, basically all static fields (i.e. class variables) in classes and all objects on the stack (local variables). The runtime system must also be able to somehow enumerate all the objects that are currently alive.

  The algorithm works by traversing the graph of all objects reachable by either of these starting objects and marking each object as "seen" when it has been touched. In a second step, it iterates over all objects in the system and discards all of them that are no longer reachable.

- Object Copying

  A copying collector maintains a list of potentially reachable objects that is updated from time to time by copying all currently reachable objects to a new list and discarding the old list. This is an attempt to avoid memory fragmentation at the cost of the extra copying operations. However, such a scheme is generally only possible when it is possible to move an object around freely in memory and update any references to it from other objects.

Some systems use a combination of *reference counting* with another scheme, often a *mark and sweep* or a *copying* collector, in order to catch reference cycles. The Java runtime does this as well. The motivation for this in Java was that reference counting is fast and covers most cases rather well. This is all fine as long as only one system is involved, but gets a lot more difficult when objects may be distributed among several processes. In the architecture described here, there is fortunately one factors that helps to significantly reduce the complexity:

The *only* references that actually cross the boundaries between the Java virtual machine and the GUI library are the internally maintained references between a Java component proxy and its counterpart in the toolkit library, and these are completely under the control of the toolkit client and server, respectively. There is no way for the application to create references to other objects in the toolkit server. The way to handle this problem is to use so-called *weak references* provided by the `java.lang.ref` package for all references that cross this boundary in Java. A weak reference is a reference to an object that does not prevent the Java garbage collector from considering the referenced object for finalization. The actual term used in Java is that an object is eligible for finalization when it is *weakly reachable*, and an object is said to be *weakly reachable* if it is *only* reachable through one or more weak references.

This means that the lifetime of the proxy object is tied to the lifetime of the actual component is such a way that it keeps the component alive but not vice versa. The typical life cycle of a graphical component actually looks like this:

- The application creates a new component (which is in fact a proxy object):

  ```
  Button b = new Button("Click Me");
  ```

- The proxy registers a weak reference to itself in a global lookup table to allow it to be found later on when a reference to it is passed back as part of an event. The identifier used here is a string that uniquely identifies the object (the current implementation derives this from the internal object identifier used by the Java virtual machine).

- The proxy generates a constructor request message for its corresponding type and sends this via the dispatcher to the toolkit server.

- The toolkit server instantiates the real component object and registers it in a lookup table on the server side. Further messages send to the same object will retrieve this object from the lookup table. This time, a strong (i.e. non-weak) reference is used, which ensures that the GUI server will not dispose the object as long as the client has a valid reference to it.

- Once the last strong reference to the client side proxy disappears, the proxy object is notified via its `finalize()` method and will notify the component

it represents in the server of this. Then it will delete itself from the client side lookup table and is ready for finalization.

- The toolkit server will remove the corresponding component from its lookup table on the server and release its reference to it. It then checks whether the corresponding component has any strong references left on the server, this will the case for example when the button has been inserted into a layout container. If there are no references (i.e. the component is "un-owned" on the server as well), the component will be destroyed.

The concrete implementation of course depends on the capabilities of the toolkit library used on the server side.

5.5 Java Toolkit Server

To show that the general architecture presented here is not in any way tied to native toolkits or GTK in particular, and that it is in fact capable of supporting a wide range of toolkits, an alternative toolkit server has been implemented in Java that can be used in combination with a Java dispatcher class. Building a Java toolkit server may sound strange at first, because it just seems to wrap one Java component API into another, yet there is a big difference between the toolkit used for implementing components in the server and the view of the components provided to the application. In fact, all the benefits of the native toolkit server apply in combination with the Swing-based server as well:

- Accessing the user interface components is fully thread-safe and supports parallel event handling.
- The component API is not that of Swing, but rather the consistent abstraction also used for the GTK components, so the backends can be exchanged transparently.
- When using a TCP socket connection to the toolkit server, the application's user interface can be displayed on a different system.

The following sections describe this toolkit server based on Swing components.

5.5.1 The Java Dispatcher

The implementation of the `JavaDispatcher` is really straightforward: Because it is only used when the toolkit server is running in the same Java virtual machine as the dispatcher, the Java dispatcher can hold a direct reference to the server object (which in fact will be an instance of the `JavaServer` class):

```
public class JavaDispatcher extends Dispatcher
{
    private Server server = Server.getServer();

    protected JavaDispatcher ()
    {
        start();
    }
```

The dispatcher runs in its own thread and simply forwards any incoming requests synchronously to the toolkit server. The handling of message replies and events is done completely by the `JavaServer`, because the AWT event thread here takes the role of the event thread that was used in the case of the native dispatcher. This is described in more detail in the next section.

```
    public void run ()
    {
        try {
            for (;;) {
                Request request = (Request) channel.receive();

                server.handleRequest(request);
            }
        } catch (InterruptedException ex) {
            // terminate loop
        }
    }
}
```

5.5.2 Swing Toolkit Server

On the toolkit server side, the general architecture is similar to that used for the dispatcher on the client side: The actual server is a singleton object that is an instance of a concrete subclass of the abstract `Server` class. The class method `Server.getServer()` will always return the server singleton, creating one if necessary.

```
public abstract class Server
{
    private static Server server;

    public static synchronized Server getServer ()
    {
        if (server != null)
            return server;
        else
            return new JavaServer();
    }
```

```
protected boolean debug;
protected boolean sync;

protected Server ()
{
    synchronized (Server.class) {
        server = this;
    }

    debug = "true".equals(getProperty("gui.debug"));
    sync  = "true".equals(getProperty("gui.sync"));
}
```

The Java system properties *gui.debug* and *gui.sync* can be set for the server to enable request logging (a debugging aid) and to disable buffering of the toolkit's graphics state (used for timing tests), respectively. Like already explained for the dispatcher, trying to access system properties from an applet will result in a `SecutityException`, so the same simple wrapper is used here as well:

```
private static String getProperty (String name)
{
    try {
        return System.getProperty(name);
    } catch (SecurityException ex) {
        return null;
    }
}
```

There are two methods that are left as `abstract` in the `Server` class: `postEvent()` and `handleRequest()`. `postEvent()` is used in the same way as the function of the same name in the native toolkit server: It allows the event listener to post event messages to the event dispatcher in the client. The `handleRequest()` method is called by the Java dispatcher to deliver requests to the toolkit server:

```
public abstract void postEvent (EventMessage message);
public abstract void handleRequest (Request request);
```

Additionally, the toolkit server needs to know whether it is running on behalf of an application or an applet, because in the latter case it is responsible for loading and initializing the applet class. To support this, the `loadApplet()` function is provided in the base class. In the case of an applet, the applet parameters are used instead of system properties:

```
public void loadApplet (JApplet applet, String name)
{
    this.applet = applet;
```

```
        debug = "true".equals(applet.getParameter("gui.debug"));
        sync  = "true".equals(applet.getParameter("gui.sync"));
    }
```

Just like the native toolkit server implementation, the Java toolkit server has to maintain a mapping of client side component proxies to the Swing components and vice versa. This can be handled completely in the base class:

```
    private Map map = new HashMap();
    private Map rmap = new HashMap();

    protected Object lookupObject (Object key)
    {
        return map.get(key);
    }

    protected Object lookupKey (Object object)
    {
        return rmap.get(object);
    }
```

These lookup tables are maintained by the two methods `registerObject()` and `unregisterObject()`: Whenever a new components is instantiated by the client, the server creates the corresponding Swing component and stores a new two-way mapping:

```
    protected void registerObject (Object object, Object key)
    {
        map.put(key, object);
        rmap.put(object, key);

        if (object == applet) {
            synchronized (applet) {
                applet.notify();
            }
        }
    }

    protected void unregisterObject (Object object)
    {
        Object key = lookupKey(object);

        if (key != null) {
            map.remove(key);
            rmap.remove(object);
        }
    }
```

Applets require a special case here, however, because the logic for an applet is different: Normally, the proxy component is always created first inside the applet and the Swing component thereafter. But this cannot work for applets because the Swing component for an applet is already there at the very start of the applet, so that the server does not really know whether the applet component proxy on the client side has already been created or not. Waiting for the notification generated here solves this problem.

The concrete server implementation that is used for the Swing toolkit is the `JavaServer` class. It inherits the basic methods to register its components from the `Server` class and has to implement the `handleRequest()` and `postEvent()` methods. `handleRequest()` is equivalent to the `handle_request()` function of the native toolkit server and in fact looks almost identical. It decodes the request message and invokes the corresponding method on the component or creates a new component:

```
public void handleRequest (Request request)
{
    Object target = request.getTarget();
    String method = request.getMessage();
    Object object = lookupObject(target);

    if (object != null) {
        Object args[] = (Object[]) decodeObject(request.getArguments());
        Method m_call = methods.lookupCall(method);

        if (m_call != null) {
            invokeRequest(request, m_call, new Object[] {object, args});
        }
    } else {
        Method m_init = methods.lookupInit(method);

        if (m_init != null) {
            try {
                object = m_init.invoke(null, new Object[] {});
                registerObject(object, target);
            } catch (InvocationTargetException ex) {
                ...
            }
        }
    }
}
```

There is one important difference here compared to the native server, however: Because `handleRequest()` itself is called from the request dispatcher thread, the actual method call to the component cannot be made directly. Only the AWT/Swing event thread may interact with the components, so the method call has to be pushed into the event thread using the `invokeLater()` method of the

`java.awt.EventQueue` class. The result of the operation is also sent back from this thread:

```
private void invokeRequest (final Request request, final Method method,
                            final Object args[])
{
    EventQueue.invokeLater(new Runnable() {
        public void run () {
            try {
                Object result = method.invoke(null, args);

                if (result != JavaGUI.VOID) {
                    request.reply(encodeObject(result));
                }
            } catch (InvocationTargetException ex) {
                System.err.println(ex.getCause());
            } catch (IllegalAccessException ex) {
                System.err.println(ex);
            }
        }
    });
}
```

The `postEvent()` implementation is really simple because it can directly dispatch the message to the toolkit client's main loops (remember that `postEvent` is always called from the event thread). The only special thing that needs to be done here is translating the source component into the corresponding component proxy for the client:

```
public void postEvent (EventMessage message)
{
    message.setSource(lookupKey(message.getSource()));
    message.dispatch();
}
```

Finally, the java server is also responsible for loading the client applet class when it is used inside an applet. The general strategy used for running applets in the Swing server consists of several steps:

1. The Swing applet proxy is loaded into the browser and invokes the toolkit server's `loadApplet()` method to start loading the applet. This applet proxy is a subclass of *javax.swing.JApplet* as this is an absolute requirement for applets. As the name implies, it just acts as a proxy for the real applet implementation that is used as the toolkit client and mainly forwards operations like `init()` and `start()` to the client applet.

2. The toolkit server arranges loading of the client applet using the Java reflection API and creates a thread for the applet's initial main loop. The server

then waits until the client applet is fully initialized.

3. The applet proxy sends the "applet-init" pseudo event to the client to indicate that the applet can now start to run.

The first step in this process is handled by the web browser and the third by the applet proxy implementation shown in section 5.8.1 on page 118. The second step is the job of the toolkit server and is reflected in its `loadApplet()` method:

```
public void loadApplet (JApplet applet, String name)
{
    try {
        super.loadApplet(applet, name);
        Class applet_class = Class.forName(name);
        Applet component = (Applet) applet_class.newInstance();
        Thread thread = new Thread(component);

        synchronized (applet) {
            thread.start();

            while (lookupKey(applet) == null) {
                applet.wait();
            }
        }
    } catch (ClassNotFoundException ex) {
        ...
    }
}
```

5.5.3 Swing Event Listener

The generic event listener for the Swing toolkit server is the `EventListener` class, and it implements the listener interfaces for all supported event types, so that instances of this class can be used to handle any kind of event:

```
public class EventListener implements
        ActionListener, ChangeListener,
        MouseMotionListener, WindowListener
        ...
{
    private String type;

    public EventListener (String type)
    {
        this.type = type;
    }
```

Each event that is received by this listener is wrapped into an event message and

sent to the event dispatcher in the client. To avoid making this listener class depend on the communication mechanism used by the server, the event messages are passed to the toolkit server's `postEvent()` message for transport to the client. The listener methods for the basic event types do not require any special code:

```
public void actionPerformed (ActionEvent event)
{
    Object source = event.getSource();

    server.postEvent(new EventMessage(source, type));
}
```

In the case of events that want to deliver arguments to the client these have to be copied into the event message out of Swing event object. This is in fact exactly the same idea already seen before when looking at the native event listener. A good example here is again the mouse motion listener:

```
public void mouseMoved (MouseEvent event)
{
    Object source = event.getSource();
    Object args[] = new Object[] {
        new Integer(event.getX()), new Integer(event.getY())
    };

    server.postEvent(new EventMessage(source, type, args));
}
```

Attaching an event listener to a Swing component then involves finding the correct *add...Listener()* method for the type of event that this listener has been created for, because there is no single method for attaching event handlers as provided in GTK for example:

```
public void attach (Component component)
{
    if (type.equals("activate"))
        ((JTextField) component).addActionListener(this);
    else if (type.equals("clicked"))
        ((AbstractButton) component).addActionListener(this);
    else if (type.equals("closing"))
        ((Window) component).addWindowListener(this);
    else if (type.equals("changed"))
        ((JSlider) component).addChangeListener(this);
    else if (type.equals("mouse-motion"))
        ((Component) component).addMouseMotionListener(this);
    else if (type.startsWith("applet-"))
        /* ignore applet pseudo-events */;
    else
        System.err.println("unknown event: " + type);
}
```

Detaching an event listener works in exactly the same way as attaching it, the only difference is that the corresponding *remove...Listener()* method is used instead.

5.5.4 Swing Components

The approach used for the implementation of the components in the Swing toolkit server is quite similar to the one used for the GTK toolkit server: Each operation is mapped onto a corresponding method of a Java class again using a method table:

```
public class JavaMethods
{
    public JavaMethods ()
    {
        Class init_types[] = new Class[] {};
        Class call_types[] = new Class[] {Object.class, Object[].class};
        int index;

        String init_methods[][] = {
            { "gui.Applet",             "gui_applet_new" },
            { "gui.Button",             "gui_button_new" },
            { "gui.Grid",               "gui_grid_new" },
            ...
        };

        String gui_methods[][] = {
            { "gui.Box.add",            "gui_box_add" },
            { "gui.Button.getText",     "gui_button_get_text" },
            { "gui.Button.setText",     "gui_button_set_text" },
            ...
        };

        ...
    }
```

This mapping is stored in a hash table for efficiency and the methods `lookupInit()` and `lookupCall()` of the `JavaMethods` class are then used to locate the appropriate method corresponding to the name of a method that appears in a request. `lookupInit()` is used specifically for constructors, `lookupCall()` for everything else (i.e. normal methods):

```
    public Method lookupInit (String name)
    {
        return (Method) initMethods.get(name);
    }

    public Method lookupCall (String name)
```

```
    {
        return (Method) guiMethods.get(name);
    }
}
```

The implementation of the individual methods for each component is now very similar to the one already seen for the native components before in section 5.4.4 on page 84. Again, these methods have to implement the abstract component interface offered to the application in terms of the methods provided by the toolkit used in the server, in this case the Swing API.

Once more, two methods from the Swing version of the toolkit interface for the `Window` class are shown here to illustrate that: The constructor used to create new windows and the code used for the `setSize()` method:

```
public static Object gui_window_new ()
{
    return new JFrame();
}

public static Object gui_window_set_size (Object _this, Object args[])
{
    JFrame self = (JFrame) _this;
    int width = ((Integer) args[0]).intValue();
    int height = ((Integer) args[1]).intValue();

    self.setSize(width, height);
    return VOID;
}
```

5.6 Distributed Communication

Using the message-based approach makes it possible to move from a model where the application and the toolkit both run in the same process to a truly distributed model with minimal effort. Here, the toolkit server forms a generic "GUI server" for applications that runs as standalone program. When the toolkit backend is not running in the same process as the application itself, it is of course no longer viable to directly call methods inside the toolkit server. Instead, an approach similar to *Remote Procedure Calls* or Java's *Remote Method Invocation* needs to be applied to the communication between the toolkit client and the server:

All request, reply and event messages need to be serialized upon sending into a byte stream suitable for transport over e.g. a TCP socket, and it must be possible to reconstruct these messages again from their serialized representation at the other end of the network connection.

5.6.1 The Remote Dispatcher

The remote dispatcher implementation allows the request and event messages to be passed transparently across a TCP network connection to a different machine, completely decoupling the application threads in the client from the user interface running in the toolkit server. Apart from using this dispatcher (which can be selected using a Java system property), no other changes are required in the client. The `Dispatcher` subclass on the client side suitable for communicating across TCP sockets with a toolkit server is the `TCPDispatcher` class. Of course, a similar implementation could be done for *Unix domain sockets* or *named pipes* if desired. The `TCPDispatcher` needs to be initialized with a description of the server address:

```
public class TCPDispatcher extends Dispatcher
{
    private List requests;
    private Socket server;
    ...

    protected TCPDispatcher (String display_name) throws IOException
    {
        Display display = new Display(display_name);
        Socket server = new Socket(display.getHost(), display.getPort());

        init(server);

        new EventThread().start();
        start();
    }
```

Once the socket connection is established, the input and output streams used for the communication are set up and a `CodingDelegate` is prepared. As explained in section 5.6.2, the prototype uses a very simple text based protocol for the message exchange, so the streams actually are character streams, not byte streams as one might expect. The job of the `CodingDelegate` is explained later:

```
    private void init (Socket server) throws IOException
    {
        InputStream in = server.getInputStream();
        OutputStream out = server.getOutputStream();

        this.server = server;
        display_in = new BufferedReader(new InputStreamReader(in, "UTF-8"));
        display_out = new BufferedWriter(new OutputStreamWriter(out, "UTF-8"));
        delegate = new CodingDelegate();

        requests = new LinkedList();
    }
```

The `run()` method of the remote dispatcher is now very similar to the one already described before for the native dispatcher, the only differences here are that now a request queue is maintained of outstanding requests for which no reply has been received yet (this is used to match incoming replies to the corresponding request objects), and that requests are sent across the network instead of executing them in the same process:

```
public void run ()
{
    try {
        for (;;) {
            Request request = (Request) channel.receive();

            if (request.hasResult()) {
                synchronized (requests) {
                    requests.add(request);
                }
            }

            display_out.write(request.toString(delegate));
            display_out.newLine();
            display_out.flush();
        }
    } catch (InterruptedException ex) {
        // terminate loop
    } catch (IOException ex) {
        System.err.println(ex);
    }
}
```

Again, all events and request replies are handled in their own thread. For each incoming message it first needs to be determined whether this is an event or the reply to the oldest unanswered request (requests are always processed in order by the server). Each Event is then decoded by the `EventMessage` class and dispatched locally to the appropriate main loops. A reply is decoded likewise by the `Reply` class and the result value is passed on to the thread that sent the original request, using the channel specified in the request object:

```
public class EventThread extends Thread
{
    public void run ()
    {
        ...
            while ((line = display_in.readLine()) != null) {
                if (EventMessage.isEventMessage(line)) {
                    EventMessage msg = new EventMessage(line, delegate);

                    msg.dispatch();
                } else {
```

```
                        Reply reply = new Reply(line, delegate);
                        Request request;

                        synchronized (requests) {
                            request = (Request) requests.remove(0);
                        }

                        if (!request.matches(reply.getNumber()))
                            throw new InternalError("unknown sequence number");

                        request.reply(reply.getValue());
                    }
                }
            ...
        }
    }
```

The reason why the event and reply handling needs to done in its own thread here is that it is simply impossible in Java for a thread to simultaneously wait for messages arriving at a channel (which is internally just a call to the `wait()` synchronization method) and to wait for input from an I/O stream. So there is no real choice but to separate these two wait operations into two threads.

5.6.2 Communication Protocol

For the distributed case, the dispatcher and the toolkit server have to exchange requests, request responses and events across a network connection. To be able to do that, it is first necessary to define a protocol that is used for this message exchange. The prototype implementation described here uses a simple text based protocol for this where each message corresponds to one line of text, the main motivation being here that text is easy to monitor and debug if required. It also does no require thinking about byte ordering on different hardware.

All message parameters are converted to Unicode strings, which are then encoded in UTF-8 for transport across the communication layer. While this is sufficient for a testing prototype, a compact binary protocol would certainly be preferable here in the long run to achieve better performance at this level. Using a binary protocol would for example completely avoid the need to convert integer and floating point values to textual form and later on parsing this text to recover the original value on the receiver's side. A format similar to XDR [24] would be feasible for this, which is used as the general encoding format for *Remote Procedure Calls* [17]. Another option would be to adopt the object serialization format used by the Java *Remote Method Invocation* API, which would have the additional benefit that the Java encoder/decoder comes for free, but no such encoder or decoder exists for native C programs yet.

```
<request> ::= <seqnum> , <reference> , <method> , <value> \n

<reply>   ::= <seqnum> , <value> \n

<event>   ::= <reference> , <type> , <value> \n

<seqnum>  ::= <integer>

<method>  ::= <string>

<type>    ::= <string>

<value>   ::= <null>      |
              <boolean>   | <integer>   | <float>     |
              <reference> | <string>    | <array>

<array>   ::= { }         | { <values> }

<values>  ::= <value>     | <value> , <values>

<null>    ::= *
```

Figure 5.1: BNF definition of simple text protocol

An incomplete formal definition of the text protocol syntax in *Backus-Naur Form* is given in figure 5.1.[4] It is incomplete because it does not include the definition for the non-terminals (basic types) `<boolean>`, `<integer>`, `<float>`, `<reference>` and `<string>`, which are easier described informally in the following list:

`<boolean>` A boolean value is represented as one of the constants "`0`" (false) or "`1`" (true), prefixed with the character "`b`" to indicate the type.

`<integer>` Integer values use the standard decimal string representation of their type, prefixed with the character "`i`" to indicate the type.

`<float>` Likewise, this is the standard string representation of an IEEE 754 double precision floating point value, prefixed with the character "`d`" to indicate the type.

`<reference>` A reference is an opaque string identifier referring to a client side object (as described before) enclosed in single quotes. References must only contain ASCII characters.

`<string>` Strings are always transferred as UTF-8 encoded byte sequences enclosed in double quotes, with double quotes and the backslash character inside strings escaped with a backslash character (as done in Java or C).

[4] Note that this is BNF, not extended BNF, so the curly brackets are terminal symbols, not meta symbols.

To illustrate how the protocol messages look like in practice, figure 5.2 shows a short part of the message flow for a simple demo application[5]. Versions of the protocol encoder and decoder exist for Java and C (both versions use a simple recursive descend parser) and could be combined with different transports like sockets or names pipes as required.

```
-->  i13,"gui.Slider@173a10f","gui.Slider.setMinimum",{d0}
-->  i14,"gui.Slider@173a10f","gui.Slider.setMaximum",{d100}
-->  i15,"gui.Box@a62fc3","gui.Container.add",{'gui.Slider@173a10f'}
-->  i16,"gui.Box@a62fc3","gui.Container.add",{'gui.TextField@10b30a7'}
-->  i17,"gui.Box@a62fc3","gui.Container.add",{'gui.Button@1a758cb'}
-->  i18,"gui.Window@14318bb","gui.Container.add",{'gui.Box@a62fc3'}
-->  i19,"gui.Window@14318bb","gui.Window.setVisible",{b1}
-->  i20,"gui.Window@14318bb","gui.Window.getTitle",{}
<--  i20,"Hello World"
-->  i21,"gui.Button@1a758cb","gui.Button.getText",{}
<--  i21,"Click Me!"
-->  i22,"gui.TextField@10b30a7","gui.TextField.getText",{}
<--  i22,"Some Text..."
-->  i23,"gui.Window@14318bb","gui.Window.setTitle",{"Click"}
<--  'gui.Button@1a758cb',"clicked",*
<--  'gui.Slider@173a10f',"changed",*
-->  i24,"gui.Slider@173a10f","gui.Slider.getValue",{}
<--  i24,d1.65289
-->  i25,"gui.TextField@10b30a7","gui.TextField.setText",{"1"}
```

Figure 5.2: Communication fragment of simple text protocol

5.6.3 Encoding Messages for Transport

The encoding and decoding of the different data types used in Java for the messages is performed with the help of the `Encoder` and `Decoder` classes. These contain methods that implement the necessary steps for most of the basic data types supported in the protocol: *Boolean*, *Integer*, *Double*, *String* and *Array*.

Encoder

The `Encoder` encapsulates a string buffer that holds the encoded data and supports the use of an optional *encoding delegate* that is asked to encode any object whose type the `Encoder` cannot handle. Such delegates are used in the prototype to determine how components should be encoded, although the general mechanism could be used for other types as well.

[5] The directional arrows before each line have been added for improved readability, they are not part of the transmitted messages.

The central method of the `Encoder` class is the `encodeObject()` method, which automatically chooses an appropriate encoding method depending on the object's type. Java arrays are encoded by recursively invoking this method for each object inside the array:

```
public void encodeObject (Object object) throws CodingException
{
    if (object == null)
        buffer.append('*');
    else if (object.getClass().isArray())
        encodeArray((Object []) object);
    else if (object instanceof String)
        encodeString((String) object);
    else if (object instanceof Boolean)
        encodeBoolean((Boolean) object);
    else if (object instanceof Integer)
        encodeInteger((Integer) object);
    else if (object instanceof Double)
        encodeDouble((Double) object);
    else if (delegate != null)
        encodeComponent(object);
    else
        throw new CodingException("no encoding delegate set");
}
```

Decoder

The `Decoder` class is the counterpart to the `Encoder` and is used to recreate the Java data types from their textual representation. The type prefixes for the values defined at the protocol level make it much easier to build a simple decoder without the need for a sophisticated scanner, because the decoder can always decide a value's type from the first character of the encoded text (making the values easy for humans to type is not a concern here):

```
public Object decodeObject () throws CodingException
{
    switch (input.charAt(pos++)) {
        case '*':
            return null;
        case 'b':
            return decodeBoolean();
        case 'i':
            return decodeInteger();
        case 'd':
            return decodeDouble();
        case '\'':
            return decodeComponent();
        case '"':
            return decodeString();
```

```
            case '{':
                return decodeArray();
            default:
                throw new CodingException("unrecognized type: " + input);
        }
    }
```

Again, the actual task of decoding a component reference is left here to a *decoding delegate*. Doing so ensures that the knowledge of how to encode and decode components is encapsulated inside these delegate objects, and allows the `Encoder` and `Decoder` classes to be reusable.

Coding Delegates

The two delegate interfaces used for this each just declare one method:

```
public interface EncodingDelegate
{
    public String encodeObject (Object object) throws CodingException;
}

public interface DecodingDelegate
{
    public Object decodeObject (String string) throws CodingException;
}
```

When used by the remote dispatcher on the client side, the implementation of these interfaces relies on the "object identifier" to component mapping maintained internally by the `Component` class (as explained in section 5.3.6 on page 67):

```
public class CodingDelegate implements DecodingDelegate, EncodingDelegate
{
    public Object decodeObject (String string) throws CodingException
    {
        Object result = Component.getComponent(string);

        if (result == null)
            throw new CodingException("cannot decode unregistered object");

        return result;
    }

    public String encodeObject (Object object)
    {
        String result = object.toString();

        return result;
    }
}
```

On the toolkit server side, the lookup methods based on the component registration mechanism provided by the `Server` base class are used instead (these are described in section 5.5.2 on page 89). This ensures that the correct toolkit components are substituted for the component proxy identifiers when decoding a request and vice versa when encoding a response or an event message:

```
public class CodingDelegate implements DecodingDelegate, EncodingDelegate
{
    public Object decodeObject (String string) throws CodingException
    {
        Object result = Server.getServer().lookupObject(string);

        if (result == null)
            throw new CodingException("cannot decode unregistered object");

        return result;
    }

    public String encodeObject (Object object) throws CodingException
    {
        Object result = Server.getServer().lookupKey(object);

        if (result == null)
            throw new CodingException("cannot encode unregistered object");

        return (String) result;
    }
}
```

Request Messages

In order to allow request messages to be encoded and decoded for transport across the network, it is necessary to add one additional method for encoding and a second constructor for decoding to the `Request` class discussed is section 5.3.5:

```
    public String toString (EncodingDelegate delegate)
    {
        Encoder encoder = new Encoder(delegate);

        encoder.encodeObject(new Integer(number));
        encoder.encodeCharacter(',');
        encoder.encodeObject(target.toString());
        encoder.encodeCharacter(',');
        encoder.encodeObject(message);
        encoder.encodeCharacter(',');
        encoder.encodeObject(args);

        return encoder.toString();
    }
```

The `toString()` method above is pretty self-explanatory, and the additional constructor for decoding a message from a textual representation is simple as well:

```
public Request (String input, DecodingDelegate delegate)
{
    Decoder decoder = new Decoder(input, delegate);

    number = ((Integer) decoder.decodeObject()).intValue();

    decoder.skipCharacter(',');
    target = (Object) decoder.decodeObject();

    decoder.skipCharacter(',');
    message = (String) decoder.decodeObject();

    decoder.skipCharacter(',');
    args = (Object[]) decoder.decodeObject();
}
```

Response and Event Messages

The response and event messages can be encoded and decoded in the same manner as requests, using methods almost identical to the ones shown above. All the hard work is done by the `Encoder` and `Decoder` classes and the corresponding coding delegates of the client and the server.

5.6.4 GTK TCP Toolkit Server

The toolkit server can also be adapted to support such a distributed communication model with a few small changes. For the implementation using native components, the only part of the code that needs to be replaced is the module that uses the *Java Native Interface* to interact with the dispatcher and the event thread. This module is replaced by an equivalent one that implements a TCP based transport and contains a native encoder and decoder for the message protocol described in section 5.6.2. All the rest of the native server implementation can be shared between both versions without any modifications.

The new module now needs to contain a `main()` function, as the TCP server must be able to run as a standalone process instead of being loaded as a shared library into the Java virtual machine. This `main()` function is a lot similar to the `run()` method of the JNI based server. Like before, it needs to initialize the GLib and GTK libraries and the function and object mapping tables. The one major difference here is that the GNet library is used to install a server function for handling an incoming socket connection on the server's TCP port:

```
int main (int argc, char *argv[])
{
    int port = 5000;            // default port number
    GServer *server;

    g_type_init();
    gnet_init();
    gtk_init(&argc, &argv);
    ...
    server = gnet_server_new(NULL, port, server_func, NULL);

    gtk_main();
    return 0;
}
```

The GNet library is a general purpose networking library providing an object-oriented wrapper around the BSD socket API. It supports both synchronous and asynchronous connection handling and represents connections as I/O channels that can be added to the GLib main loop as additional input sources. This is very convenient, as it allows the toolkit server process to be completely single-threaded without any further efforts: Both the connection to the client as well as the event handling for the toolkit can be managed by the toolkit's main loop, unlike the in-process model where reading requests from a channel needed to be integrated with the GTK main loop.

When the client has established a connection to the server, the data socket is added to the main loop as a new *I/O watch*, so that any incoming requests can be processed from the native main loop:

```
static void server_func (GServer *server, GConn *conn, gpointer data)
{
    io_channel = conn->iochannel;
    g_io_add_watch(io_channel, G_IO_IN|G_IO_HUP, io_watch, NULL);
    gnet_server_delete(server);
}
```

As soon as new data becomes available on the socket, the `io_watch()` function is called by the main loop to read the request. For the text based protocol, each request spans exactly one line of input, so this process becomes fairly simple. The server decodes the request into a `GValue` array and passes it to the `handle_request()` function:

```
static gboolean io_watch (GIOChannel *source, GIOCondition cond, gpointer data)
{
    char *buf = NULL;
    gsize len;
    GIOError status = gnet_io_channel_readline_strdup(source, &buf, &len);

    if (len > 0) {
```

```
        GValueArray *array = decode_request(buf);

        handle_request(array);
        g_value_array_free(array);
    } else {
        gtk_exit(0);
    }

    g_free(buf);
    return TRUE;
}
```

The actual request handling is then done by that function in exactly the same way as for the in-process case already described in section 5.4.2 on page 73. The toolkit interface function corresponding to the request is looked up and called and the result value is sent back to the client:

```
static void handle_request (GValueArray *request)
{
    ...
    if (object) {
        GValueArray *args =
            g_value_get_boxed(g_value_array_get_nth(request, 3));
        GValue result = {0};
        call_func_t call_func = lookup_call(method);

        if (call_func) {
            call_func(object, args, &result);

            if (G_IS_VALUE(&result)) {
                post_reply(seqnum, &result);
            }
        }
    } else {
        init_func_t init_func = lookup_init(method);

        if (init_func) {
            object = init_func();
            register_object(object, target);
        }
    }
}
```

Of course the sending of the result value now involves generating a reply message that can be transmitted across the TCP connection, so the result value must be properly encoded into a text message first:

```
static void post_reply (long seqnum, GValue *result)
{
```

```
    ...
    encode_long(buf, seqnum);
    g_string_append_c(buf, ',');
    encode_value(buf, result);
    g_string_append_c(buf, '\n');

    gnet_io_channel_writen(io_channel, buf->str, buf->len, &len);
    status = g_io_channel_flush(io_channel, NULL);
    ...
    g_string_free(buf, TRUE);
}
```

Apart from the request handling shown above, the other major part of the toolkit server is the forwarding of events generated by the native components to the client. The same event listener module is used here that was done for the in-process toolkit server, so all event messages are routed through the function `post_event()` in the server. This function is responsible for properly encoding the event source and parameters according to the protocol and sending the constructed event message via the socket connection to the event dispatcher in the client:

```
void post_event (gpointer source, const char *event, GValueArray *args)
{
    ...
    encode_component(buf, source);
    g_string_append_c(buf, ',');
    encode_string(buf, event);
    g_string_append_c(buf, ',');

    if (args != NULL) {
        encode_value_array(buf, args);
    } else {
        encode_pointer(buf, args);
    }
    g_string_append_c(buf, '\n');

    gnet_io_channel_writen(io_channel, buf->str, buf->len, &len);
    status = g_io_channel_flush(io_channel, NULL);
    ...
    g_string_free(buf, TRUE);
}
```

The text protocol encoding and decoding is actually very similar to the implementation shown before in Java (encapsulated in the `Encoder` and `Decoder` classes): `encode_value()` tries to determine the type of the value to encode and then invokes one of the corresponding type-specific encoding functions:

```
static void encode_value (GString *buf, GValue *value)
{
```

```
    if (G_VALUE_HOLDS_POINTER(value))
        encode_pointer(buf, g_value_get_pointer(value));
    else if (G_VALUE_HOLDS(value, G_TYPE_VALUE_ARRAY))
        encode_value_array(buf, g_value_get_boxed(value));
    else if (G_VALUE_HOLDS_STRING(value))
        encode_string(buf, g_value_get_string(value));
    else if (G_VALUE_HOLDS_BOOLEAN(value))
        encode_boolean(buf, g_value_get_boolean(value));
    else if (G_VALUE_HOLDS_LONG(value))
        encode_long(buf, g_value_get_long(value));
    else if (G_VALUE_HOLDS_DOUBLE(value))
        encode_double(buf, g_value_get_double(value));
    else if (G_VALUE_HOLDS_OBJECT(value))
        encode_component(buf, g_value_get_object(value));
    else
        fputs("invalid type in encode_value()\n", stderr);
}
```

`encode_value()` works exactly the other way round and is again based on type-specific decoding functions, as expected. All the rest of the server code is shared with the in-process implementation.

5.6.5 Swing TCP Toolkit Server

It should now be fairly obvious that the same procedure seen before can also be applied to the Java toolkit server: Replacing the `JavaServer` implementation from section 5.5.2 with a different subclass of the abstract `Server` class that supports remote communication with the client. The overall structure of this class is in fact very similar to the native TCP toolkit server described above. It also needs to be able to run in its own process now, and it too must be able to read requests from the socket connection to the client while simultaneously handling events from the toolkit. But this already points to one difference: There is no way in Java to integrate the process of waiting for data on a socket into the main loop of the Swing toolkit. As a consequence, the request handling will have to be done in a separate thread. Again, apart from this class, all the rest of the code on the server side – and in particular the component and event listener classes – can be simply reused completely unchanged with this `TCPServer`.

The `main()` method has to set up the toolkit server to listen on its server port for connections from a client. Once a connection has been established, the request handling loop can be started:

```
public class TCPServer extends Server implements Runnable
{
    public static void main (String args[])
    {
```

```
    try {
        String display_name = System.getProperty("gui.display");
        Display display = new Display(display_name);
        TCPServer server = new TCPServer();

        server.bind(display.getPort());
        server.run();
    } catch (IOException ex) {
        ...
    }
}
```

The server's `bind` method is used to accept an incoming client connection and sets up the input and output streams used for the communication with the client:

```
private void bind (int port) throws IOException
{
    ServerSocket socket = new ServerSocket(port);

    init(socket.accept());
    socket.close();
}
```

The thread that is responsible for processing the client request messages spends its time in the `run()` method, waiting for data from the socket and decoding any request messages that arrive there. The actual request handling is again pushed into the `handleRequest()` method:

```
public void run ()
{
    try {
        String line;

        while ((line = client_in.readLine()) != null) {
            Request request = new Request(line, delegate);

            handleRequest(request);
        }
    } catch (IOException ex) {
        ...
    }
}
```

The job of `handleRequest()` is then the same procedure as always: Look up the toolkit method corresponding to the request and invoke the operation with the decoded argument list provided in the request message:

```
public void handleRequest (Request request)
{
    ...
```

```
    if (object != null) {
        Object args[] = request.getArguments();
        Method m_call = methods.lookupCall(method);

        if (m_call != null) {
            invokeRequest(request, m_call, new Object[] {object, args});
        }
    } else {
        Method m_init = methods.lookupInit(method);

        if (m_init != null) {
            try {
                object = m_init.invoke(null, new Object[] {});
                registerObject(object, target);
            } catch (InvocationTargetException ex) {
                ...
            }
        }
    }
}
```

As explained above, the requests cannot be received directly by the event thread because reading requests would block the event thread, which must *not* be done. As a consequence, the thread that is used to handle the client requests is not permitted to talk to the user interface components, due to the known limitations of the Swing toolkit. This can be solved, however, by pushing the code that actually calls the toolkit methods into the event thread using the same `invokeLater()` trick seen before. Method results are also sent back from this thread in the form of encoded reply messages:

```
private void invokeRequest (final Request request, final Method method,
                            final Object args[])
{
    EventQueue.invokeLater(new Runnable() {
        public void run () {
            try {
                Object result = method.invoke(null, args);

                if (result != JavaGUI.VOID) {
                    int seqnum = request.getNumber();
                    Reply reply = new Reply(seqnum, result);

                    client_out.write(reply.toString(delegate));
                    client_out.newLine();
                    client_out.flush();
                }
            } catch (IOException ex) {
                ...
            }
```

```
        }
    });
}
```

The complete event handling process in the `TCPServer` works in the same way as for the `JavaServer`: The Swing event listener is used as a generic event handler that forwards all events to the `postEvent()` method in the toolkit server. The implementation provided by this class then just has to send the encoded representation of each message across the socket connection to the client:

```
public void postEvent (EventMessage message)
{
    try {
        client_out.write(message.toString(delegate));
        client_out.newLine();
        client_out.flush();
    } catch (IOException ex) {
        ...
    }
}
```

The role of the `loadApplet()` method in the server has already been explained at the very end of section 5.5.2 on page 93: Its responsibility is to arrange the loading of an applet that is used as a toolkit client. Contrary to the implementation seen in the `JavaServer` class, the applet cannot be directly loaded here, because for a distributed communication model the applet of course must be loaded on the client, not on the server. This means that for an applet using the `TCPServer`, the applet class is not loaded into the browser (that contains the toolkit server for the applet) but into the Java process on the remote system where the toolkit client is running. This is why in this case the applet's class name is simply transmitted to the client instead:

```
public void loadApplet (JApplet applet, String name)
{
    try {
        ...
        Thread thread = new Thread(this);

        init(new Socket(host, display.getPort()));
        client_out.write(name);
        client_out.newLine();
        client_out.flush();

        synchronized (applet) {
            thread.start();

            while (lookupKey(applet) == null) {
                applet.wait();
```

```
                }
            }
        } catch (IOException ex) {
            ...
        }
    }
```

Much more details on the handling of applets in general and an example applet are shown in section 5.8 on page 118.

5.7 Example Application

To show how the architecture presented here looks from the view of an application programmer, this section discusses a simple example program to illustrate that point. The program simply creates a window containing a grid layout with three buttons (that this is done in the `run()` method can be ignored for now):

```
public class ThreadDemo extends Thread
{
    public void run () {
        Window window = new Window("ThreadDemo");
        Button create = new Button("Create New Window");
        Button doquit = new Button("Quit Application");
        Button counter = new Button("Start Countdown");
        Grid grid = new Grid(3, 1);
```

The component classes `Window`, `Button` and `Grid` used here should not need any explaining, they work just like the corresponding AWT classes. `Grid` is a layout container with a configurable grid layout, somewhere in between the `GridLayout` and `GridBagLayout` found in the AWT. Unlike the layout classes, it is however a container with a fixed layout, in the same way that the `Box` class is used in Swing.

Putting the component hierarchy together by adding the individual components to the layout containers is also really straightforward:

```
        grid.add(create, 0, 0);
        grid.add(doquit, 1, 0);
        grid.add(counter, 2, 0);
        window.add(grid);
        window.setVisible(true);
```

The `add()` method of the `Grid` class accepts optional coordinates to specify the cell where the component should be placed. Much more interesting is to look at how the event handling is done. The event handlers are implemented in separate *event listener* classes much like `ActionListener` implementations are used for AWT or

Swing event handling, and these are attached to the components using the generic `addEventListener()` method:

```
create.addEventHandler("clicked", new StartHandler());
doquit.addEventHandler("clicked", new QuitHandler());
counter.addEventHandler("clicked", new LoopHandler());
window.addEventHandler("closing", new CloseHandler());
```

Because this method is used for all types of events, it is necessary to specify the type of event the handler is interested in separately. For this, a set of standard *event names* is defined (strings are a lot more flexible than the event IDs used in the AWT), for example the "clicked" event type corresponds to a click on a button object etc.

Finally, the application needs to start the main event loop. This is made explicit in the API because there simply is not "the one" event loop: There can be as many as the application needs, and to be able to control them they have to be visible to the program. To make the common case easy, the `MainLoop` class provides a class method to obtain the *default main loop* for a thread, automatically creating one if necessary. Each thread can have its own default main loop, of course. So, the code to start the main loop is very simple:

```
        MainLoop.defaultMainLoop().run();
    }
```

To be able to actually start the program, it additionally needs to have a `main()` method:

```
    public static void main (String args[]) {
        new ThreadDemo().start();
    }
```

The `main()` method creates a new `ThreadDemo` object (which extends the `Thread` class) and starts it. The new thread will begin its execution in the `run()` method seen above, opening a new window and starting a main loop in this thread. The original main thread simply terminates by leaving the `main()` method. As a result, there is one thread, one window, and one main loop for handling events.

The next aspect to look at is what the event handlers are doing: The simplest one is the handler for the *Quit Application* button that simply terminates the program using `System.exit()`:

```
    public class QuitHandler implements EventListener {
        public void handleEvent (Event event) {
            System.exit(0);
        }
    }
```

As can be seen here, the `EventListener` interface is in fact very similar to the `ActionListener` mentioned before: It declares a single method, `handleEvent()`, that needs to be implemented with the handler code. The only parameter is the event object of type `Event`, which is the base class of all events, so the actual type of the event may also be a subclass of this.

The event handler for the "Create New Window" button creates a new `ThreadDemo` object when clicked, and starts the new thread, just like `main()` did:

```
public class StartHandler implements EventListener {
    public void handleEvent (Event event) {
        new ThreadDemo().start();
    }
}
```

What does this do? It creates a second thread that also executes its `run()` method, which creates a new window and starts a new main loop in the second thread. This is because (unless arranged otherwise) each thread gets its own *default main loop*. So after one click on this button there are two threads, two windows and two main loops. It should now be obvious how this continues: Each press on this button in any of the windows will create an additional thread, window and main loop.

This leads to the next question: How do the components inside the individual windows know which main loop is responsible for handling the events they generate? The answer to this question is simple: Since no particular main loop was specified that should be used for delivering the events when connecting the event handlers, they were connected to the *default main loop* of the thread that did the `addEventListener()` call. This also explains why the code was placed in the `run()` method of the new thread: If the same thread that connects the event handlers also wants to handle them, nothing special needs to be done. Of course there are also variants of the `addEventListener()` method that let the caller determine which main loop should be used for event delivery.

The bottom line is that one may have any number of windows, each with its own main loop responsible for handling the event generated inside that window. But how does one get rid of the threads again? This is equally simple as shown in the handler for the window close button:

```
public class CloseHandler implements EventListener {
    public void handleEvent (Event event) {
        MainLoop.defaultMainLoop().terminate();
    }
}
```

When a window is closed, the handler for the "closing" event is called, which in turn

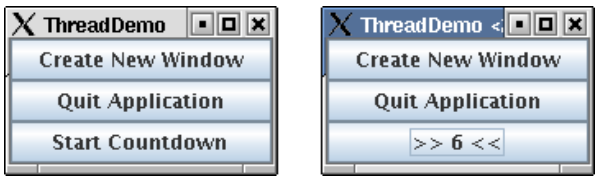

Figure 5.3: Screen shot of the ThreadDemo application

calls the `terminate()` method of the main loop for the thread that is responsible for this window. Terminating an event loop causes the `run()` method of the main loop to return, which here was called as the last statement in the `run()` method of the thread itself, so that this also causes the thread's execution to end.

To illustrate that the individual main loops can really run independently of each other, the handler for the last button is doing something seemingly stupid: It blocks the event handling thread for ten seconds and displays a countdown during that time on the button:

```
public class LoopHandler implements EventListener {
    public void handleEvent (Event event) {
        Button counter = (Button) event.getSource();
        String text = counter.getText();

        for (int i = 10; i > 0; --i) {
            counter.setText(">> " + i + " <<");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {}
        }
        counter.setText(text);
    }
}
```

Figure 5.3 shows a screen shot of the running application on the Linux platform using the Swing toolkit server. The event thread of the left window is idle while the thread of the second window is executing the `LoopHandler` code. Despite the fact that one event handler is currently busy, both windows repaint fine and the left window can process events normally. The countdown can run in both (or even more) windows in parallel, without disturbing the other windows.

When trying to do the same thing in a normal Swing application, one will notice that while it is possible to open multiple windows, starting the countdown in one of them will cause *all* windows to become unresponsive until the countdown is over. They will not repaint either if they are temporarily overlapped with another window, meaning that the countdown will not even be visible in the window where it was started. This does not even work if there is just one open window, because the repaints queued by the `setText()` operations on the button cannot be performed

while the event handler is still running.

More examples as well as the complete source code for the prototype implementation can be found on the project web site [16].

5.8 Java Applets

Due to the fact that a Swing toolkit server exists, it is also possible to run this server inside an applet, using either the in-process model or a TCP transport for the message exchange.

The main limitation of an applet is that it may not interact freely with the system on which is it running due to security considerations, for example it cannot work with local files or open arbitrary network connections. There are two exceptions, however: It is allowed to communicate with the web browser in which it is running and it may open a network connection to the server from which this applet itself was loaded.

Applets in general work a little different from Java applications: They do not have a `main()` method and often do not create their own windows (though an applet is allowed to do this). Instead, an applet is allotted a fixed part of the browser's window to render itself into and can thus become an integral part of a web page. Once the space has been reserved by the browser, the applet class is loaded and instantiated. Then, the applet's `init()` method is called to signal that the applet's life cycle has begun.

To implement this model on top of the message-based approach presented here, two components are necessary: An implementation of a Swing `JApplet` subclass that can be loaded into a browser and runs the toolkit server, and an `Applet`[6] class similar to the standard one which can then be subclassed and used as a template for the actual applet specific code.

5.8.1 Swing Applet Proxy

The Java applet running the toolkit server can actually be reused for *any* applet, independent of what the applet is doing, since it just needs to setup the stage for the toolkit server to run. In the implementation provided in the prototype, this class is called `JavaApplet` and extends the `JApplet` class as expected. It can be parametrized using the applet parameters *gui.applet* and *gui.display*: *gui.applet* selects the name of an applet class to be loaded as a *client* for the toolkit server

[6] This does *not* refer to the AWT applet class but rather to the very similar class provided by this prototype.

running in this applet, which must be a subclass of the `Applet` class discussed below. The property *gui.display* determines which kind of message transport is used between the client applet and the toolkit server (TCP or in-process):

```
public class JavaApplet extends JApplet
{
    public void init()
    {
        String name = getParameter("gui.applet");
        String display = getParameter("gui.display");

        if (display != null)
            new TCPServer();

        if (name != null)
            Server.getServer().loadApplet(this, name);

        Server.getServer().postEvent(new EventMessage(this, "applet-init"));
    }
```

The operations `init()`, `start()`, `stop()` and `destroy()` that control the applet's life cycle are simply encapsulated as pseudo event messages and sent to the applet client:

```
    public void start()
    {
        Server.getServer().postEvent(new EventMessage(this, "applet-start"));
    }

    public void stop()
    {
        Server.getServer().postEvent(new EventMessage(this, "applet-stop"));
    }

    public void destroy()
    {
        Server.getServer().postEvent(new EventMessage(this, "applet-destroy"));
    }
}
```

5.8.2 Applet Template Class

The `Applet` class that acts as the template for the client applets now just needs to attach event handlers to each of these events to make the forwarding of these operations to the client completely transparent:

```
public class Applet extends Container implements Runnable
{
```

```
MainLoop loop = new MainLoop();

protected Applet (Class type)
{
    super(type);

    try {
        addEventHandler("applet-init", loop, this, "appletInit");
        addEventHandler("applet-start", loop, this, "appletStart");
        addEventHandler("applet-stop", loop, this, "appletStop");
        addEventHandler("applet-destroy", loop, this, "appletDestroy");
    } catch (NoSuchMethodException ex) {
        System.err.println(ex);
        System.exit(1);
    }
}
```

Actually, the applet client needs to have its own main loop, just like an application, so this is an example of how to connect event handlers to a specific main loop. Once that is done, the applet's main loop can be started with the `run()` method:

```
public void run ()
{
    loop.setDefaultMainLoop();
    loop.run();
}
```

One problem remains, however: Because an applet is not allowed to open a listening socket (which is called a `ServerSocket` in Java) for security reasons, the connection process has to be reversed for an applet when using the TCP message transport: The applet client needs to open a `ServerSocket` to which the toolkit server can then connect from inside the web browser. This works because an applet is allowed to open an outgoing socket connection to the same server the applet itself was loaded from.

5.8.3 Generic Applet Server

To avoid having to include the connection handling code (i.e. opening the socket and instantiating the dispatcher) into each applet client, there is a simple *applet server* class that can act as a generic applet client loader. It establishes the server socket, obtains the applet class name from the applet proxy (via the toolkit server) and tries to instantiate the given applet class using the Java reflection API. If this succeeds, it will proceed to start the applet client's main loop:

```
public class AppletServer
{
```

```
    public static void main (String args[])
    {
        try {
            ...
            ServerSocket socket = new ServerSocket(display.getPort());
            Socket server = socket.accept();

            TCPDispatcher dispatcher = new TCPDispatcher(server);
            String applet_name = dispatcher.getAppletName();
            Class applet_class = Class.forName(applet_name);
            Applet applet = (Applet) applet_class.newInstance();

            socket.close();
            applet.run();
        } catch (ClassNotFoundException ex) {
            ...
        }
    }
}
```

5.8.4 An Example Applet

Just to give an idea of how this is used in practice, the class for a very simple example applet is included here. The applet itself only contains a single button that reacts to *clicked* events, and for each of the `init()`, `start()`, `stop()` and `destroy()` methods it will print the method name to the Java console:

```
public class DemoApplet extends Applet
{
    public void init()
    {
        System.err.println("[init]");
        Button button = new Button("Applet");

        add(button);
        button.addEventHandler("clicked", new EventListener() {
            public void handleEvent (Event event) {
                System.err.println(event.getSource());
            }
        });
    }

    public void start()
    {
        System.err.println("[start]");
    }

    public void stop()
    {
```

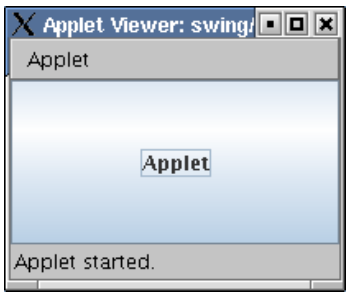

Figure 5.4: Demo applet in the AppletViewer

```
        System.err.println("[stop]");
    }

    public void destroy()
    {
        System.err.println("[destroy]");
    }
}
```

The HTML code used to embed this applet into a web page illustrates that it has to reference the `JavaApplet` proxy class for the applet *code* attribute (instead of the example applet class), and that it includes the `DemoApplet` class as the value of the *gui.applet* applet parameter. Also, the *gui.display* parameter indicates the port number on which the applet server is running on the system that is serving the web page:

```
<!DOCTYPE html public "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Demo Applet</title>
  </head>
  <body>
    <center>
      <applet code="swing/JavaApplet.class" width="200" height="100">
        <param name="gui.applet" value="demo.DemoApplet">
        <param name="gui.display" value=":5000">
      </applet>
    </center>
  </body>
</html>
```

Figure 5.4 shows this applet running inside the JDK's applet viewer for example. Though this is not immediately obvious when using the applet, clicking on the button actually executes the handler code on the server side, not in the proxy applet that is running inside the applet viewer, so the method names printed by the applet will appear on the server.

5.9 Summary

This chapter presented the prototype implementation for the message-based approach proposed in chapter 4. Though the current prototype only contains a fairly small selection of component classes and only supports a subset of the possible event types, it nonetheless proves that the general approach presented here is in fact viable and can be offered in the form of a rather nice application programming interface to the user of the toolkit.

As already discussed in section 4.10 on page 55, the message-based approach can fulfil all of the objectives presented in the design:

- It offers transparent thread-safe access to user interface components.
- It supports parallel even handling in a way that is very simple to use.
- It can present a clean, abstract API to the application with easily replaceable backend implementations.
- It allows applications to display their interface on a remote system.

The implementation of the four different toolkit servers has also shown that it is possible to isolate the changes required for making a toolkit server capable of transparent access across a network from the generic code that interacts with the native toolkit components. Only a fairly small module has to be redone in order to use a given toolkit server with a completely different communication mechanism.

Chapter 6

Evaluation of the Results

The fact that the approach presented in the previous chapters works for small programs does not automatically imply, however, that it may not have practical problems that may prevent is from being used for real-world applications.

The main point where this could be a problem is the question of the performance implications incurred by the proposed architecture: It adds a lot of indirection to the originally very simple process of calling a method on a component. On the other hand, communicating with graphical components already is a fairly involved process, because the component rendering and the physical event handling are often managed outside of the program anyway: for example by the X11 display server on Unix platforms of the Quartz WindowServer on Mac OS X. All of this does it not make very easy to evaluate the performance implications without doing actual measurements, which are presented in the following section.

But this chapter discusses other potential problems as well: How the thread interaction is designed in way that helps to avoid potential deadlocks and which limitations the overall architecture places on the flexibility available to the application developer, especially regarding the design of custom user interface components.

6.1 Performance Evaluation

All of the extra flexibility provided by this approach comes at a price: performance. So one has to ask the question: Is using an (albeit somewhat simplified) distributed object model feasible for building a graphical program's user interface considering its potential performance implications?

Based on the design presented in chapter 4, there are two main areas that are likely to introduce performance degradation: locking when accessing the request

and event queues and thread communication overhead: encoding and decoding the actual request and event objects, as well as mapping object identifiers to the component object references or the corresponding proxy objects, respectively.

To evaluate this, several factors need to be considered:

- Which percentage of the time used by the program is spent interacting with the GUI components?

  This is an aspect that is fairly difficult to evaluate objectively. One would expect that the performance of an application is determined mainly by the application logic, not the component interactions or the event dispatching time, though this varies of course depending on the concrete application one is looking at. This factor is very difficult to measure in practice as well, since this would include simulating use cases for a program, which is well beyond the scope of this thesis. So, the worst case assumption is made here that the application performance is in fact limited by the toolkit performance.

- What is the overhead involved in using the distributed object system on the level of an individual method call?

  Here, two separate cases need to be looked at: Operations that produce a result value from the toolkit require a full roundtrip to the component implementation to be able to deliver the result value before the application can proceed. This class of operations includes for example all methods that request component properties.

  Operations that do not have a result value can be executed completely asynchronously, that is the application can already proceed while the method is still queued for invocation by the toolkit. This class of operations includes all methods that only update component properties and cannot report failure (as explained earlier, exceptions on the component side need to be handled as result values and can be reconstructed at the proxy level). All of this is invisible to the application, since the ordering of the requests guarantees that a synchronous operation always waits for all previous requests to complete. This is necessary because an application can rely on the fact that methods on the components are not executed out of order. Asynchronous operations only involve half the time that a full roundtrip would take.

- What is the delay incurred for handling user events? Does the program's responsiveness suffer in a noticeable way?

  To determine this, it is necessary to measure the amount of time it takes to dispatch an event from the toolkit server to the application. Since events and request replies are actually very similar (the only real difference being the channel they are dispatched to in the application code), this can be

calculated from the roundtrip time for a request: Assuming that the time spent on constructing and transporting a request is about equal to that spent on a reply, the time taken to dispatch an event will be half the roundtrip time.

The second of these questions will be looked at here first: One way to evaluate this is to measure the time spent for a single roundtrip from the user application to the toolkit library. The actual timing was done on the `getText()` method of a button object, as this is a very basic operation: Is it just reading a GUI component's attribute, no update of the user interface is required. The results of these measurements are shown in the first column of figure 6.1 on the next page.

A second measurement has been performed on a typical method that needs to update a component's display and can be executed asynchronously. Note that for all toolkits involved here synchronous GUI updates have been forced, since updating the user interface is always handled by a separate thread (or even a separate process in a windowing system like X11). So the time given here includes the time spent by the toolkit drawing the updated component. For the message-based approach it also includes half the roundtrip time from the first column. The actual timing here was done on the `setText()` method of a button object, again a very basic operation that requires a clearly defined GUI update: drawing a new button label.

For comparison purposes, the third column gives the time spent for the same `setText()` operation *without* forcing synchronous GUI updates. This number is always lower, of course, since this test just updates the internal component state and queues a redraw notification on the component. Note however, that this is the default mode of operation.

Each timing is given in microseconds per operation (averaged over 1000 calls in 5 runs for each test), measured using Vlad Roubtsov's high resolution native timer implementation for Java [23]. The test system was an Intel Pentium 4 based PC system (2.0 GHz, 512 MB RAM) running Linux 2.6 and XFree86 4.3. The Java runtime environment used for all tests is Sun's JDK release 1.5.0_06 for Linux with the HotSpot JIT compiler enabled.

There are some interesting observations that can be made from these results:

- The numbers indicate that the Swing toolkit is actually a lot faster than the AWT for the measured operation. Looking at the test window while the test is running points to a likely reason for this: Both the AWT and GTK toolkits can be seen updating the text on the screen while the test is running, while the Swing version just displays the final text after the test run is complete. Swing is obviously a lot smarter about dropping unnecessary repaints that have become obsolete due to further repaints on the same component. This

GUI toolkit used	roundtrip	sync update	async update
standard Java AWT toolkit	n/a	450 μs	224 μs
standard Java Swing toolkit	n/a	166 μs	85 μs
message passing, JNI, GTK	134 μs	542 μs	206 μs
message passing, TCP loopback, GTK	417 μs	694 μs	246 μs
message passing, Channels, Swing	130 μs	510 μs	24 μs
message passing, TCP loopback, Swing	674 μs	711 μs	119 μs

Figure 6.1: Performance evaluation of single method calls

might be a good optimization to apply to the other toolkits as well.

- Considering that the measured `setText()` operation does not have a return value and so only requires half the roundtrip time in communication overhead, one can conclude that the actual time spent in the method for updating the GUI is around 480 μs for the GTK server and 400 μs for the Swing server. Given that one would expect the native toolkit to perform better, there is obviously still some room for improvement in either the GTK server or the GTK toolkit itself.
- The extra cost incurred by the message passing is noticeable, but in the same order of magnitude as the cost for the GUI update itself. Compared with the AWT toolkit, using the message-based approach with an in-process toolkit server is not significantly slower, so such an approach is clearly usable in general. The message transport across the TCP loopback interface is a bit more costly, but that is to be expected given the extra flexibility (and required context switches between two processes).
- Given a roundtrip time of about 130 μs for the in-process case, one can expect half that time for dispatching a single event from the toolkit server to the application, which translates to a maximum limit of above 10000 events per second on typical consumer hardware, ignoring the time spent in the event handler itself. So, for all practical purposes, the limiting factor will be the actual event handler code in the application, not the event handling mechanism itself.

6.2 Locking Considerations

Whenever a complex system involving multiple threads is designed, one should take a careful look at the possibility of *deadlocks* created by the thread interaction. A *deadlock* is formally any situation within the program where two or more threads cannot continue because each is waiting for one of the other threads to complete

some operation.

The general requirements for a deadlock to occur are:

1. exclusive (i.e. non-shared) use of resources
2. processes wait while trying to allocate a resource
3. allocated resources cannot be reclaimed forcefully
4. there is a cyclical wait condition consisting of a ring of processes each waiting for the next one.

The easiest way to avoid deadlocks here is to avoid the last one of these requirements: the cyclical wait condition. To understand this, one has to take a closer look at the message flow between the involved threads:

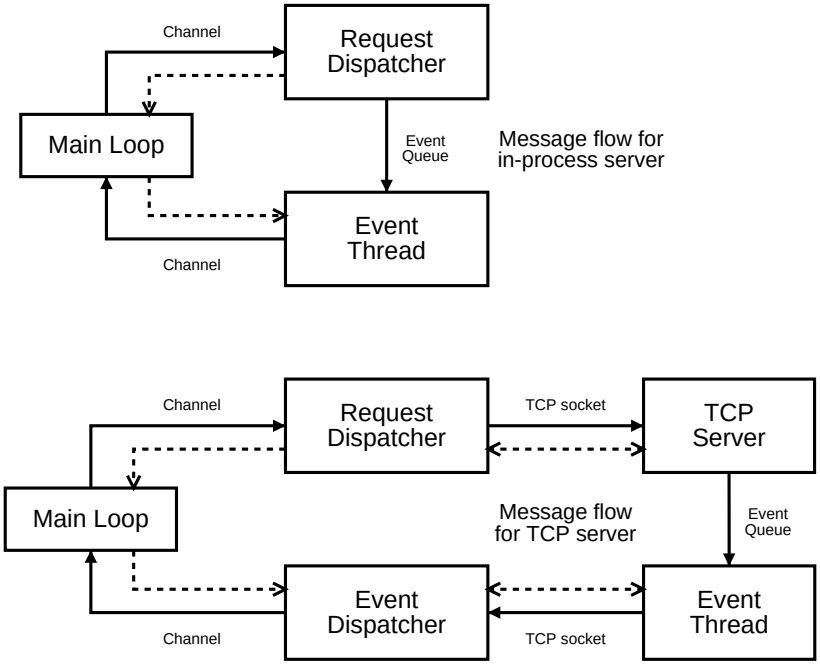

Figure 6.2: Analysis of the message flow

The solid lines indicate the direction of the message flow[1] and the dotted lines show which thread might potentially wait for another thread. As can be seen from figure 6.2, there is indeed a cycle in the message flow, but this does not translate into a cyclical wait condition. To understand why, it is necessary to first look at the blocking behaviour of the different communication mechanisms used in this model:

[1] The two streams of the bidirectional TCP socket have only been split here for clarity, they together form one socket connection.

- Channels are used in fully buffered mode, i.e. the only operation that can block is a `receive()` operation on the channel. That directly translates into the fact that only the reader on a channel may have to wait on the writer.
- Sockets are different: Here, both the reader as well as the writer may be blocked when the network buffer is either empty upon read or full upon write. But since the buffer cannot be empty and full at the same time, only one of these wait conditions can occur at the same time.
- The event queue is completely non-blocking: Posting requests for execution in the event thread will never block the sender, as the event queue inside the application is not limited in length. The event thread will never actively wait for such requests, either, so there is no wait condition to examine here.

Avoiding possible deadlocks is actually one of the main reasons for splitting the event dispatcher from the request dispatcher on the application side: If both were running in the same thread, a deadlock might potentially occur due to a cyclical wait condition. But running both tasks in the same thread would not be feasible anyway because is it simply impossible in Java to simultaneously wait for input on a socket and a channel, which translates to a `wait()` operation on a monitor internally.

6.3 Restrictions

The next section discusses the two main restrictions inherent in the architecture presented in this thesis: Portability to different platforms that do not share the same graphical toolkit library and creating custom components for use in an application.

6.3.1 Portability

Since the general approach is similar to the model used by the *Abstract Windowing Toolkit* or the SWT where the set of component classes available to the application programmer is limited by the common subset of the components provided by the underlying toolkit(s), the same basic limitation applies here as well.

Depending on the richness of the toolkits in question, this can become a serious problem (as has been the case for the AWT), but this limitation can be avoided in general in one of two ways:

- By also using external component implementations, the palette of components can be extended greatly. For example, the GTK toolkit itself does not provide a component for rendering HTML content, and neither does the

Win32 API. There are, however, some embeddable web browser components that fill this need: Both the Internet Explorer as well as the Gecko rendering engine of the Mozilla Firefox browser can be used for this. This is the route that the Eclipse SWT is taking in this regard: Using (mostly) native components combined with selected third party add-ons to the components provided by the platform libraries.

- The other way to solve this problem is to enforce the use the same toolkit across all supported platforms. That was simply not an option back then when the AWT was designed, but today there are already several toolkits that provide the necessary portability across all the major platforms (Windows, Mac OS X and Unix/X11). *Qt*, *GTK* and *wxWidgets* could be possible choices for this.

6.3.2 Creating Custom Components

Assume that an application is not satisfied with the set of components provided by the toolkit library and would like to use for example a turnable knob instead of a slider in its user interface. Though the general architecture allows for creating custom components in principle by sending the repaint events to the application and also supporting paint operations as requests on a component, the performance may suffer badly when a large number of drawing requests is required each time a component is repainted, though this can be alleviated somewhat by buffering of window contents in the window system.

This eventually leads to the question of how many applications really need to implement their own components from scratch, and what alternatives there are to avoid performance problems in such a case:

Since implementing new components at the toolkit level is hard either way even using native code, there is a growing trend among toolkit designers to avoid this in favour of a more general approach: a highly flexible canvas element. A canvas is basically – as the name implies – a blank component that can be painted upon by the application to display anything it wants. It also can trap the physical events (mouse and keyboard operations) in order to allow for interaction of the user with the component. For example Java supports two canvas classes: `Canvas` in the AWT and `JPanel` in the Swing toolkit. Both enable the programmer to design specialized components not provided by the toolkit itself. There is one major disadvantage however: The Java canvas does not provide anything beyond the basic empty space on which the component can draw itself.

A different idea can be seen when looking at the canvas component provided in John Ousterhout's *Tk toolkit* [20]: The Tk canvas (which is turn was inspired by Bartlett's *ezd* structured graphics application for scheme) is much more flexible

due to the fact that is behaves more like a kind of generic container for graphic primitives. Tk even allows putting other components into a canvas.

Newer toolkits tend to follow the Tk approach: Make the canvas more flexible by changing it from a simple drawing area into a generic container for such graphic primitives as lines, boxes, text etc. Additionally, each of these primitives can react to events triggered by the user using the usual event handling mechanism. The canvas thus basically turns into a small rendering system that also automatically handles scrolling, clipping and z-ordering for overlapping primitives[2].

In fact, the original motivation in the GTK toolkit came from the fact that code for custom components (like the drawing area in GIMP or Inkscape, the table view of Gnumeric or the slide view in a presentation program) is often duplicated across many applications, for example the canvas used in Inkscape shares a lot of code with the canvas used in the Dia diagram editor. A proposed implementation for a generic GTK canvas in described in the article [2].

Building the custom control from the example suddenly becomes quite simple now: The application only needs to define the layout of the control once, using the provided graphic primitives or alternatively simply loading an image into an image element of the canvas. When the user drags the knob, the application only has to tell the canvas to rotate its content by the corresponding amount, the actual painting is then handled completely inside the toolkit. This concept makes building custom components much easier and has the added bonus that it would work just fine in combination with this message-based approach, completely avoiding the need for paint events to be exchanged between the application and the toolkit server.

6.4 Summary

As seen in the first section of this chapter, the extra overhead introduced by the message passing is clearly measurable, but not as big as one might have expected and certainly in no way prohibitive, even for a worst case scenario where the overall performance of an application primarily depends on the performance of the graphical toolkit. Whether such a worst case scenario even applies to many real-world applications is doubtful, but even when making this assumption, the approach presented here performs not significantly worse than the original AWT toolkit used for comparison.

The extra overhead introduced by the message passing is proportional to the number of requests and events that are interchanged between the application and the

[2]There is even a canvas implementation for GTK that aims to provide the ability to load a subset of SVG for this in the future.

toolkit backend, so one can expect two use cases where this extra cost may not be acceptable: large numbers of events and large numbers of requests (or possibly a combination of the two). So the question to ask is: How common are those use cases, and what can be done to mitigate them?

A standard user interface component can act mostly independent of the application, for example a list that supports drag and drop can implement all the mouse movement handling itself without notifying the application of anything beyond the final drop event. There is actually little need in this case for the application itself to track the mouse movement, which is a fairly typical example where a large number of events is generated inside the toolkit. Generally, pointer tracking, which is the prime example of generating a rapid sequence of events, is rarely seen outside of component implementations, which can often better be done using a generic canvas element anyway, moving the performance critical parts into the component implementation in the toolkit.

Chapter 7

Further Work

There still remains a lot of room for further work and in particular optimizations that could be applied to the implementation on different levels:

Extend the range of available components

Although enough component classes are included in the prototype for some simple applications, the wide range of components provided by either Swing, GTK or similar toolkits is not covered by far. Doing so is mostly a matter of time spent on writing the component proxies on the client side and implementing the corresponding forwarding functions in the toolkit server.

Expanding the available set of components and their functionality could be made a lot easier by creating some form of automatic tool to (at least) generate the proxy code, since these could easily be auto-generated.

Use a binary protocol instead of the text protocol

The text based protocol currently used for network transmission of requests, responses and events should be replaced by a compact binary protocol (as already mentioned before in section 5.6.2 on page 100) for better overall performance. This could also help to significantly reduce the roundtrip time for synchronous operations in particular.

Also, the range of types supported in the protocol might be extended a bit as well, but since all of the basic types (apart from byte arrays which might be used for images) are already covered there, there is not that much to gain from this.

Optimize the request handling if possible

Avoiding or coalescing multiple pending redraw requests on the same component can provide a significant performance gain in specific cases, as seen for the Swing toolkit in the performance analysis in chapter 6.1 on page 124. However, this is something that really needs to be done at the toolkit level, not in some intermediate code like the prototype.

Offer a simple way to build custom components

While it is possible for the application developer to subclass the user interface component classes defined by the GUI library, doing so cannot be used to build custom components that actually draw their own appearance on the screen, since repaint events are not forwarded in the prototype implementation. This would in fact be fairly easy to do, but that would be useful only if the complete component drawing API is made available to the application as well, which is a lot more work.

While none of this is a technical problem – is would be possible to do this in the current architecture without problems – the performance considerations at this level are very different: Events are only generated sporadically, but paint operations involve lots of method calls in a small time window to generate the component's visual appearance, and lots of small delays here can easily add up to a noticeable visual delay overall.

Adding support for a generic *canvas* component that implements its redrawing directly at the toolkit level as described in the previous chapter is probably the best way to provide application specific components.

Chapter 8

Summary and Conclusion

This thesis proposed a generic message-based approach to event handling in Java modelled around a client/server like architecture and examined the general applicability of such an approach for use in typical application programs and applets.

The comparison of the currently available toolkits for Java presented in chapter 3 has shown that none of them can currently support truly multi-threaded access, and that it may be a worthwhile idea to try to combine and apply some of the ideas found especially in the non-Java toolkits as used for example on the Inferno system and BeOS to a Java toolkit as well.

Chapter 4 presented the general architecture of the approach and described the objectives on which the design has been based. It also discussed why most of these cannot easily or not at all be achieved within the limitations of the conventional Java toolkits. As a possible solution to these problems, the prototype implementation of the message-based architecture was presented in chapter 5 and the applicability of such an approach was discussed in chapter 6.

The prototype implementation has clearly shown that the objectives laid out are actually attainable, in particular regarding transparent thread-safe access to user interface components and the ability to support parallel even handling in a way that is very simple to use, all of which none of the existing toolkits really can provide. Additionally, the architecture allows the separation of the application from its user interface into separate processes to allow the display component to run on a remote server, enabling a new form of location transparency for Java programs as well as a way for applets to be used in a distributed model very similar to web applications.

By simply admitting the reality that the native user interface toolkits have been designed for single-threaded access, it is possible to avoid the problems of locking altogether and simply use a message-based model of component interaction based

on channels instead. Here, the synchronization is implicit in the message handling, and it is much easier to verify than complex locking code distributed across native implementations of Java classes. It is interesting to note in this context that even for the toolkit server based on native code no explicit locking is done. The only place where synchronization actually takes place is at the channels and sockets that connect the different pieces of the architecture.

The answer of the designers of the Swing toolkit to requests for thread support has always been that using multi-threading in a graphical application is somehow "evil" and that the application should be made to live without it whenever possible, and they presented workarounds as in [19]. This thesis has shown that a multi-threaded toolkit can be made to actually work in a deadlock free way, even in a distributed environment. It shows that a multi-threaded toolkit does not have to be the "failed dream" that Graham Hamilton called it [9], and that on the contrary, it can actually work quite well. Yes, there is a performance cost involved in using message passing at the core of the toolkit, but it is comparatively small, and considering how applications can benefit from a redesign of the general toolkit architecture along the lines presented in this thesis, this small price might well be worth paying.

Bibliography

[1] E. Briot, J. Brobecker, and A. Charlet. GtkAda user's guide. 2004. `libre2.adacore.com/GtkAda/docs/gtkada_ug.html`.

[2] Damon Chaplin. GooCanvas - A cairo canvas widget for GTK+. 2005. `www.dachaplin.dsl.pipex.com/goocanvas/`.

[3] Damon Chaplin and Matthias Clasen. GTK+ reference manual. 2005. `developer.gnome.org/doc/API/2.0/gtk/`.

[4] Matthias K. Dalheimer. *Programming with Qt.* O'Reilly, 2002.

[5] Edsger Wybe Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[6] Sean Dorward and Rob Pike et al. The Inferno operating system. *Bell Labs Technical Journal*, 2:5–18, 1997. `www.vitanuova.com/inferno/papers/bltj.html`.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[8] Jesse James Garrett. Ajax: A new approach to web applications. 2005. `www.adaptivepath.com/publications/essays/archives/000385.php`.

[9] Graham Hamilton. Multithreaded toolkits: A failed dream? 2004. `weblogs.java.net/blog/kgh/archive/2004/10/multithreaded_t.html`.

[10] Michael Herczeg. *Software-Ergonomie: Grundlagen der Mensch-Computer-Kommunikation.* Addison Wesley, 1994.

[11] Gerald Hilderink, André Bakkers, and Jan Broenink. A distributed real-time Java system based on CSP. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 400–407, Newport Beach, California, 2000. IEEE Computer Society. `www.ce.utwente.nl/javapp/information/CTJ/DRTJSBCSP.pdf`.

[12] Charles Anthony Richard Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[13] Brian W. Kernighan. *A Descent into Limbo.* Lucent Technologies, Murray Hill, New Jersey, 2000. `www.vitanuova.com/inferno/papers/descent.html`.

[14] Sheng Liang. *The Java Native Interface – Programmer's Guide and Specification.* Addison-Wesley, 1999.

[15] Elmar Ludwig. Kanalarbeit - GTK statt AWT/Swing in Java-programmen. *iX Magazine*, 1, 2001. `www.heise.de/kiosk/archiv/ix/2001/1/127`.

[16] Elmar Ludwig. A multi-threaded gui toolkit for java. 2006. `www.inf.uos.de/elmar/projects/java-gtk/`.

[17] Sun Microsystems. RPC: Remote procedure call protocol specification version 2. *RFC 1057*, 1988. `www.ietf.org/rfc/rfc1057`.

[18] Sun Microsystems. *JavaBeans API specification*, 1997. `java.sun.com/products/javabeans/docs/spec.html`.

[19] Hans Muller and Kathy Walrath. Threads and Swing. *The Swing Connection*, 1998. `java.sun.com/products/jfc/tsc/articles/threads/threads1.html`.

[20] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[21] John K. Ousterhout. Why threads are a bad idea (for most purposes). *Invited talk at the 1996 USENIX Conference*, 1996. `home.pacbell.net/ouster/threads.pdf`.

[22] A.W. Roscoe. The threory and practice of concurrency. *Prentice Hall International Series in Computer Science*, 1997.

[23] Vladimir Roubtsov. My kingdom for a good timer! 2003. `www.javaworld.com/javaqa/2003-01/01-qa-0110-timing.html`.

[24] Raj Srinivasan. External data representation standard. *RFC 1832*, 1995. `www.ietf.org/rfc/rfc1832.txt`.

[25] Dan Parks Sydow. *Programming the Be Operating System.* O'Reilly, 1999.

[26] Kathy Walrath, Mary Campione, Alison Huml, and Sharon Zakhour. *The JFC Swing Tutorial: A Guide to Constructing GUIs.* Addison-Wesley, 2004.

[27] Peter Welch. Process oriented design for Java: Concurrency for all. 2002. `www.cs.ukc.ac.uk/projects/ofa/jcsp/jcsp.pdf`.

[28] Phil Winterbottom. Alef language reference manual. *Plan 9 Programmer's Manual*, 1995. `www.cs.bell-labs.com/who/rsc/thread/alef.pdf`.

Concluding Notes

I would like to thank the following persons for their ongoing support during the time I was working on this thesis:

- Axel Tobias Schreiner for giving me both the opportunity and the incentive to start this work
- Jürgen Schönwälder and Oliver Vornberger for a lot of fruitful discussions over the years and for bearing with me for such a long time
- all my colleagues at the University of Osnabrück
- the members of my family for supporting me during that time

I hereby state in accordance with the regulations applicable to Ph.D. students at the University of Osnabrück that this thesis and the work presented therein are my own work, and that all referenced material has been listed properly in the bibliography.

Osnabrück, May 2006 _____