The Reaction Asynchronous Programming Framework

Christopher J. Holgate

Version 0.06 (August, 2016)

Contents

Chapter 1

Introducing the Reaction Framework

Reaction is a flexible asynchronous programming framework which may be used to implement complex event-driven applications. It is heavily influenced by the Twisted programming framework developed by TwistedMatrix Labs [3] for the Python programming language. While event-driven programming is not a new concept, the approach adopted by the Reaction and Twisted frameworks has a certain elegance which makes it applicable to a wide range of problem domains.

1.1 Why Asynchronous Programming?

Generally speaking, asynchronous programming is applicable to problems where the programmer has to deal with multiple tasks which can take an arbitrarily long time to run. This may be down to the tasks in question requiring a significant amount of computation or needing to interact with a remote system. In a naive programming approach these tasks might be wrapped inside standard functions. A function call can then be issued by the application when the task is to start and it will return to the application once the task is complete.

This naive approach suffers from the fairly obvious problem that the application is blocked from further execution until the long running task is complete. It also prevents multiple such tasks from executing concurrently. More realistic implementations would make use of an additional thread to process the long running task. This allows the application to continue execution and even start up additional long running tasks in parallel with the first. The application may then periodically poll the long running tasks to check for completion.

The model of farming out long running tasks and then either polling for or waiting on completion is a common design pattern – so much so that it has been formalised in the Java 5.0 concurrency library [5] using the `Callable` and `Future` interfaces.

Listing 1.1 A Simple Callback Interface

```
public interface Callback {
  public void onCallback (Object result);
}
```

Listing 1.2 A Simple Long Running Task Interface

```
public interface LongRunningTask {
  public void run (Callback callback, Object params);
}
```

The reaction framework takes a different approach to handling task completion in that it makes use of the callback idiom, whereby long running tasks issue callbacks to the application on completion. This eliminates the need for the application to poll for or wait for the task to complete. Callbacks are commonly used in programming languages such as C/C++ which provide native function pointers, but are rarer in Java based applications due to the lack of function pointer support. That said, it is actually quite easy to implement Java based callbacks by defining a suitable callback interface, as shown in Listing 1.1.

Given an application object which implements this interface, it is then possible to support callbacks from long running tasks by passing a reference to the callback interface when the task is started up, as shown in Listing 1.2.

On completing execution, the long running task then only needs to make a call to `callback.onCallback(result)` in order to pass the result of executing the task back to the application.

As can be seen from the preceding example, using callbacks to indicate completion of long running tasks can provide a very elegant API. However, this does not address the underlying threading and synchronisation issues which are inherent in a callback based design. To make such a programming model work it therefore becomes necessary to provide an underlying framework which hides these complexities from the application developer.

1.2 Adding a Reactor

An asynchronous programming framework which supports the callback model has many similarities to conventional event-driven frameworks such as GUI toolkits. Central to this is the concept of an event loop which detects events and dispatches callbacks to any registered event handlers. In the context of a GUI toolkit an event would typically be a button click or keystroke; for the asynchronous programming framework the event would signal the completion of a long running task.

A significant consideration is that the use of an event loop implies that all callbacks will be made within the context of the event loop thread. This

Listing 1.3 A Basic Threadable Interface Definition

```
public interface Threadable {
  public Object run (Object data)
      throws Exception;
}
```

essentially removes the need for the application developer to deal with thread synchronisation, at the expense of the constraint that callback methods cannot block or be long running.

Within the Reaction asynchronous programming framework, the component which provides the event loop is referred to as the *reactor*. This term originally derives from the reactor design pattern [7], although the features of the reactor used in the Reaction framework have been expanded considerably from the standard pattern description.

In addition to providing the event loop for dispatching callbacks, the reactor is also responsible for scheduling long running tasks. This means that instead of the application starting up such a task directly it can delegate the whole messy business to the reactor. In fact this can be reduced to the point where the long running task only needs to implement a single conventional method, as illustrated by Listing 1.3.

This is a basic version of the interface definition for long running tasks which is implemented by the Reaction framework. In keeping with the interface naming, such tasks will subsequently be referred to as *threadable* objects. Execution of the `run` method will be farmed out to a worker thread by the Reactor and on completion the Reactor will take the returned value and pass it back to the application as the callback parameter. All thread creation and synchronisation is taken care of by the reactor component.

1.3 Callback Error Handling

Any attempt to execute the `run` method on a threadable object can result in an exception being thrown. This is to be expected, as bad things do happen and exceptions are the correct way of dealing with such an eventuality. The problem with throwing an exception in this context is that information about the exception condition needs to be propagated back to the application so that it can take the appropriate action. This is solved by extending the application's callback interface as shown in Listing 1.4.

Again, this is a basic version of the interface definition which is used to define callbacks for the Reaction framework. As well as the original callback method, the `onErrback` method has been added which may be used for passing back exception conditions detected while executing a threadable object. In keeping with the name of the extended interface, objects which implement `Deferrable` are subsequently referred to as *deferrable objects*.

Listing 1.4 A Basic Deferrable Interface Definition

```
public interface Deferrable {
  public Object onCallback (Deferred deferred, Object data)
      throws Exception;
  public Object onErrback (Deferred deferred, Exception error)
      throws Exception;
}
```

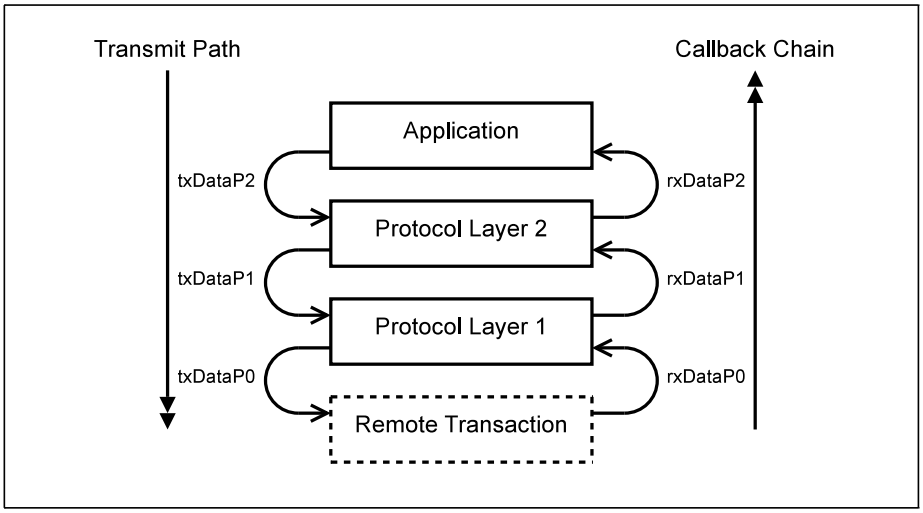

Figure 1.1: Callback Chaining for Protocol Stacks

1.4 Callback Chaining

A further enhancement to the callback idiom is the concept of callback chaining. This is particularly useful in the context of layered protocol stacks, such as those conforming to the OSI protocol model. An example of such a layered protocol stack is shown in Figure 1.1.

The example illustrates a request for remote data via the protocol stack. As the request is passed down the protocol stack, the request data is modified according to the protocol requirements – implementing packetisation, encryption, framing etc. At the same time, it is also possible to build up a callback chain for handing the response.

The presence of a callback chain for the transaction then allows the response data to be passed back up the stack using the callback mechanism. Within each callback method the received data may then be processed according to the layered protocol requirements. This is the reason that the simplified `onCallback` method returns a data Object – the returned value of each callback can be passed

Listing 1.5 A Basic Signalable Interface Definition

```
public interface Signalable {
  public void onSignal (Signal signal, final Object data);
}
```

Listing 1.6 A Basic Timeable Interface Definition

```
public interface Timeable {
  public void onTick (Object data);
}
```

directly into the next callback in the chain for further processing.

While conceptually elegant, the process of dynamically constructing and then executing callback chains is a non-trivial exercise. Fortunately the Reaction framework provides a simple mechanism for managing this process called *deferred events*. These will be discussed in more detail in Chapter 5.

1.5 Signal Events

In Section 1.2 the Reactor component was described as being similar to a conventional event loop. This means that it is quite capable of processing external events such as GUI button presses or the receipt of an automatic data sensing event. In the context of the Reaction framework, this type of event is distinguished by referring to it as a *signal event*.

Objects which can receive signal events are referred to as *signalable objects* and implement the `Signalable` interface. A basic version of this interface is shown in Listing 1.5.

Multiple signalable objects may be attached to a single signal source, with the signal parameter data being passed to each signalable object in turn. The implementation of signalable objects and their relation to signal events is discussed in more detail in Chapter 4.

1.6 Timed Events

A final type of event supported by the Reaction framework are *timed events*. In many applications it is necessary to schedule timeouts on long running transactions, and this is one application of timed events. An alternate use is to generate periodic events for triggering tasks which must be carried out at regular intervals. Objects which can receive timed events are referred to as *timeable objects* and implement the `Timeable` interface. A basic version of the `Timeable` interface is shown in Listing 1.6.

Listing 1.7 Type Safe Timeable Interface Definition

```
public interface Timeable<T> {
  public void onTick(T data);
}
```

Timed events are probably the simplest form of event in the Reaction framework repertoire, which is why they are the first to be discussed in detail in Chapter 3.

1.7 Using Java Generics For Type Safety

One thing that is noticeable about the basic interface definitions previously presented is that many of them pass an arbitrary `data` parameter which uses the Java root object type, `Object`. This is because the Reaction framework treats all internal event data as opaque, arbitrary objects. This may seem to be an anathema in the context of a strongly statically typed language such as Java, but there are valid reasons for making this design decision.

A key property of the Reaction framework is that it allows for the run-time dynamic configuration of program flow by attaching and triggering callbacks. This dynamic behaviour means that the Reaction framework has no a-priori information about the parameter data types to be used for these callbacks - and the only reasonable assumption to make is that they are all subclassed from the common Java root object type.

Even though all callback parameter data is internally treated as arbitrary opaque objects, type safety may be restored at the interfaces by making use of Java generics. An example of the use of Java generics to enforce type safety on a timeable interface is shown in Listing 1.7.

The type safe implementation of the interface parameterises the basic version shown in Listing 1.6 by the addition of the generic type specifier `<T>`. This enforces a compile time constraint on the interface, such that only objects of type `T` may be passed to the `onTick` callback method. All the basic interface specifications previously presented in this introduction will adopt similar type parameterisation in order to enforce type safety throughout the Reaction framework.

1.8 Target Platforms

The Reaction programming framework is intended for use as either a standalone package in plain old Java objects (POJO) based applications or as a service module within the OSGi framework [2].

The majority of this book describes the framework as a standard POJO package, since this is the easiest way of demonstrating the various features.

Chapter 8 expands on this to discuss the use of the Reaction framework as an OSGi service, for those interested in this mode of operation.

The extensive use of Java generics in order to provide support for type safe callbacks means that the Reaction framework is only suitable for use with Java compilers from release 5.0 onwards. However, the Reaction framework adopts the same approach to backward compatibility as used in OSGi release 4.3. This means that the Reaction framework can still be compiled for the JSR14 compiler target which is required for Java CDC support.

1.9 The Reaction Packages

Before introducing the various components of the Reaction framework in detail, it is worth reviewing the package structure of the framework distribution. The root package name is `com.zynaptic.reaction` and this is subdivided into the following packages:

`com.zynaptic.reaction`

This is the API package which provides the complete external API for the Reaction framework. Application code only needs to include the API package in order to make full use of an active reactor. When running as an OSGi service, the API package is the only package exported by the Reaction service.

`com.zynaptic.reaction.core`

The core package provides the main functionality of the Reaction framework. In addition, it provides the API which is used for managing the lifecycle of the reactor component. When using the Reaction framework in a POJO application, the application is able to use this API to initialise and run the application. When used as an OSGi service this process is automated by the OSGi bundle activator.

`com.zynaptic.reaction.util`

The utility package contains common support classes for the Reaction framework. It is provided because certain aspects of the framework setup, such as logging and the monotonic clock source will vary between deployments. The relevant components have been collected in the utility package in order to provide a range of alternate implementations.

`com.zynaptic.reaction.osgi`

The OSGi package includes the additional infrastructure code which is required to wrap the Reaction framework as an OSGi service.

`com.zynaptic.reaction.examples.reactor`
`com.zynaptic.reaction.examples.timer`
`com.zynaptic.reaction.examples.signal`

```
com.zynaptic.reaction.examples.deferred
com.zynaptic.reaction.examples.thread
com.zynaptic.reaction.examples.osgi
com.zynaptic.reaction.examples.swing
com.zynaptic.reaction.examples.javafx
```

The example packages contain source code for the various programming examples described in this text. Where an example is given in the text, the corresponding file in the examples package will be indicated.

1.10 Chapter Summary

This chapter provided a very brief outline of the asynchronous programming model and introduced some of the features that the Reaction framework provides in order to support it. However, in order to really appreciate the advantages that asynchronous programming can bring it is necessary to dig a little deeper. The following chapters introduce the various aspects of the Reaction framework in more detail.

Chapter 2

The Reactor Core

Central to the Reaction framework is the reactor core. This component is responsible for scheduling events, thread management and acting as a factory for other framework objects. This chapter introduces the reactor core and its ancillary components and describes the steps required in order to start up the reactor ready for use by the application. However, before activating the reactor itself, we need to put in place a monotonic clock and some logging support.

2.1 The Monotonic Clock

The first ancillary component required by the reactor is a monotonic clock. A monotonic clock is essentially a counter which when read returns a value which is guaranteed never to be less than any previously read value and which increments at a fixed rate. This seems to be such an obviously useful feature that there must be a standard Java implementation; and there is – sometimes.

Early versions of Java only provided the `System.currentTimeMillis` call, which returns the integer number of milliseconds since the UNIX epoch at midnight on the 1st of January 1970. This is the *wallclock* time which can typically be adjusted by the system user. Because the wallclock time is adjustable in this way, the values returned by `System.currentTimeMillis` cannot be guaranteed to be monotonic – being prone to large jumps forwards and backwards as the wallclock time is adjusted.

The situation was improved with the addition of the `System.nanoTime` call to Java 5.0. This will attempt to use the host system's monotonic clock when available. However, not all host systems provide a monotonic clock and under those circumstances `System.nanoTime` falls back to using the non-monotonic wallclock.

A final complication is that if the host does not have a standard monotonic clock, there may be one available as a hardware peripheral. This may be the case when running on an embedded platform or using a hardware security peripheral.

Given that there is no standard way of effectively implementing a monotonic

Listing 2.1 Interface Definition for Monotonic Clock Source

```
package com.zynaptic.reaction.util;

public interface MonotonicClockSource {
  public void init ();
  public long getMsTime ();
}
```

clock source across all possible target platforms this functionality has been factored out of the reactor core. Alternate monotonic clock implementations can therefore be provided as part of the utility package. The common interface to a monotonic clock implementation is defined in Listing 2.1.

The `init` method is called on reactor startup and is used to reset the monotonic clock to zero. This means that any value subsequently read back is equivalent to the reactor's uptime. The `getMsTime` method returns the current value of monotonic clock's timer counter. This encodes the number of milliseconds since the clock was reset as a Java native long integer.

There are three standard implementations of the monotonic clock source present in the utility package. Each one is applicable to a different host environment, as discussed in the following sections.

2.1.1 Fixed Up Wallclock

This monotonic clock source is the most generally applicable implementation and should work across all supported platforms. It uses the standard wallclock as a clock source and then attempts to detect and correct for any user adjustments. This ensures that the clock source is monotonic, but it will increment at an unpredictable rate whenever adjustments to the wallclock are made. This is the default option since it is applicable to all platforms and is provided in the utilities package as `FixedUpMonotonicClock`.

2.1.2 Java Monotonic Clock

This monotonic clock source may be used when running the Reaction framework in a Java environment which provides `System.nanoTime` and the underlying operating system is known to provide the JVM with a monotonic clock. This is typically the case with POSIX compliant systems that provide `CLOCK_MONOTONIC` support. This option is provided in the utilities package as `JavaMonotonicClock`.

2.1.3 Native Monotonic Clock

This monotonic clock source may be used when running the Reaction framework on a platform which provides a non-standard native monotonic clock, often as

Listing 2.2 Interface Definition for Reactor Log Target

```
package com.zynaptic.reaction.util;

import java.util.MissingResourceException;
import com.zynaptic.reaction.Logger;

public interface ReactorLogTarget {
  public Logger getLogger(String loggerId, String loggerResources)
      throws MissingResourceException;
}
```

Listing 2.3 Public Reactor API for Accessing Reactor Log Target

```
package com.zynaptic.reaction;
import java.util.MissingResourceException;

public interface Reactor {
  ...
  public Logger getLogger(String loggerId);
  public Logger getLogger(String loggerId, String loggerResources)
      throws MissingResourceException;
  ...
}
```

a dedicated hardware peripheral. This assumes that monotonic clock support is provided using native routines accessed via the Java native interface [6]. The class definition is provided in the utilities package as `NativeMonotonicClock` but no native library is provided as this is obviously platform specific.

2.2 Logging Support

The second ancillary component which is required by the reactor is a message logger. There are a number of Java logging frameworks available and the choice of the 'best' logging framework will differ between applications. Therefore the logging functionality has been factored out of the reactor core, allowing different logging framework wrappers to be provided as part of the utility package.

The common logging interface is defined as shown in Listing 2.2. This acts as a factory which allows multiple logger objects to be created for different parts of the overall system. These factory methods are exposed via the public reactor API as shown in Listing 2.3.

The public interface for each of the logger components is shown in Listing 2.4 and makes use of the logging level enumeration provided by the standard Java logging library. Logger objects will usually be created using unique identifiers that employ the conventional hierarchical naming system. This means that

Listing 2.4 Interface Definition for Reactor Logger Objects

```
package com.zynaptic.reaction;
import java.util.logging.Level;

public interface Logger {
  public void log(Level level, String msg);
  public void log(Level level, String msg, Throwable thrown);
  public String getLoggerId();
  public Level getLogLevel();
  public void setLogLevel(Level logLevel);
}
```

the core reactor functionality will be logged with the identification name of 'com.zynaptic.reaction'.

The common logging interface is a fairly standard logging API, providing a set of defined log message severity levels and some methods for logging messages at one of these severity levels. This can easily be mapped onto existing logging frameworks using one of the wrappers described in the following subsections.

2.2.1 System Console Output

The simplest logging option involves printing all log messages to the console output. This can be useful during early development and is the default option used in the examples presented in this document. The system console log component is present in the utilities package as `ReactorLogSystemOut`.

2.2.2 Java Standard Logging Library

The most complete logging support is currently provided by a wrapper for the standard Java logging library (`java.util.logging`). This maps the members of the generic interface shown in Listing 2.4 onto the equivalent calls used by the conventional Java logger components. When used in this manner, the logging subsystem behaves in the same way as would be expected when directly accessing the standard Java logging library.

2.2.3 OSGi Logging Service

A wrapper around the OSGi log service API is included for situations where the Reaction framework is being run in an OSGi context. The wrapper is designed for use when the OSGi log service is declared as an optional import package for the Reaction bundle. This means that the OSGi logging package must be available when the Reaction bundle is resolved in order for it to be used.

If the OSGi logging service is not available on startup – or subsequently becomes unavailable during operation – logging is redirected to a backup log service instead. The backup log service to be used is specified as part of the

Listing 2.5 Reactor Interface Accessor Methods

```
package com.zynaptic.reaction.core;

public final class ReactorCore... {
  public static Reactor getReactor() {...}
  public static ReactorControl getReactorControl() {...}
  ...
}
```

Listing 2.6 Reactor Control Interface Definition

```
package com.zynaptic.reaction.core;

public interface ReactorControl {
  public void start(MonotonicClockSource clockSource,
    ReactorLogTarget logTarget)
      throws ThreadableRunningException;
  public void stop();
  public void join()
      throws InterruptedException;
}
```

constructor for the OSGi log service wrapper and will typically be an instantiation of the console output logger. The OSGi log service wrapper component is present in the OSGi package as `ReactorLogOsgiOptional`.

2.3 The Reactor Singleton Object

Within any application there will only be a single reactor component. This is implemented as the `ReactorCore` singleton, which provides static accessor methods for obtaining a handle on its various interfaces. The accessor methods of interest are shown in Listing 2.5. All access to the reactor itself should be via these two interfaces - the `Reactor` interface providing the user API and the `ReactorControl` interface providing control over the reactor lifecycle.

2.4 The Reactor Control Interface

The reactor control interface is used to control the reactor lifecycle and is a part of the core package. Direct access to this API is only required for application startup code and it should not be accessed by more general application code. The reactor control interface is defined as shown in Listing 2.6.

The reactor creates its own thread for running the event loop, so three reactor control functions are provided for controlling it from the main program

Listing 2.7 Simple Reactor Startup Example

```
public class ReactorStartupExample {
  public static void main(String[] args) throws InterruptedException {

    // Obtain a handle on the reactor control interface.
    ReactorControl reactorCtrl = Reactor.getReactorControl();

    // Start reactor with new monotonic clock and log service.
    reactorCtrl.start
      (new FixedUpMonotonicClock(), new ReactorLogSystemOut());

    // Let the reactor run for a few seconds.
    Thread.sleep(5000);

    // Request that the reactor stop running.
    reactorCtrl.stop();

    // Wait for the reactor to stop running.
    reactorCtrl.join();
  }
}
```

thread. Note that the API calls are synchronised, so they may be called from other thread contexts if required. The first method is the `start` method which passes in references to the monotonic clock and log service components which are to be used by the reactor. Calling this method creates the reactor thread and starts the reactor running before returning to the main thread.

In order to stop the reactor, a call to the `stop` method will initiate reactor shutdown, requesting that the reactor thread be stopped and all outstanding events either cancelled or dispatched. The `stop` method returns immediately and the main application thread must then wait for the reactor shutdown to complete. This is done by calling the `join` method which will block until the reactor thread has exited.

2.5 Starting and Stopping the Reactor

The process of starting and stopping the reactor may be carried out using only the `ReactorControl` interface, typically in the context of the main application thread. A simple example of starting and stopping the reactor from the main application thread is shown in Listing 2.7. This is present in the `reactor` examples package as `ReactorStartupExample`.

This example includes all the code which is required for managing the Reactor lifecycle in a standard Java application. When the reactor is run as an OSGi service this will all be done automatically, since the reactor lifecycle is managed by the OSGi container.

Listing 2.8 Reactor Uptime Getter Method

```
package com.zynaptic.reaction;

public interface Reactor {
  ...
  public long getUptime();
  ...
}
```

Listing 2.9 Example Code for Accessing Reactor Uptime

```
...
  long reactorUptime;
  do {
    Thread.sleep(1000);
    reactorUptime = reactor.getUptime();
    System.out.println("Current reactor uptime = " +
      reactorUptime + " ms.");
  } while (reactorUptime < 10000);
...
```

2.6 Obtaining The Reactor Uptime

The reactor's user interface provides a mechanism for reading back the current time, as given by the monotonic clock source. This may be interpreted as the current uptime of the reactor component, specified as an integer number of milliseconds using a Java native long integer. The API for accessing the current reactor uptime is illustrated in Listing 2.8.

The reactor's `getUptime` method may be called from any thread context in order to obtain the current uptime. An example of this method call in use is illustrated by the code fragment shown in Listing 2.9. This code fragment may be used to replace the delay introduced by the `Thread.sleep` call in the reactor lifecycle example of Listing 2.7. A full implementation of this example is included in the `reactor` examples package as `ReactorUptimeExample`.

Chapter 3

Working With Timers

The simplest form of event provided by the Reaction framework is the *timed event*. These are events which occur a set period of time after the event was requested and which are handled by event handlers implementing the `Timeable` interface. This chapter describes the Reaction timer API and provides a number of examples of the API in use.

3.1 The Reaction Timer API

The methods used for managing timer events are provided by the standard reactor interface (`Reactor`), as defined in the API package. The full set of methods used for timer management are shown in Listing 3.1.

The three timer API methods provide support for issuing a single timed callback (`runTimerOneShot`), issuing multiple timed callbacks at regular intervals (`runTimerRepeating`) and cancelling a running timer (`cancelTimer`). In each case a single timeable object which provides the `Timeable` interface is passed as a parameter. This is the timeable object to which the callbacks will be issued and is also used as a timer 'ID' when cancelling or rescheduling a timer. The use of these various API methods is described in more detail in the following sections.

3.2 Running One-Shot Timers

One shot timers are used to generate a single timed event a specific period of time after the timed event was requested. There are no constraints over which thread actually issues the request, so it is perfectly legitimate to request a timed event from the main application thread, as shown in Listing 3.3. This code is present in the `timer` examples package as `OneShotTimerExample1`.

The example shown in Listing 3.3 uses the timed callback to shut down the reactor after a specific amount of time. The reactor startup code is identical to that previously described in Section 2.5. However, rather than issuing the

Listing 3.1 Reactor Timer Management Methods

```
package com.zynaptic.reaction;

public interface Reactor {
  ...
  public <T> void runTimerOneShot
    (Timeable<T> timeable, int msDelay, T data)
      throws ReactorNotRunningException;
  public <T> void runTimerRepeating
    (Timeable<T> timeable, int msDelay, int msInterval, T data)
      throws ReactorNotRunningException;
  public void cancelTimer(Timeable<?> timeable);
  ...
}
```

Listing 3.2 The Timeable Interface Definition

```
package com.zynaptic.reaction;

public interface Timeable<T> {
  public void onTick(T data);
}
```

reactor stop request from the main thread it is instead implemented within a timeable callback.

In order for the timeable callback to issue the reactor stop request it must have a handle on the reactor's control interface. In the example this is passed as the timer's `reactorCtrl` parameter. The `reactorCtrl` parameter is a reference to the reactor control interface which is held by the timer and then passed on to the timeable callback once the timer has expired. The use of a generic type specification ensures that only objects of the correct type may be passed in this manner.

In the event that an invalid `reactorCtrl` parameter is passed to a timeable callback an exception will be thrown. All exceptions thrown by timeable callbacks are caught by the reaction framework and will be logged with a log level of *warning*. The continued operation of the reactor is unaffected. In the example, the second timer request passes a null reference and will therefore generate an exception.

One thing which must be taken into account when using timers is that the specified delay must be treated as a *minimum* delay parameter. This is because the timer callback will be issued an arbitrary – but usually small – time after the delay period has expired. This delay may be caused by underlying operating system context switches, Java thread scheduling artefacts or long-running callbacks which are active at the point when the delay period expires.

Listing 3.3 Running a One Shot Timer

```
public class OneShotTimerExample1 {
  public static void main(String[] args) throws InterruptedException {

    // Get a handle on the reactor interfaces.
    ReactorControl reactorCtrl = ReactorCore.getReactorControl();
    Reactor reactor = ReactorCore.getReactor();

    // Start the reactor using monotonic clock and log service.
    reactorCtrl.start
      (new FixedUpMonotonicClock(), new ReactorLogSystemOut());

    // Request a callback from a one-shot timer.
    reactor.runTimerOneShot (new TimerCallback(), 5000, reactorCtrl);

    // Request a callback which will throw an exception.
    reactor.runTimerOneShot (new TimerCallback(), 1000, null);

    // Wait for the reactor to stop running.
    reactorCtrl.join();
  }

  // The timer callback just stops the reactor when called.
  private static class TimerCallback
      implements Timeable<ReactorControl> {
    public void onTick(ReactorControl reactorCtrl) {
      reactorCtrl.stop();
    }
  }
}
```

Figure 3.1 shows the delay in issuing callbacks relative to the fixed reactor timebase.

3.3 Cancelling One-Shot Timers

Once a timer is running it is possible to subsequently cancel the timer in order to prevent it issuing its timeable callback. This may be illustrated by using timer cancellation to prevent the erroneous callback in Listing 3.3 from triggering and therefore generating an exception. Listing 3.4 shows the changes required. The full code for this example is included in the `timer` examples package as `OneShotTimerExample2`.

In Listing 3.4 a reference to the timeable callback object is passed to the `cancelTimer` method. There is a one-to-one mapping between timers and timeable callback objects which means that the reference to the timeable object can

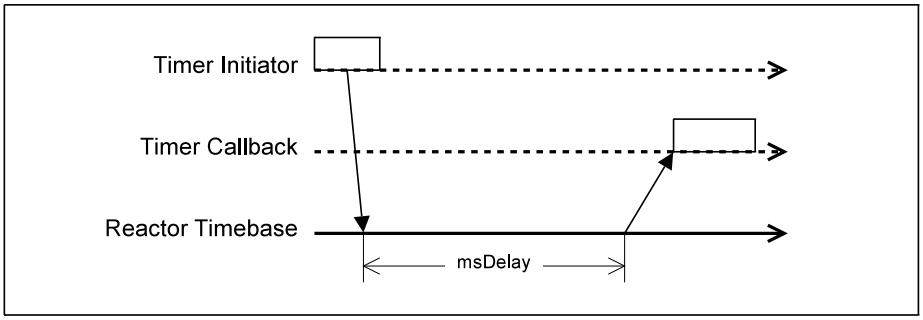

Figure 3.1: Timing Diagram For One Shot Timeouts

Listing 3.4 Cancelling a One Shot Timer

```
...
    // Request a callback which will throw an exception.
    Timeable<ReactorControl> badCallback = new TimerCallback();
    reactor.runTimerOneShot(badCallback, 1000, null);

    // Cancel the erroneous callback.
    reactor.cancelTimer(badCallback);
...
```

be used to select the timer for cancellation. Note that this means that it is not possible to attach multiple callbacks to a single timer.

Timer cancellation is particularly useful for implementing the *timeout* idiom. In many applications there are long running tasks where a failure to complete the task within a specified time constitutes an error. An example might be a request from a remote database which if not serviced within a given time implies that the remote database server is inaccessible. This works by requesting a timer callback on issuing the remote request and cancelling it on receiving a response. In the event that a callback to the timeable callback object is actually issued this will be interpreted as a timeout error.

3.4 Rescheduling One-Shot Timers

In addition to being able to cancel running timers, it is also possible to reschedule them – effectively resetting the timer delay to a new value. This may be demonstrated by using timer rescheduling to delay the reactor shutdown in Listing 3.3. The changes required are shown in Listing 3.5. As may be seen from the example, rescheduling a timer is simply a case of calling the `runTimerOneShot` method with a reference to the associated timeable callback object and the new timer delay. The full code for this example is included in the `timer` examples

Listing 3.5 Rescheduling a One Shot Timer

```
...
    // Request a callback from a one-shot timer.
    Timeable<ReactorControl> timerCallback = new TimerCallback();
    reactor.runTimerOneShot(timerCallback, 5000, reactorCtrl);

    // Reschedule callback after a short delay.
    Thread.sleep(2500);
    reactor.runTimerOneShot(timerCallback, 5000, reactorCtrl);
...
```

package as `OneShotTimerExample3`.

Timer rescheduling may be used to implement the *watchdog timer* idiom. This provides a mechanism for monitoring a remote system which provides a regular 'heartbeat' or 'liveness' notification. Each time a heartbeat event is received the timer is rescheduled, preventing it from issuing the timer callback. However, if the remote system becomes inoperable it will stop sending heartbeat messages and the timer will expire. This causes the timer to issue a callback to the timeable callback object. The callback may then be interpreted as notification that the remote system is no longer available.

3.5 Running Repeating Timers

Repeating timers are used to generate multiple timed event callbacks at fixed intervals. Each timer is associated with a single timeable callback handler implementing the `Timeable` interface. The `onTick` callback method of the timeable callback object will then be called repeatedly at the specified interval until the timer is either cancelled or rescheduled. In other respects, repeating timers behave identically to the one-shot timers previously discussed.

An example of a repeating timer in use is shown in Listing 3.6. This is similar to previous one-shot timer examples, except that a number of callbacks have to be made before the callback handler stops the reactor. The full code for this example is included in the `timer` examples package as `RepeatingTimerExample1`.

When running a repeating timer there are two timing parameters which are passed; the first sets the delay before the initial callback is made and the second sets the interval between successive callbacks. In this example the first callback will occur 5 seconds after the timer is initiated, with successive callbacks occurring at 1 second intervals.

One thing to consider when using repeating timers is that while the timebase interval is constant, each callback can be subject to an arbitrary – but usually short – delay. As for the case with one-shot timers, these additional delays may be down to waiting for a running callback to complete or may be artefacts of the underlying operating system and the Java thread library. The variability in the callback timing is referred to as the timer *jitter* and is illustrated in Figure 3.2.

Listing 3.6 Running a Repeating Timer

```
public class RepeatingTimerExample1 {
  public static void main(String[] args) throws InterruptedException {

    // Get a handle on the reactor interfaces.
    ReactorControl reactorCtrl = ReactorCore.getReactorControl();
    Reactor reactor = ReactorCore.getReactor();

    // Start reactor with new monotonic clock and log service.
    reactorCtrl.start
      (new FixedUpMonotonicClock(), new ReactorLogSystemOut());

    // Request callbacks from a repeating timer.
    reactor.runTimerRepeating
      (new TimerCallback(), 5000, 1000, reactorCtrl);

    // Wait for the reactor to stop running.
    reactorCtrl.join();
  }

  // Timer callback handler stops the reactor after 5 callbacks.
  private static class TimerCallback
      implements Timeable<ReactorControl> {
    private int callbackCount = 5;
    public void onTick(ReactorControl reactorCtrl) {
      callbackCount--;
      if (callbackCount == 0) {
        reactorCtrl.stop();
      }
    }
  }
}
```

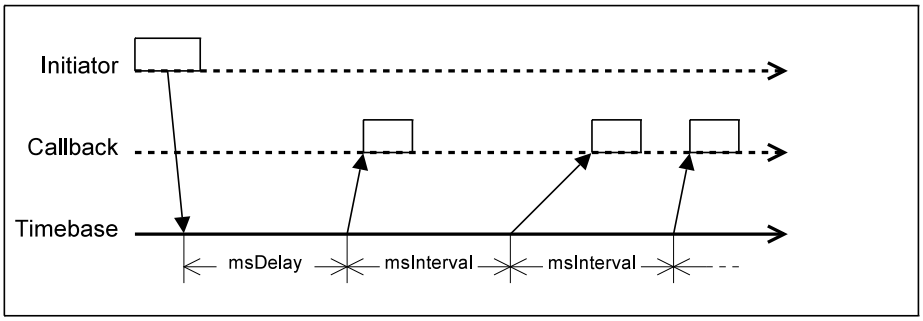

Figure 3.2: Example of Timing Jitter For Repeating Timers

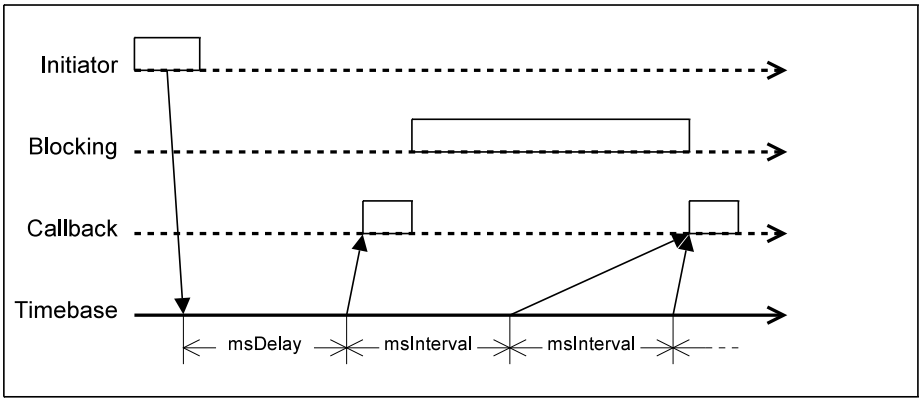

Figure 3.3: Example of Callback Merging For Repeating Timers

In extreme cases it is possible that a timer callback will be delayed by more than the specified timer interval. In this case, two or more delayed callbacks will be merged into a single timer callback, as shown in Figure 3.3. Such a situation should be treated as an indication that the system is overloaded and is unable to process all application callbacks in a timely manner. Therefore, all such conditions are logged as warning conditions.

The most pathological case of timer event merging occurs when the callback itself takes longer to execute than the specified timer interval. This is demonstrated by adding an excess delay to the timer callback shown in Listing 3.6. The full example demonstrating this behaviour is included in the `timer` examples package as `RepeatingTimerExample2`. While the minimum viable timer interval will be platform and application dependent, it is generally recommended that only timer intervals in excess of a quarter of a second (250ms) are used.

3.6 Rescheduling Repeating Timers

The cancellation and rescheduling of repeating timers behaves identically to the cancellation and rescheduling of one-shot timers. However, there is one non-obvious way in which timers may be rescheduled. Specifically, it is possible to reschedule a repeating timer as a one-shot timer and vice-versa. In practise this is because a one-shot timer is equivalent to a repeating timer with the interval field set to zero – interpreted as 'don't repeat'.

The example shown in Listing 3.7 demonstrates the way in which a repeating timer may be rescheduled as a one-shot timer. This is a minor change to the example previously given in Listing 3.6. On the penultimate call to the timeable callback handler the timer is now rescheduled as a one-shot timer with a 5 second delay. The full code for this example is included in the `timer` examples package as `RepeatingTimerExample3`.

Listing 3.7 Rescheduling a Repeating Timer as a One-Shot Timer

```
...
  private static class TimerCallback
      implements Timeable<ReactorControl> {
    private int callbackCount = 5;
    public void onTick(ReactorControl reactorCtrl) {
      callbackCount--;
      if (callbackCount == 0) {
        reactorCtrl.stop();
      } else if (callbackCount == 1) {
        ReactorCore.getReactor().runTimerOneShot
          (this, 5000, reactorCtrl);
      }
    }
  }
...
```

A final thing of note about the example shown in Listing 3.7 is that the timer is being rescheduled within the timeable callback handler. This is legitimate since the reactor is designed in such a way that there are no problems with rescheduling or cancelling a timer from within its own callback handler.

Chapter 4

Using Signal Events

The *signal event* is intended for use in situations where a single event needs to be propagated to multiple event handlers. In this case the event handlers are *signalable* objects which implement the `Signalable` interface. This chapter examines signal events in more detail and introduces the concept of *signal event objects* which are used for managing signal propagation to signalable event handlers.

4.1 The Signal Event API

Signal events are manipulated within the Reaction framework using signal event objects. The reactor core acts as a factory for these objects and the standard reactor interface (`Reactor`) provides the API shown in Listing 4.1.

The signal factory methods will return an object which implements the `Signal` interface as shown in Listing 4.2. Such objects are referred to here as being *signal event objects* since they encapsulate all the state associated with a single signal. The `Signal` interface provides methods for subscribing and unsubscribing signalable objects (ie, those implementing `Signalable`) and for generating new signal events notifications.

4.2 Receiving Signal Events

Listing 4.4 provides the simplest possible demonstration of signal event handling. The reactor provides a special signal which is used to indicate that the reactor is shutting down – and it is possible to obtain a handle on this signal via the `getReactorShutdownSignal` method on the `Reactor` interface. This call returns a reference to the signal event object which is responsible for propagating the reactor's shutdown signal notification.

Signal event objects implement the `Signal` interface which allows signalable event handlers to be attached and removed from the signal. In this example, a signalable callback is attached to the reactor's shutdown signal using the

Listing 4.1 Reactor Signal Factory Methods

```
package com.zynaptic.reaction;

public interface Reactor {
  ...
  public <T> Signal<T> newSignal();
  public Signal<Integer> getReactorShutdownSignal();
  ...
}
```

Listing 4.2 Signal Event Interface Definition

```
package com.zynaptic.reaction;

public interface Signal<T> {
  public void subscribe(Signalable<T> signalable)
    throws SignalContextException, ReactorNotRunningException;
  public void unsubscribe(Signalable<T> signalable)
    throws SignalContextException, ReactorNotRunningException;
  public void signal(final T data)
    throws RestrictedCapabilityException, ReactorNotRunningException;
  public void signalFinalize(final T data)
    throws RestrictedCapabilityException, ReactorNotRunningException;
  public Signal<T> makeRestricted();
}
```

`subscribe` method. When the reactor shutdown signal is triggered, it is then propagated to all subscribed signalable event handlers. In this case there is only a single handler, which prints a reactor shutdown message to the console. The full source code for the example shown in Listing 4.4 is provided in the `signal` examples package as `ReactorShutdownSignal`.

4.3 Generating Signal Events

The example given in Listing 4.4 demonstrated the way in which signalable event handlers can be registered with a signal event object in order to receive signal event notifications. In that example, signal events are generated internally to the reactor core to provide notification of reactor shutdown. However, in the general case signal events will need to be generated by application code instead.

The way in which signal events may be generated by application code is shown in Listing 4.5. The full source code for this example is available in the `signal` examples package as `SignalEventExample1`.

New signal event objects are created using the `newSignal` factory method on the `Reactor` interface. In common with the reactor shutdown signal in the

Listing 4.3 The Signalable Interface Definition

```
package com.zynaptic.reaction;

public interface Signalable<T> {
  public void onSignal(Signal<T> signalId, final T data);
}
```

Listing 4.4 Receiving Reactor Shutdown Signal Events

```
public class ReactorShutdownSignal {
  public static void main(String[] args) throws InterruptedException {

    // Start reactor with new monotonic clock and log service.
    ReactorControl reactorCtrl = ReactorCore.getReactorControl();
    Reactor reactor = ReactorCore.getReactor();
    reactorCtrl.start
      (new FixedUpMonotonicClock(), new ReactorLogSystemOut());

    // Add a signal event handler to the reactor shutdown signal.
    Signal<Integer> shutdownSignal = reactor.getReactorShutdownSignal();
    shutdownSignal.subscribe(new ShutdownSignalCallback());

    // Let the reactor run for a few seconds.
    Thread.sleep(5000);

    // Request that the reactor stop running.
    reactorCtrl.stop();
    reactorCtrl.join();
  }

  // Just print a shutdown message to the console.
  private static class ShutdownSignalCallback
      implements Signalable<Integer> {
    public void onSignal(Signal<Integer> signal, Integer status) {
      System.out.println("Reactor shutdown status = " + status);
    }
  }
}
```

Listing 4.5 Generating Signal Events

```
public class SignalEventExample1 {
  public static void main(String[] args) throws InterruptedException {

    // Start reactor with new monotonic clock and log service.
    ReactorControl reactorCtrl = ReactorCore.getReactorControl();
    Reactor reactor = ReactorCore.getReactor();
    reactorCtrl.start
      (new FixedUpMonotonicClock(), new ReactorLogSystemOut());

    // Add a signal event handler to a new signal event object.
    Signal<ReactorControl> signal = reactor.newSignal();
    signal.subscribe(new SignalCallback());

    // Generate a signal event and wait for reactor to stop.
    signal.signal(reactorCtrl);
    reactorCtrl.join();
  }

  // The signal handler stops the reactor.
  private static class SignalCallback
      implements Signalable<ReactorControl> {
    public void onSignal
        (Signal<ReactorControl> signal, ReactorControl reactorCtrl) {
      reactorCtrl.stop();
    }
  }
}
```

previous example, signal event handlers may then be attached to the signal event object using its `subscribe` method. Once signal event handlers have been registered they will then receive any signal events which are generated by calling the `signal` method on the signal event object.

One feature of signal event generation demonstrated by Listing 4.5 is the passing of arbitrary data as part of signal event generation. In this example a handle on the reactor's control interface is passed, allowing the signal event handler to shut down the reactor when called. Java generics are used to enforce type safety, ensuring that the data type expected by the callback handler is consistent with the parameter data type passed by the signal event object.

In normal operation, signal handlers which implement the `Signalable` interface should not generate exceptions. There is no well defined way of propagating such error conditions, so any exceptions generated in this manner will just be caught by the reactor and logged as a warning. This may be demonstrated by adding a call to the `signal` method which passes an invalid parameter, as shown in 4.6. The full code for this example is included in the `signal` examples package as `SignalEventExample2`.

Listing 4.6 Generating Signal Events With Invalid Parameter Data

```
...
    // Generate a signal event with invalid parameter.
    signal.signal(null);

    // Generate a signal event and wait for reactor to stop.
    signal.signal(reactorCtrl);
    reactorCtrl.join();
...
```

Listing 4.7 Simple Publisher Interface

```
public interface Publisher {
  public Signal<Integer> getChangeNotificationSignal();
  public String getPublisherData();
}
```

4.4 Subscribing and Unsubscribing

The signal event API provides a mechanism for signalable objects to subscribe to a given signal in order to receive signal event notifications and also to unsubscribe from the signal when such notifications are no longer required. This makes the signal mechanism ideally suited for implementing the *Publisher-Subscriber* pattern [4].

In the Publisher-Subscriber pattern one component acts as a data source (the publisher) and one or more components act as data consumers (the subscribers). The publisher provides a number of data accessor methods and some mechanism for notifying its subscribers of changes to that data. In the case of the Reaction framework the notification mechanism is the signal event. An example interface for a publisher component is shown in Listing 4.7.

This publisher interface is a minimal implementation with a single notification signal and a single data accessor method. The data accessor converts the internal data to a string and the change notification signal is triggered whenever the internal data changes. A subscriber may then subscribe to the change notification signal so that it is informed whenever the data value changes. This is illustrated in the `setup` method of the simple subscriber implementation shown in Listing 4.8.

Once subscribed, signal notifications which are received via the `onSignal` method are used as a trigger for the subscriber to read back the latest publisher data via the publisher's accessor method. This will continue until the subscriber component is unsubscribed from the publisher, as illustrated by the `teardown` method in the example. Once unsubscribed, no further change notifications will be received unless the subscriber re-subscribes to the publisher service. A full implementation of this example is provided in the `signal` examples package as

Listing 4.8 Simple Subscriber Implementation

```
public class Subscriber implements Signalable<Integer> {
  private Publisher publisher;

  // The setup method is used to subscribe to the publisher.
  public void setup(Publisher publisher) {
    this.publisher = publisher;
    publisher.getChangeNotificationSignal().subscribe(this);
    System.out.println("Subscribed : " + publisher.getPublisherData());
  }

  // The teardown method will unsubscribe from the publisher.
  public void teardown() {
    publisher.getChangeNotificationSignal().unsubscribe(this);
    System.out.println("Unsubscribed.");
  }

  // On change notifications, read back the publisher data.
  public void onSignal(Signal<Integer> signal, Integer signalData) {
    System.out.println("Changed : " + publisher.getPublisherData());
  }
}
```

Listing 4.9 Passing Sequence Number in Update Notifications

```
public void onTick(Integer timerData) {
  counter += 1;
  changeNotificationSignal.signal(new Integer(sequence++));
}
```

`SignalSubscribeExample`.

In the example implementation of the publisher service, it is worth commenting on the use of the signal data parameter. The example shows the data parameter being used as a sequence number which is incremented each time the update signal is triggered, as demonstrated in Listing 4.9. Use of this idiom is considered good practise in that it provides a way for subscribers to synchronise state with the publisher if required.

4.5 Finalising Signals

The `signalFinalize` method provided by the `Signal` interface is designed for situations where a service is being shut down and all subscribed signalable objects need to be notified. It issues a standard signal notification to all subscribed signalable objects and then automatically unsubscribes them from the signal. By way of example, this is the mechanism used by the reactor to issue the

Listing 4.10 Returning Restricted Capability Signal References

```
...
    // Get a handle on the change notification signal.
    public Signal<Integer> getChangeNotificationSignal() {
      return changeNotificationSignal.makeRestricted();
    }
...
```

reactor shutdown signal.

One thing to consider when using finalising signals is that there is no predefined way of notifying the signal handlers of the difference between standard and finalising signal callbacks. In some cases – such as the reactor shutdown signal – a signal callback will only be issued once and this can be implicitly interpreted as a finalising signal. In other situations it will be necessary to encode the fact that the signal is a finalising signal as part of the signal data parameter. A common approach is to use the passing of a `null` parameter here as an indication of a finalising signal.

4.6 Restricting Signal Capabilities

In order to make the use of signal event objects more secure, subscribers to a signal event should not normally be able to generate signal event notifications. For example, it would be undesirable if application code could 'fake' a reactor shutdown by calling the `signal` or `signalFinalize` methods on the signal event object obtained via the `getReactorShutdownSignal` method. This is addressed by the presence of the `makeRestricted` method on the signal event object, which returns a *restricted capability* reference. Any attempt to call the `signal` or `signalFinalize` methods on a restricted capability reference will then result in a `RestrictedCapabilityException` being thrown.

Since the restricted and full capability signal references are distinct objects, it is not safe to use the object identity test (`==`) to check equivalence between them. However, the `Object.equals` and `Object.hashCode` methods are fully implemented – so that different references to the same signal event objects are considered to be equivalent for all other comparison tests. Note that the signal event handle passed back to signalable callback handlers as the first parameter in the `onSignal` method will always be a restricted capability reference.

It is generally good practice to restrict the capability of signal event objects which are returned over a publisher interface, such as the one described in Listing 4.7. This ensures that the only entity which can generate notifications for those signal event objects is the publisher itself. Listing 4.10 demonstrates the way in which this may be achieved for the publisher / subscriber example previously presented in Section 4.4.

4.7 Execution Context

Signals may be triggered from within any thread context by calling the `signal` or `signalFinalize` method on a signal event object. However, subsequent execution of the `onSignal` method of any attached signalable objects will always take place within the main reactor thread context. By ensuring that all callbacks take place within the same thread context, synchronisation between callbacks is much simplified. One limitation is that signals may not be subscribed and unsubscribed within the context of an `onSignal` callback handler and attempting to do so will result in a `SignalContextException` being thrown.

Chapter 5

Managing Deferred Events

The *deferred event* is key to the asynchronous programming model since it provides a highly flexible way of managing deferred callbacks. In this case, the event handlers are *deferrable* objects which implement the `Deferrable` interface. This chapter examines deferred events in more detail and introduces the concept of *deferred event objects* which are used to manage callback chains consisting of one or more deferrable objects.

5.1 The Deferred Event API

Deferred events are managed within the Reaction framework using deferred event objects. The reactor core acts as a factory for these objects and the standard reactor interface (`Reactor`) provides the API shown in Listing 5.1.

The `newDeferred` factory method will return an object which implements the `Deferred` interface as shown in Listing 5.2. Such objects are referred to here as being *deferred event objects* since they encapsulate all the state associated with a single deferred callback event. The `Deferred` interface provides methods for attaching deferrable objects (ie, those implementing `Deferrable`) and for triggering callbacks on the deferred event.

Of the methods provided by the deferred event interface, the `callback` and `errback` methods are used for triggering event callbacks and the `addDeferrable` method is used to attach deferrable objects to the deferred event object in order to receive those callbacks. The `setTimeout` and `cancelTimeout` methods are used for managing callback timeouts and the `discard` method provides a clean way of 'ignoring' deferred callbacks. All these API methods will be discussed in more detail in the following sections.

5.2 Simple Deferred Callbacks

The starting point for working with deferred event objects is to consider an example of a long running task which will issue callbacks using the deferred

Listing 5.1 Reactor Deferred Factory Methods

```
package com.zynaptic.reaction;

public interface Reactor {
  ...
  public <T> Deferred<T> newDeferred();
  public <T> Deferred<T> callDeferred(T data);
  public <T> Deferred<T> failDeferred(Exception error);
  ...
}
```

Listing 5.2 Deferred Event Interface Definition

```
package com.zynaptic.reaction;

public interface Deferred<T> {
  public void callback(T data)
      throws RestrictedCapabilityException, DeferredTriggeredException;
  public void errback(Exception error)
      throws RestrictedCapabilityException, DeferredTriggeredException;
  public <U> Deferred<U> addDeferrable
    (Deferrable<T, U> deferrable, boolean terminal)
      throws DeferredTerminationException;
  public <U> Deferred<U> addDeferrable(Deferrable<T, U> deferrable)
      throws DeferredTerminationException;
  public Deferred<T> terminate()
      throws DeferredTerminationException;
  public Deferred<T> setTimeout(int msTimeout)
      throws ReactorNotRunningException;
  public Deferred<T> cancelTimeout();
  public <T> defer()
      throws DeferredContextException, Exception;
  public void discard();
  public Deferred<T> makeRestricted();
}
```

Listing 5.3 The Deferrable Interface Definition

```
package com.zynaptic.reaction;

public interface Deferrable<T, U> {
  public U onCallback(Deferred<T> deferred, T data)
      throws Exception;
  public U onErrback(Deferred<T> deferred, Exception error)
      throws Exception;
}
```

Listing 5.4 Simple Timer-Based Long Running Task

```
public class LongRunningTask implements Timeable<Integer> {
  private Deferred<String> deferred = null;

  // Method used to initiate the long running task.
  public Deferred<String> runTask() {
    reactor.runTimerOneShot(this, 1000, null);
    deferred = reactor.newDeferred();
    return deferred.makeRestricted();
  }

  // Timer event is used to trigger deferred callback.
  public void onTick(Integer data) {
    deferred.callback(new String("Callback Parameter"));
  }
}
```

event mechanism. One such example is given in Listing 5.4, which uses a timer as a proxy for processing or I/O delays.

In this example, the long running task is initiated by making a call to the `runTask` method, which returns a reference to a new deferred event object. The `onTick` timer callback signifies the completion of the task, at which point the deferred callback is triggered – passing the 'result' of the operation as the callback parameter. A Java generic type specifier is used to identify the callback parameter type, which in this case is set to a standard Java `String`.

To accompany the simple timer-based task, a callback handler is required in order to process the generated result. This can take the form of a simple deferrable object which implements the `Deferrable` interface as shown in Listing 5.5.

The `onCallback` callback handler implemented in Listing 5.5 simply prints out the deferred result of the long running task and then stops the reactor in order to terminate the test. The `onErrback` error handler is not used in this context but must be provided in order to fully implement the `Deferrable` interface. Note that Java generic type specifiers are used to parameterise the input and output types of the callback functions. In this case the callbacks expect an input parameter of type `String` and will generate an output which is also of type `String`.

Given these implementations of the simple long running task and simple deferrable callback handler, the long running task may be executed using just the two lines of code shown in Listing 5.6. The full implementation is provided in the `deferred` examples package as `DeferredEventExample1`.

The first of the two lines creates the new long running task and starts it running, accepting the deferred event object which is returned. The second of the two lines attaches the simple callback handler to the deferred event object so that the callback can be executed on completion of the long running task.

Listing 5.5 Simple Deferrable Callback Handler

```
public class SimpleDeferrable implements Deferrable<String, String> {

  // Callbacks report the returned result then shut down the reactor.
  public String onCallback(Deferred<String> deferred, String data) {
    System.out.println("Deferred result : " + data);
    reactorCtrl.stop();
    return null;
  }

  // Errbacks are ignored in this example.
  public String onErrback(Deferred<String> deferred, Exception error) {
    return null;
  }
}
```

Listing 5.6 Executing the Simple Long Running Task

```
...
  Deferred<String> deferred = new LongRunningTask().runTask();
  deferred.addDeferrable(new SimpleDeferrable(), true);
...
```

As part of attaching the callback handler, the `terminal` parameter is set to `true` which indicates that no further callback handlers will be attached to the deferred event object. The use of the `terminal` parameter will be explored in more detail when callback chaining is introduced in Section 5.4.

5.3 Deferred Error Handling

The use of deferred event callbacks may be used to pass back the results of long running tasks on successful completion. However, another mechanism is required for dealing with error and exception conditions which may occur during execution. This is provided through the use of *errbacks*, or *err*ored call*backs*.

In order to demonstrate the use of errbacks, the long running task example given in Listing 5.4 may be extended as shown in Listing 5.7. This shows a typical case where any exceptions generated during the long running task are caught and passed back via the deferred event object's `errback` method. As an alternative to using `try/catch`, new exceptions may also be explicitly created as the errback parameter.

Errors are processed by deferrable callback handler using the `onErrback` method. In place of the generically typed data parameter which is passed on callbacks, the `onErrback` method accepts a parameter of type `Exception`, which is the exception object passed by the deferred `errback` call. An example of a

Listing 5.7 Simple Long Running Task With Errbacks

```
public class LongRunningTask implements Timeable<Integer> {
  private boolean generateError = false;
  private Deferred<String> deferred = null;

  // Method used to initiate the long running task.
  public Deferred<String> runTask(boolean generateError) {
    this.generateError = generateError;
    this.deferred = reactor.newDeferred();
    reactor.runTimerOneShot(this, 1000, null);
    return deferred.makeRestricted();
  }

  // Timer event is used to trigger deferred callback.
  public void onTick(Integer data) {
    try {
      if (generateError)
        throw new Exception("Errback Parameter");
      deferred.callback(new String("Callback Parameter"));
    } catch (Exception exc) {
      deferred.errback(exc);
    }
  }
}
```

simple deferrable callback handler with support for error processing is shown in Listing 5.8. The full code for this example is included in the `deferred` examples package as `DeferredEventExample2`.

5.4 Deferred Callback Chaining

What sets deferred event objects apart from conventional callback implementations is their ability to support callback chaining. It is possible to attach multiple deferrable callback handlers to a single deferred event object such that when a callback is made the chained callback handlers are each invoked in turn. For the purposes of demonstrating this feature, the simple deferrable callback handler of Listing 5.5 is extended as shown in Listing 5.9.

Of particular interest in this version of the deferrable callback handler is that the `onCallback` method processes the input data and returns the modified version. In this case, it is simply a case of wrapping the string parameter with function-style parentheses. It is then possible to attach multiple deferrable callback handlers to the deferred event object as shown in Listing 5.10.

The example shown in Listing 5.10 also illustrates the use of the `terminal` parameter of the `addDeferrable` method. When adding the final deferrable in a chain the `terminal` flag must be set to `true`. This terminates the callback chain,

Listing 5.8 Simple Deferrable Callback Handler With Errback

```
public class SimpleDeferrable implements Deferrable<String, String> {

  // Callbacks report the returned result then shut down the reactor.
  public String onCallback(Deferred<String> deferred, String data) {
    System.out.println("Deferred result : " + data);
    reactorCtrl.stop();
    return null;
  }

  // Errbacks are reported to the console in this example.
  public String onErrback(Deferred<String> deferred, Exception error) {
    System.out.println("Deferred error : " + error.toString());
    return null;
  }
}
```

Listing 5.9 Simple Chained Callback Handler

```
public class SimpleDeferrable implements Deferrable<String, String> {
  String name;

  // Constructor assigns name of deferrable.
  public SimpleDeferrable(String name) {
    this.name = name;
  }

  // Callbacks report the returned result and pass on processed data.
  public String onCallback(Deferred<String> deferred, String data) {
    System.out.println(name + " input = " + data);
    return new String(name + " (" + data + ")");
  }

  // Errbacks are ignored in this example.
  public String onErrback(Deferred<String> deferred, Exception error) {
    return null;
  }
}
```

Listing 5.10 Creating Simple Deferrable Callback Chain

```
...
  // Start up a long running task.
  Deferred<String> deferred = new LongRunningTask().runTask();

  // Attach callbacks to the long running task.
  deferred.addDeferrable(new SimpleDeferrable("C1"), false);
  deferred.addDeferrable(new SimpleDeferrable("C2"), false);
  deferred.addDeferrable(new SimpleDeferrable("C3"), false);
  deferred.addDeferrable(new SimpleDeferrable("C4"), true);
...
```

Listing 5.11 Output of Simple Deferrable Callback Chain

```
INFO    : Started reactor.
C1 input = Callback Parameter
C2 input = C1 (Callback Parameter)
C3 input = C2 (C1 (Callback Parameter))
C4 input = C3 (C2 (C1 (Callback Parameter)))
INFO    : Halting reactor.
INFO    : Reactor stopped.
```

making it ready for execution. For this reason, the final deferrable object in a callback chain is generally referred to as the *terminal deferrable*. Once a callback chain has been terminated, any attempt to add further deferrable callback objects will generate a runtime exception of type `DeferredTerminatedException`.

The full implementation of the chained callback example is present in the `deferred` examples package as `DeferredChainExample1`. Running the example code results in the console output shown in Listing 5.11. From this it is possible to see that processing the callback chain using the deferred event object is equivalent to making a set of nested function calls with the first callback method acting as the 'innermost' function and the final callback method acting as the 'outermost' function.

5.5 Deferred Error Callback Chaining

The process of handling exception conditions within a callback chain is similar to conventional callback chaining. Error conditions are passed up the callback chain by throwing exceptions from within the deferrable callback handler's `onErrback` method, as shown in Listing 5.12.

On calling the `onErrback` method on a given deferrable callback handler, the deferred event object passes the exception condition which was thrown by the previous stage in the callback chain. In the example implementation the exception is simply re-thrown at each stage. Generally speaking, the error con-

Listing 5.12 Simple Chained Errback Handler

```
public class SimpleDeferrable implements Deferrable<String, String> {
  String name;

  // Constructor assigns name of deferrable.
  public SimpleDeferrable(String name) {
    this.name = name;
  }

  // Callbacks are ignored in this example.
  public String onCallback(Deferred<String> deferred, String data) {
    return null;
  }

  // Errbacks are propagated in this example.
  public String onErrback(Deferred<String> deferred, Exception error)
      throws Exception {
    System.out.println(name + " input = " + error);
    throw (error);
  }
}
```

Listing 5.13 Output of Simple Deferrable Errback Chain

```
INFO    : Started reactor.
C1 input = java.lang.Exception: Errback Exception
C2 input = java.lang.Exception: Errback Exception
C3 input = java.lang.Exception: Errback Exception
C4 input = java.lang.Exception: Errback Exception
WARNING : Unhandled exception in deferred closure.
THROWN  : java.lang.Exception: Errback Exception
  at com.zynaptic.reaction.examples.DeferredChainExample2...
  at com.zynaptic.reaction.core.Reactor.processExpiredTimers...
  at com.zynaptic.reaction.core.Reactor.run(Reactor.java:478)
  at java.lang.Thread.run(Thread.java:636)
INFO    : Halting reactor.
INFO    : Reactor stopped.
```

dition should always be dealt with by the terminal deferrable, since unhandled error conditions at this position in the callback chain will effectively be 'lost'. Listing 5.13 demonstrates the way in which such unhandled error conditions are reported to the reactor log as warning messages. The full code for this example is included in the `deferred` examples package as `DeferredChainExample2`.

5.6 Mixing Callbacks and Errbacks

The preceding examples of callback chain operation dealt exclusively with either callback or errback propagation. In practise it is possible that an `onCallback` method within the callback chain may throw an Exception or an `onErrback` method within the callback chain may successfully resolve an error condition, allowing normal processing to resume.

The processing of return values and exception conditions for `onCallback` and `onErrback` methods is actually identical. If an `onCallback` or `onErrback` method returns normally, the next deferrable callback handler in the chain will always have its `onCallback` method executed – passing in the data object returned by the previous stage of the callback chain. Similarly, if an `onCallback` or `onErrback` method throws an exception, this will be caught and passed to the `onErrback` method of the next deferrable callback handler in the chain.

Listing 5.13 demonstrates the way in which error conditions which are not dealt with by the terminal deferrable will appear in the reactor logs. In order to prevent these warnings, neither the `onCallback` or `onErrback` method of a terminal deferrable should throw an exception – these will usually return a `null` object reference since no further processing is possible.

5.7 Callback Parameter Type Conversion

The deferrable callback examples previously presented all make the assumption that while callbacks may modify the callback data, the callback parameter type remains fixed. This has helped to simplify the initial examples, but it does not reflect the more realistic situation where callback handlers will typically transform callback parameters from one type to another.

Callback parameter type conversion is expressed in the deferrable interface definition given in Listing 5.3 by the two Java generics `<T>` and `<U>`. The generic type `<T>` identifies the type of the callback parameter which is passed into the deferrable callback handler, while the generic type `<U>` identifies the data type which is returned by the callback handler on completion. As an example, Listing 5.14 shows a deferrable callback handler which converts an input of type `Integer` to an output of type `String`.

The deferred event interface presented in Listing 5.2 is also parameterised by the generic type `<T>`, which in this case specifies the data type which will be output by the previous stage of the callback chain. Therefore when appending deferrable callback objects to a deferred event object, the generic type `<T>` of the deferrable callback input must match the generic type `<T>` specified for the deferred event object. This means that in the case of the example shown in Listing 5.14, the deferrable callback object may only be added to a deferred event object whose complete type specification is `Deferred<Integer>`.

When a new deferrable callback is added to the callback chain of a deferred event object, it modifies the generic type of the deferred event object to match the output type of the deferrable callback. This is carried out automatically

Listing 5.14 Deferrable Callback Handler With Type Conversion

```
public class TypeConversionDeferrable
  implements Deferrable<Integer, String> {

  // Callbacks convert the integer input to a string output.
  public String onCallback(Deferred<Integer> deferred, Integer data) {
    System.out.println("Integer parameter => " + data);
    return new String("Value = " + data);
  }

  // Errbacks are unused in this example.
  public String onErrback(Deferred<Integer> deferred, Exception error) {
    return null;
  }
}
```

Listing 5.15 Adding Deferrable Callbacks With Type Conversion

```
...
  // Start up a long running task.
  Deferred<Integer> deferredInteger = new LongRunningTask().runTask();

  // Attach callback to convert from integer to string.
  Deferred<String> deferredString = deferredInteger.addDeferrable
    (new TypeConversionDeferrable(), false);

  // Attach callback to report string result.
  deferredString.addDeferrable(new TerminalDeferrable(), true);
...
```

by the `addDeferrable` method, which returns a new reference to the deferred event object with suitable type casting applied.

Listing 5.15 shows the way in which type conversion may be applied to a deferred event object. The initial reference to the deferred event object (`deferredInteger`) is returned on starting a new long running task that will pass up an integer result on completion. When the deferrable callback object given in Listing 5.14 is added to the callback chain, a new reference to the deferred event object (`deferredString`) is automatically created with the correct type specification. This new reference may then be used to append another deferrable callback object, which in this case will accept a Java string object as its input. The full implementation of this example is present in the `deferred` examples package as `DeferredChainExample3`.

5.8 Restricting Deferred Capabilities

In order to make the use of deferred event object more secure, only the original source of the deferred event object should normally be able to issue callbacks. This may be enforced by calling the `makeRestricted` method on the deferred event object, which returns a *restricted capability* reference. Any attempt to call the `callback` or `errback` methods on a restricted capability reference will then result in a `RestrictedCapabilityException` being thrown. This is demonstrated by the simple long running task given in Listing 5.4.

Since the restricted and full capability deferred references are distinct objects, it is not safe to use the object identity test (`==`) to check equivalence between them. However, the `Object.equals` and `Object.hashCode` methods are fully implemented – so that different references to the same deferred event objects are considered to be equivalent for all other comparison tests. Note that the deferred event handle passed back to deferrable callback handlers as the first parameter in the `onCallback` and `onErrback` methods will always be a restricted capability reference.

5.9 Execution Context

The processing of deferrable callback chains is always carried out in the context of the reactor thread. Callbacks will only be processed once a terminal deferrable has been added to the callback chain *and* a call to either the `callback` *or* the `errback` method is made. Note that a call to the `callback` or `errback` method must be made exactly once per deferred event object, and any additional attempts to call these methods will result in a runtime exception of type `DeferredTriggeredException` being thrown.

There is no restriction on the ordering of calls to the `callback`, `errback` and `addDeferrable` methods – so deferrable callback handlers can actually be added to the callback chain *after* a call to the `callback` or `errback` method has already been made, provided the callback chain has not already been terminated. Calls to the `callback`, `errback` and `addDeferrable` methods may be made from any thread context since deferred event objects are thread safe.

5.10 Deferred Callback Timeouts

There are certain situations – particularly when performing external I/O – that a long running task may fail to complete. When such a situation occurs, the `callback` or `errback` methods on the associated deferred event object will never be called and the callback chain would not then be executed.

To provide a clean way of dealing with long running tasks which fail to complete it is possible to make use of deferred event timeouts. These are supported by the `setTimeout` and `cancelTimeout` methods on the `Deferred` interface. An example of setting a deferred callback timeout is shown in Listing 5.16. The

Listing 5.16 Setting a Deferred Callback Timeout

```
...
  // Attach a deferrable callback handler and set a timeout.
  deferred.addDeferrable(new SimpleDeferrable(), true);
  deferred.setTimeout(500);
...
```

full version of this example is included in the `deferred` examples package as `DeferredTimeoutExample`.

In the event that the timeout expired before a call to the `callback` or `errback` method is made, an exception of type `DeferredTimedOutException` will be passed back via the `onErrback` method of the first deferrable callback handler in the chain. The callback handlers may then process the timeout condition as if it were a conventional error condition. A subtle consequence of triggering a timeout condition in this manner is that a subsequent call to the `callback` or `errback` methods will be silently discarded, since the callbacks have already been triggered.

An alternative mechanism for setting timeouts on threadable tasks is described in Section 6.6. In addition to the behaviour described here, the alternative method will also automatically cancel the underlying thread execution.

5.11 Waiting For Deferred Events

In certain circumstances it may be desirable to combine asynchronous calls and more conventional blocking I/O calls within the same function. Typically, such sections of code will be executed in the context of a threadable task as described in Section 6.2. However, this type of interworking can also be achieved for arbitrary threaded code provided that the blocking calls are not executed in the context of the main reactor thread.

In order to achieve this, each deferred event object implements the `defer` method on the `Deferred` interface. On calling this method, the callback chain will be automatically terminated with a synchronizing callback handler and the calling thread will be blocked until a deferred callback has been received. The callback data is then passed back as the return value for the `defer` method call as the calling thread is resumed. In the event that an errback condition is received, the exception which was passed as the errback parameter will be re-thrown in the context of the calling thread.

An example of mixing conventional blocking I/O calls with asynchronous APIs is shown in Listing 5.17. In this example, raw data is being read in 16 byte chunks from a standard Java input stream using blocking I/O. Within the loop, an asynchronous API is being used to write out the data. The asynchronous API call returns a deferred event object that on completion of the write operation will pass back a status value as the callback parameter.

Listing 5.17 Implementing a Blocking Wait For Deferred Callbacks

```
...
  // Perform repeated blocking reads on an input stream.
  byte[] byteBuffer = new byte[16];
  int byteCount;
  while ((byteCount = inputStream.read(byteBuffer)) >= 0) {

    // Write out data using an asynchronous API.
    Deferred<Status> deferred = asyncOut.write(byteBuffer, byteCount);

    // Wait for asynchronous completion and check callback data.
    try {
      Status status = deferred.defer();
      if (status != Status.OK) {
        // Handle unexpected status.
      }
    }
    catch (Exception error) {
      // Handle errback exceptions.
    }
  }

...
```

In order to process the write status value within the loop, a call to the `defer` method is made, which causes execution of the loop to block until the status value has been returned from the asynchronous write operation. It is also possible that an exception could be passed back via the deferred errback handler, in which case it will be re-thrown and must be handled in the conventional manner. Note that the `defer` method declaration specifies that any exception derived from the base `Exception` class may be thrown, since exact details of the errback conditions which may be generated will be specific to a given asynchronous API call.

5.12 Pre-Triggered Deferred Factory Methods

In addition to the conventional `newDeferred` factory method for deferred event objects, the reactor provides two methods for generating pre-triggered deferred event objects. These are useful in situations where an asynchronous API call can return an immediately available value as the callback parameter or an immediately generated exception as the errback parameter. The `callDeferred` convenience method takes the data value which is to be passed as the callback parameter and creates a new deferred event object where the `onCallback` method has already been invoked using that data value. Similarly, the `failDeferred` convenience method takes the exception object which is to be passed as the er-

Listing 5.18 Using Pre-Triggered Deferred Event Objects

```
...
  public Deferred<String> getName() {
    if (cachedName != null) {
      return reactor.callDeferred(cachedName);
    }
    else if (cachedError != null) {
      return reactor.failDeferred(cachedError);
    }
    else {
      return asyncGetName().addDeferrable(cacheUpdateHandler);
    }
  }
...
```

rback parameter and creates a new deferred event object where the `onErrback` method has already been invoked.

An example of using pre-triggered deferred event objects is shown in Listing 5.18. In this case, the desired return value may already have been accessed and locally cached. If present, the cached value can be returned immediately using the `callDeferred` method. Alternatively, a cached error condition can be returned immediately using the `failDeferred` method. If no cached value is present, a conventional asynchronous call can be made, which as well as returning the data value to the caller can also be configured to update the cached values when the callback chain is executed.

5.13 Discarding Deferred Event Objects

A potential issue arises where deferred event objects are returned from method calls that are part of a higher level API. There may be situations where the caller just wants to initiate a transaction and is not concerned with processing the returned callbacks. If the returned deferred event object is allowed to go out of scope without being correctly terminated, a warning will be appended to the reactor logs on object finalisation.

To prevent the generation of these warning messages in the reactor logs, the deferred event object may be explicitly discarded by calling the `discard` method on the `Deferred` interface. This has the effect of terminating the callback chain with an implicit 'empty' terminal deferrable.

5.14 Fluent-Style Deferred Programming

The API for deferred event objects as shown in Listing 5.2 also provides support for fluent-style programming. Since most of the API calls return an appropriately typed reference to the deferred event object itself, it is possible to chain

Listing 5.19 Building a Fluent-Style Deferred Callback Chain

```
...
  // Using fluent-style programming to build a callback chain.
  terminatedDeferred = reactor.newDeferred()
    .addDeferrable(deferrableA)
    .addDeferrable(deferrableB)
    .addDeferrable(deferrableC)
    .setTimeout(500)
    .terminate();
...
```

together multiple method calls within a single statement. An example of this approach is shown in Listing 5.19.

In order to make code written in this manner more elegant, a reduced-form version of the `addDeferrable` method is provided which omits the `terminal` parameter. All deferrable callbacks which are added via this method will then be appended to the callback chain without terminating it. The additional `terminate` method is provided for explicitly terminating callback chains which have been assembled in this manner

Since the `setTimeout`, `cancelTimeout` and `makeRestricted` API calls return an appropriately typed reference to the deferred event object, these may also be used in a fluent-style manner. As well as building the callback chain, the supplied example shows the additional assignment of a 500 millisecond timeout.

Chapter 6

Running Threadable Tasks

One of the main goals of the Reaction framework is to provide an easy way to manage long running tasks within an event driven framework. This allows the clean integration of activities which would normally block the flow of execution – such as synchronous RPC calls or long-running processing loops. It also provides an easy and intuitive way of managing concurrent execution. Central to these goals is the implementation of *threadable task* objects, which will be described in detail in this chapter.

6.1 The Threadable Task API

The methods used for managing threadable tasks are provided by the standard reactor interface (`Reactor`), as defined in the Reaction API package. The full set of methods used for threadable task management are shown in Listing 6.1.

The reactor interface methods provide support for running a threadable task in a new thread (`runThread`) and for cancelling a threadable task which is already running (`cancelThread`). In each case a single threadable task object is passed as the primary parameter, while the `runThread` method also passes in a generically typed input data parameter. Threadable task objects are defined as those which implement the `Threadable` interface as shown in Listing 6.2.

The only method required by the `Threadable` interface is the `run` method. This takes a data object of generic type `<T>` as its parameter, which will be the same data object originally passed to the reactor's `runThread` method. The `run` method must eventually return a data object of generic type `<U>` or throw an exception, but there is no limitation on the amount of time which it may take to run.

6.2 Running a Simple Threadable Task

Implementing a long running threadable task is just an exercise in implementing the `run` method of the `Threadable` interface. Unlike code which runs in

Listing 6.1 Reactor Thread Control Interface

```
package com.zynaptic.reaction;

public interface Reactor {
...
  public <T, U> Deferred<U> runThread
    (Threadable<T, U> threadable, T data)
      throws ReactorNotRunningException, ThreadableRunningException;
  public <T, U> Deferred<U> runThread
    (Threadable<T, U> threadable, T data, int msTimeout)
      throws ReactorNotRunningException, ThreadableRunningException;
  public void cancelThread(Threadable<?, ?> threadable);
...
}
```

Listing 6.2 The Threadable Interface Definition

```
package com.zynaptic.reaction;

public interface Threadable<T, U> {
  public U run(T data) throws Exception;
}
```

the context of the main reactor thread – such as timer, signal and deferrable callbacks – threadable tasks are allowed to block. In the example shown in Listing 6.3 there is a blocking call to the `Thread.sleep` method which halts the execution of the threadable task for 1 second.

Assuming that the long running task is not interrupted while sleeping, it will return a generically typed data object as specified by the interface definition for the `run` method. In order to run the threadable task, the `runThread` method on the `Reactor` interface is invoked as shown in Listing 6.4.

The `runThread` method returns a deferred event object which will have its callbacks executed when the threadable task's `run` method returns. The callback parameter passed up the callback chain will be the generically typed data object which is returned by the threadable task's `run` method. In order to process the callbacks, the simple deferrable callback handler previously presented in Section 5.2 may be used. The full implementation of this example may be found in the `thread` examples package as `ThreadableExample1`.

6.3 Exception Handling in Threadable Tasks

One advantage of using threadable tasks is that unlike 'raw' threads there is a well defined way of handling exceptions which occur during execution. These are automatically caught by the Reaction framework and converted to deferred

Listing 6.3 A Simple Threadable Long Running Task

```
public class LongRunningTask implements Threadable<String, String> {
  public String run(String data) throws InterruptedException {
    System.out.println("Threadable input : " + data);
    Thread.sleep(1000);
    return new String("Threadable Result");
  }
}
```

Listing 6.4 Running a Threadable Long Running Task

```
...
  Deferred<String> deferred = reactor.runThread
    (new LongRunningTask(), new String("Input Data"));
  deferred.addDeferrable(new SimpleDeferrable(), true);
...
```

errback calls. The error condition may then be handled in the same manner as conventional deferred errbacks. An example of a long running task which generates an exception is shown in Listing 6.5.

In this example an exception is explicitly thrown by the threadable `run` method, and the generated exception object will be passed back via the deferred errback chain. The full implementation of this example is provided in the `thread` examples package as `ThreadableExample2`. This includes a deferrable callback handler which prints out the details of the generated exception, as shown in Listing 6.6.

It is worth noting from the debug log messages that additional worker threads are automatically created and destroyed in order to execute the threadable task code. These are standard Java threads which are managed within the reactor core as a dynamic thread pool. By reusing threads from the thread pool, the overhead of creating and destroying thread objects for each execution of a threadable task is much reduced.

6.4 Stateful Threadable Task Objects

A useful feature of threadable task objects is that it is possible to create stateful threadable tasks which may be submitted for execution multiple times. The one constraint is that a given threadable task cannot be resubmitted if the task is already running. Any attempt to resubmit a threadable task that is already running will cause the reactor's `runThread` method to throw a runtime exception of type `ThreadableRunningException`. An example implementation of a stateful threadable task is shown in Listing 6.7.

In the example the task state is represented by the `count` member variable. This will be incremented each time the threadable task is submitted for exe-

Listing 6.5 Throwing Exception From a Threadable Long Running Task

```
public class LongRunningTask implements Threadable<String, String> {
  public String run(String data) throws InterruptedException {
    System.out.println("Threadable input : " + data);
    Thread.sleep(1000);
    throw new InterruptedException("Task failed");
  }
}
```

Listing 6.6 Console Output of Threadable Exception Handling

```
INFO    : Started reactor.
DEBUG   : Started Thread[Worker1,5,Reaction]
Threadable input : Input Data
Deferred error   : java.lang.InterruptedException: Task failed
INFO    : Halting reactor.
DEBUG   : Killing Thread[Worker1,5,Reaction]
INFO    : Reactor stopped.
```

cution, until an exception is thrown on the fifth execution of the `run` method. The corresponding deferrable callback handler used in the example is shown in Listing 6.8. The `onCallback` method automatically resubmits the threadable task object for execution and the `onErrback` method terminates the test on completion. The full implementation of this example is included in the `thread` examples package as `ThreadableResubmitExample`.

6.5 Cancelling a Running Threadable Task

The process of cancelling a running threadable task makes use of the underlying Java threads API for interrupting blocked threads. When the reactor's `cancelThread` method is called for a given threadable task object, the standard `interrupt` method is called on the underlying Java thread. This means that the blocked thread will be interrupted, throwing an exception of type `InterruptedException`. This exception will then be passed back via the threadable's deferred errback chain using the standard deferred error handling model.

The cancellation of threadable tasks may be demonstrated by making a small modification to the simple threadable test example previously used. The modified thread startup and cancellation code is shown in Listing 6.9. This results in the console output given in Listing 6.10, which clearly demonstrates the generation of the `InterruptedException` errback. The full version of this example is included in the `thread` examples package as `ThreadableCancelExample`.

Listing 6.7 A Stateful Threadable Long Running Task

```
public class LongRunningTask implements Threadable<String, String> {
  private int count = 0;

  public String run(String data) throws InterruptedException {
    System.out.println("Threadable input : " + data);
    Thread.sleep(1000);
    if (++count == 5)
      throw new InterruptedException("Test over");
    return new String(data + " " + count);
  }
}
```

Listing 6.8 Resubmitting Threadable From a Deferrable Callback

```
public class SimpleDeferrable implements Deferrable<String, String> {

  // Callbacks resubmit the threadable.
  public String onCallback(Deferred<String> deferred, String data) {
    reactor.runThread(longRunningTask, data).addDeferrable(this, true);
    return null;
  }

  // Errbacks are used to stop the reactor.
  public String onErrback(Deferred<String> deferred, Exception error) {
    System.out.println("Deferred error   : " + error);
    reactorCtrl.stop();
    return null;
  }
}
```

Listing 6.9 Cancellation of Threadable Long Running Task

```
...
  // Start up a long running task.
  Threadable<String, String> longRunningTask = new LongRunningTask();
  Deferred<String> deferred = reactor.runThread
    (longRunningTask, new String("Input Data"));

  // Attach callbacks to the long running task.
  deferred.addDeferrable(new SimpleDeferrable(), true);

  // Cancel the task part way through.
  Thread.sleep(500);
  reactor.cancelThread(longRunningTask);
...
```

Listing 6.10 Console Output of Threadable Cancellation

```
INFO    : Started reactor.
DEBUG   : Started Thread[Worker1,5,Reaction]
Threadable input : Input Data
DEBUG   : Cancelling Thread[Worker1,5,Reaction]
Deferred error   : java.lang.InterruptedException: sleep interrupted
INFO    : Halting reactor.
DEBUG   : Killing Thread[Worker1,5,Reaction]
INFO    : Reactor stopped.
```

6.6 Threadable Task Timeouts

The type of behaviour demonstrated by the example given in Listing 6.9 can be useful for managing threadable tasks which may block during execution. This essentially provides a timeout on the threadable execution - cancelling it if an underlying call blocks for an unexpectedly long time.

Support for implementing this type of timeout behaviour is included in the API via an alternate form of the `runThreadable` method which includes an extra timeout parameter. This method variant will execute the threadable task as previously described, but it also sets a timeout on the deferred callback which will automatically cancel the thread when the execution time exceeds the specified timeout period.

The initial timeout period must be specified when making the initial call to the `runThreadable` method. However, the timeout behaviour may subsequently be modified by calling the `setTimeout` and `cancelTimeout` methods on the returned deferred event object, as previously described in Section 5.10.

6.7 Execution Context

Threadable tasks may be scheduled for execution using the reactor's `runThread` method from any thread context. Similarly, the reactor's `cancelThread` method may be used to cancel a running threadable task from any context. All deferred callback processing associated with the completion of a threadable task is carried out within the context of the main reactor thread.

Threadable tasks run within independent worker threads, so the usual synchronisation issues associated with sharing data between threads still need to be considered. Thread synchronisation can be much simplified by using immutable objects for the threadable input data and return value. An alternative model is to effectively transfer ownership of the threadable input data to the threadable object by discarding references to it in the initiating thread after it has been passed as the data parameter to the `runThread` method.

Each time a threadable task is scheduled for execution by calling the reactor's `runThread` method it will be arbitrarily assigned to an existing Java thread from the reactor's thread pool. If a stateful threadable task is resubmitted for

execution, there is no guarantee that successive runs will use the same underlying Java thread object. This means that threadable tasks must not attempt to directly manipulate the underlying Java thread state.

6.8 Thread Prioritisation Levels

The Reaction framework is designed so that the main reactor thread provides low latency processing of the various different event types. For this reason the main thread is set up using the maximum thread priority level (`Thread.MAX_PRIORITY`). The worker threads which are responsible for executing threadable objects should run at a lower priority so that they do not interfere with the low latency operation of the main thread. For this reason, the worker threads are assigned the normal thread priority level (`Thread.NORM_PRIORITY`).

Thread prioritisation works correctly on most JVM implementations, but unfortunately there are some JVM's in common use which do not respect the thread priority settings. This is a particular problem when using Java under Linux. Generally speaking, JVM's which do not support thread prioritisation will still use some form of timeslicing scheduler which obviates the need to explicitly yield from executing threadable tasks. However, the timeliness of reactor event processing will degrade as the number of computationally intensive threads increases.

As a general rule, applications which only make use of threadable objects to manage blocking I/O will work correctly regardless of thread priority support. Applications which also make use of threadable objects for running computationally intensive tasks should be restricted to platforms which correctly implement the prioritised threading model in order to guarantee timely handling of reactor events.

Chapter 7

Advanced Deferred Events

The standard deferred event model provides a useful mechanism for propagating a single deferred event to multiple deferrable objects within a single callback chain. This chapter introduces two additional components which support more complex callback management. These provide additional functionality by allowing multiple callback chains to be merged into a single callback chain and a single callback chain to be split into multiple parallel callback chains.

7.1 Advanced Deferred Event API

Advanced deferred callback management is provided through the use of deferred event *concentrators* and deferred event *splitters*. These implement the `DeferredConcentrator` and `DeferredSplitter` interfaces respectively. The reactor core acts as a factory for components implementing these interfaces and the standard reactor interface provides the factory functions shown in Listing 7.1.

The `newDeferredConcentrator` factory method will return an object which implements the `DeferredConcentrator` interface as shown in Listing 7.2. Similarly, the `newDeferredSplitter` factory method will return an object which implements the `DeferredSplitter` interface as shown in Listing 7.3.

In the case of both the concentrator and splitter interfaces there is an `addInputDeferred` method for attaching deferred event inputs as well as a `getOutputDeferred` method for obtaining a handle on an output deferred event object. However, there are obvious functional differences between the implementation of these calls for concentrator and splitter components.

7.2 Using Deferred Event Concentrators

The deferred event concentrator is used to collect the results of multiple deferred event callbacks and forward them to a single deferred event callback chain. A typical example of setting up a deferred event concentrator to do this is shown in

Listing 7.1 Reactor Advanced Deferred Event API Factory Methods

```
package com.zynaptic.reaction;

public interface Reactor {
...
  public <T> DeferredConcentrator<T> newDeferredConcentrator();
  public <T> DeferredSplitter<T> newDeferredSplitter();
...
}
```

Listing 7.2 Interface Definition for Deferred Event Concentrator

```
package com.zynaptic.reaction;

public interface DeferredConcentrator<T> {
  public void addInputDeferred(Deferred<T> deferred)
      throws DeferredTerminationException;
  public Deferred<List<T>> getOutputDeferred();
}
```

Listing 7.4. The full implementation of this example is present in the `deferred` examples package as `DeferredConcentratorExample`.

In the example, a deferred event concentrator is created using the reactor's factory method. A number of threadables are then created which are used to execute long running tasks – in the case of the example these are prime factorisations of arbitrary integer values. The deferred event objects returned on starting the threadable tasks are then added as inputs to the deferred event concentrator, which automatically terminates their callback chains.

The final stage is to attach the callback handlers to the output of the deferred event concentrator. A handle on the output deferred event object is obtained by calling the `getOutputDeferred` method on the deferred event concentrator. When this method is first called on a given concentrator object, the concentrator is 'locked' so that further inputs cannot be added. Once locked, any attempts to call `addInputDeferred` on the concentrator object will result in an exception of type `DeferredTerminationException`. Callback handlers can be attached to the output deferred event object in the normal fashion, with the output callback chain being ready for triggering once it has been correctly terminated.

In the event that all input deferred events complete successfully (ie, issue callbacks rather than errbacks), the callback parameter data for each deferred event is added to an ordered list. Once all deferred event inputs have issued their callbacks, this ordered list is then passed as the parameter to the output callback chain. In the example, this list contains string representations of the calculated prime factors. Listing 7.5 shows a callback handler which may be used to print out these results to the console.

Listing 7.3 Interface Definition for Deferred Event Splitter

```
package com.zynaptic.reaction;

public interface DeferredSplitter<T> {
  public void addInputDeferred(Deferred<T> deferred)
      throws DeferredTerminationException;
  public Deferred<T> getOutputDeferred();
}
```

Listing 7.4 Example of Using a Deferred Event Concentrator

```
...
  // Create a deferred concentrator object to collect all the results.
  DeferredConcentrator<String> resultsConcentrator =
    reactor.newDeferredConcentrator();

  // Initiate the long running tasks and attach the deferred results
  // to the deferred result concentrator.
  for (int i = 0; i < nums.length; i++) {
    Deferred<String> deferredResult = reactor.runThread
      (new PrimeFactorCalculator(), new Integer(nums[i]));
    resultsConcentrator.addInputDeferred(deferredResult);
  }

  // Once all deferred results have been added to the concentrator,
  // attach the deferred output handler.
  resultsConcentrator.getOutputDeferred().addDeferrable
    (new ResultsProcessor(), true);
...
```

Listing 7.5 Example of Callback Handling for Deferred Event Concentrator

```
...
  public String onCallback
      (Deferred<List<String>> deferred, List<String> data) {
    System.out.println("Prime factorisation results :");
    Iterator<String> listIter = data.iterator();
    for (int i = 0; i < data.size(); i++) {
      System.out.println("  " + nums[i] + " => " + listIter.next());
    }
    reactorCtrl.stop();
    return null;
  }
...
```

A significant feature of the results list passed by the concentrator's output callback is that its ordering is consistent with the order in which input deferred events were added to the concentrator. This means that the result passed back by the first deferred input event to be added to the concentrator will be placed at the start of the list, with the results from other deferred input events being added in order. In the example, this means that the printed result can combine ordered data associated with the input deferred events (the `nums` array) with the ordered data placed in the results list by the concentrator (the `data` list).

When one of the deferred event inputs to a concentrator object generates an errback condition, this will be immediately propagated to the concentrator's output. Since the output deferred event object can only handle a single errback condition, only the first errback to occur will be propagated in this manner. All successful callback inputs and subsequent errback inputs will be silently discarded.

7.3 Using Deferred Event Splitters

The deferred event splitter is used to 'fork' a single callback chain so that callbacks may be processed in parallel by multiple output callback chains. This is typically used when there are multiple client components which need to wait on the same underlying transaction. An example of this is shown in Listing 7.6. The full version of this example is present in the `deferred` examples package as `DeferredSplitterExample`.

In this example a deferred event splitter object is created using the reactor's factory method. A number of deferrable callback handlers are then created, which are attached to the outputs of the deferred event splitter. Output deferred event objects are created dynamically on calling the `getOutputDeferred` method, so an arbitrary number of output callback chains can be attached.

The example then initiates a long running task and attaches the resulting deferred event object to the input of the deferred event splitter using the `addInputDeferred` method. Finally, more output handlers can be attached if required. The deferred splitter component differs from the deferred concentrator in that the only constraint is that the `addInputDeferred` method must be called exactly once. Extra output callback chains can be attached even after the input callback chain has triggered, since the output of the input callback chain is cached for the lifetime of the deferred splitter object.

It is important to note that when the callback on the input deferred event object is triggered, the data parameter which is passed by that callback will be passed in turn to each of the registered output callback chains. Therefore callback handlers attached to the outputs of a deferred splitter object must not make any changes to the parameter data object which is passed to them. In order to ensure that this is the case, it is recommended that all parameter data objects passed to the input of a deferred event splitter are made immutable.

The handling of errback conditions by the deferred event splitter is essentially the same as for callback handling. The exception object which is passed as the

Listing 7.6 Example of Using a Deferred Event Splitter

```
...
  // Create a deferred splitter object to fork the results.
  DeferredSplitter<String> resultsSplitter =
    reactor.newDeferredSplitter();

  // Attach some callback handlers to the deferred splitter outputs.
  for (int i = 0; i < CALLBACK_MAX / 2; i++) {
    resultsSplitter.getOutputDeferred().addDeferrable
      (new ResultsProcessor(i), true);
  }

  // Initiate a long running task and attach it to the deferred
  // splitter input.
  int num = 1 + new Random().nextInt(100000000);
  Deferred<String> deferredResult = reactor.runThread
    (new PrimeFactorCalculator(), new Integer(num));
  resultsSplitter.addInputDeferred(deferredResult);

  // Attach some more callback handlers if required.
  for (int i = CALLBACK_MAX / 2; i < CALLBACK_MAX; i++) {
    resultsSplitter.getOutputDeferred().addDeferrable
      (new ResultsProcessor(i), true);
  }
...
```

input errback parameter will be forwarded in turn to each of the registered output errback chains. Again, output errback handlers should not modify the underlying exception object during processing.

Chapter 8

The Reaction OSGi Service

The Reaction framework has been designed in such a way that it may be used as the basis for conventional Java applications or as an independent OSGi service. The dynamic nature of the Reaction framework makes it a good candidate for use within OSGi based systems and this chapter will examine how the Reaction service may be integrated into the OSGi platform.

8.1 Creating the Reaction OSGi Bundle

The Reaction OSGi bundle consists of the API package, the core implementation package, the OSGi wrapper package and the utilities package. OSGi bundle creation may be carried out using the `bnd` bundle creation tool originally written by Peter Kriens [1] using the configuration shown in Listing 8.1. The full version of this configuration file may be found in the root directory of the source code distribution as `Reaction.bnd`.

The bundle configuration specifies that the only exported package is the API package (`com.zynaptic.reaction`). This provides the only mechanism for accessing the functionality of the Reaction framework from other OSGi bundles. The reactor lifecycle management is now carried out automatically by the bundle activator, which starts the reactor when the bundle is started and shuts it down when the bundle is stopped.

The standard OSGi configuration makes use of the default bundle activator. This uses hardcoded implementations of the monotonic clock source and logging target as described below.

- The monotonic clock source used by the default activator is the fixed-up wall clock described in Section 2.1.1. This is the 'lowest common denominator' monotonic clock source which will work on all supported platforms.
- The standard log target is the optional OSGi log service wrapper described in Section 2.2.3. It will be used if the standard OSGi logging service is

Listing 8.1 Basic OSGi Bundle Metadata

```
Bundle-Name : Zynaptic Reaction
Bundle-SymbolicName : com.zynaptic.reaction
Bundle-Version : 0.6
Bundle-Description : Zynaptic Reaction framework.

Private-Package : \
com.zynaptic.reaction.core, \
com.zynaptic.reaction.osgi, \
com.zynaptic.reaction.util

Import-Package : \
org.osgi.framework; version="[1.6,2.0)", \
org.osgi.util.tracker; version="[1.5,2.0)", \
org.osgi.service.log; version="[1.3,2.0)"; resolution:="optional"

Export-Package : \
com.zynaptic.reaction

Bundle-Activator : \
com.zynaptic.reaction.osgi.ReactionDefaultActivator
```

active when the Reaction bundle is resolved.

- The backup log target redirects log messages to the standard console output, as described in Section 2.2.1. It will be used if the standard OSGi logging service is not available when the Reaction bundle is resolved.

The default bundle activator is applicable to all platforms supported by the OSGi framework and will be a good choice for most applications. However, there are situations where a different choice of monotonic clock source or logging target would be applicable. Creating a modified version of the bundle activator which supports a different combination of monotonic clock source and logging target is simply a matter of changing the `MonotonicClockSource` and `ReactorLogTarget` implementation classes. It is good practise to create a new activator class when making such changes, which will result in a corresponding change to the `Bundle-Activator` descriptor in the bundle metadata.

8.2 Activating the Reaction OSGi Service

Within any given OSGi based application, the Reaction service should generally be treated as an infrastructure component. This means that it should be started prior to application startup and remain active throughout a given application session. This assumption makes application development much more straightforward, since the application components can then assume that the Reaction

service is always available in normal operation. Application components will still need to track the Reaction service, but loss of the Reaction service can then be treated as a fatal error and application recovery is not required.

The standard way of enforcing this activation order is to make use of the OSGi framework's start level service, or the new start level API. Typically, the OSGi framework startup process should ensure that the following activation order is observed:

1. The standard OSGi framework service will always be started first.
2. The standard OSGi log service should be started to provide the basic logging functionality.
3. Any required backend logging services should be started and attached to the OSGi log service.
4. The Reaction service should be started and will automatically attach to the existing OSGi log service.
5. The core application components may be started.
6. Additional application plugins can be installed, started, stopped and uninstalled as required.

8.3 Accessing the Reaction OSGi Service

Once activated, the Reaction OSGi service can be used by any number of client OSGi bundles to provide full asynchronous programming support. However, there are various lifecycle considerations which must be taken into account when designing a client OSGi bundle. There are three lifecycle events which the client must observe which can be summarised by the methods provided by the example `LifecycleHandler` class shown in Listing 8.2.

The first lifecycle event is client *startup*. This should only occur once the OSGi client bundle has been started and it has been successfully bound to the Reaction service. In the example, the client startup method is used to start a repeating timer which will print a 'liveness' message to the console every second.

The second lifecycle event is client *teardown*. This occurs when the client bundle is stopped and the Reaction service is still running. Teardown implies a clean shutdown of the client, with the client cancelling all outstanding transactions with the Reaction service. In the case of the example, this involves cancelling the repeating timer which was set up during startup.

The third potential lifecycle event is client *abort*. This occurs when the client bundle is still running, but the Reaction service is shut down. In normal operation this should not occur, but clients should make every attempt to clean up on receiving an abort condition. Note that when a client is aborted it is no longer able to access the Reaction service, so cleaning up is restricted to tidying

Listing 8.2 Example of a Reaction Client Lifecycle Handler

```
public class LifecycleHandler {
  private Reactor reactor = null;
  private LivenessIndicator liveness = null;

  public void startup(Reactor reactor) {
    System.out.println("Running client startup handler.");
    liveness = new LivenessIndicator();
    reactor.runTimerRepeating(liveness, 1000, 1000, null);
    this.reactor = reactor;
  }

  public void teardown() {
    System.out.println("Performing orderly client shutdown.");
    reactor.cancelTimer(liveness);
  }

  public void abort() {
    System.out.println("Aborting client on loss of reactor.");
  }

  private class LivenessIndicator implements Timeable<Integer> {
    private int callbackCount = 0;
    public void onTick(Integer data) {
      System.out.println
        ("Reaction client is alive (" + (callbackCount++) + ")");
    }
  }
}
```

up independent resources. This will typically include such things as closing open files, killing local threads and closing GUI windows.

The lifecycle event handler shown in Listing 8.2 needs to be wrapped in the standard OSGi activation API in order to integrate as an OSGi client of the Reaction service. An example of such a wrapper is shown in Listing 8.3 which implements the standard OSGi activator and service tracker interfaces. Note that the OSGi specification makes no guarantees about the thread context to be used when issuing bundle activator and service tracker callbacks, so all method calls have been made synchronised.

A full implementation of this OSGi Reaction client example is provided in the `osgi` examples package. A corresponding bundle definition file is also included in the root directory (as `OsgiStartupExample.bnd`) which may be used in conjunction with the `bnd` bundle creation tool.

Listing 8.3 Example of a Reaction Client Activator

```
public class OsgiStartupExample implements BundleActivator,
    ServiceTrackerCustomizer<Reactor, Reactor> {

  private BundleContext context;
  private ServiceTracker<Reactor, Reactor> tracker;
  private LifecycleHandler lifecycleHandler;

  public synchronized void start
    (BundleContext context) {...}

  public synchronized Reactor addingService
    (ServiceReference<Reactor> ref) {...}

  public synchronized void stop
    (BundleContext context) {...}

  public synchronized void removedService
    (ServiceReference<Reactor> ref, Reactor service) {...}

  public synchronized void modifiedService
    (ServiceReference<Reactor> ref, Reactor service) {...}
}
```

Listing 8.4 Reaction Client Activator Start Method

```
public synchronized void start(BundleContext context) {
  System.out.println("Starting up Reaction client bundle.");
  this.context = context;

  // Starting up the reactor service.
  tracker = new ServiceTracker<Reactor, Reactor>
    (context, Reactor.class.getName(), this);
  tracker.open();
}
```

8.3.1 Starting a Reaction Service Client

As defined by the standard OSGi lifecycle, the first method to be called on starting a bundle is the bundle activator's `start` method. The implementation used by the example client activator is shown in Listing 8.4. In this case a new OSGi service tracker is created for the Reaction service, with the lifecycle wrapper being set up to receive notifications of status changes to the Reaction service. At this stage it is not known if the Reaction service is available, so it is not possible to call the lifecycle `startup` method.

Once the client bundle has been started, the OSGi framework will attempt to

Listing 8.5 Reaction Service Added Method

```
public synchronized Reactor addingService
    (ServiceReference<Reactor> ref) {
  Reactor reactor = null;

  // Multiple reactor services are not expected.
  if (lifecycleHandler == null) {
    reactor = context.getService(ref);
    lifecycleHandler = new LifecycleHandler();
    System.out.println("Found reactor.");

    // Run the client lifecycle startup.
    lifecycleHandler.startup(reactor);
  }

  // Return the reactor as the active service object.
  return reactor;
}
```

bind the client to a running instantiation of the Reaction service. If successful, this is notified using the service tracker's `addingService` callback method. The example used by the example lifecycle wrapper is shown in Listing 8.5. If a lifecycle handler is not already running, this method will create one and then invoke its `startup` method. By calling the `startup` method, the wrapper is indicating to the lifecycle handler than the client bundle has been correctly started and bound to a running Reaction service.

8.3.2 Clean Shutdown of a Reaction Client

When a Reaction client is being shut down, the OSGi framework will call the bundle activator's `stop` method while the client is still bound to the Reaction service. This allows the client bundle to perform a clean shutdown – cancelling any outstanding transactions with the Reaction service. The way in which this is handled in the example lifecycle wrapper code is illustrated in Listing 8.6.

The final action of the activator's `stop` method is to close the Reaction service tracker. This will then cause the framework to issue a callback to the service tracker's `removedService` method, as shown in Listing 8.7. Since the lifecycle handler has already undergone a clean shutdown at this stage, no further action is usually required.

8.3.3 Abortive Shutdown of a Reaction Client

As previously discussed, an abortive shutdown occurs when the Reaction service is removed while the Reaction client is still running. This causes the OSGi framework to issue a `removedService` callback as shown in Listing 8.7 *without*

Listing 8.6 Reaction Client Activator Stop Method

```
public synchronized void stop(BundleContext context) {
  System.out.println("Halting Reaction client bundle.");

  // Orderly shutdown of the client before removing reactor access.
  if (lifecycleHandler != null) {
    lifecycleHandler.teardown();
    lifecycleHandler = null;
  }
  tracker.close();
}
```

Listing 8.7 Reaction Service Removed Method

```
public synchronized void removedService
    (ServiceReference<Reactor> ref, Reactor service) {
  System.out.println("Lost reactor.");

  // Abort the client service if it is still running.
  if (lifecycleHandler != null) {
    lifecycleHandler.abort();
    lifecycleHandler = null;
  }
}
```

previously having called the activator's `stop` method. The consequence of this is that an abortive shutdown will be detected, causing the lifecycle handler's `abort` method to be executed.

Chapter 9

User Interface Design

One of the interesting aspects of the Reaction framework is that it provides event loop functionality which is completely independent of any standard GUI based event infrastructure. Part of the rationale for this is that the Reaction framework is designed to work on a wide range of target platforms – including the headless CDC Foundation profile. Another perspective is that by making this separation, use of the *Model-View-Controller* design pattern [4] is no longer a matter of the application designer's discretion; it becomes an essential part of the application framework.

9.1 The Model-View-Controller Pattern

The Model-View-Controller (MVC) is one of the most widely known design patterns in current use. It is based on the separation of responsibilities between the core application logic (the model), the presentation layer (the view) and user interaction handling (the controller). The Reaction framework is intended for use when implementing the model portion of the MVC triumvirate. Dealing with presentation and control is left to an appropriate user interface framework.

Use of the MVC pattern is particularly powerful when combined with OSGi support, since the OSGi framework allows the Reaction based model to be implemented as an independent OSGi component. GUI-specific viewer and controller components can then be attached at runtime using the OSGi service model. This means that the same application model component can be deployed across multiple platforms with different UI frameworks, such as Swing, JavaFX, SWT, Android or a GWT server.

9.2 Model Implementation Using Reaction

The concurrency model used when implementing the application model within the Reaction framework is well defined. All callback execution is carried out within the context of the main reactor thread, with long running tasks being

Listing 9.1 Control Interface For Simple Stopwatch Example

```
public interface ApplicationControl {
  public void timerStart();
  public void timerStop();
  public void timerReset();
  public void timerQuit();
}
```

farmed out to a worker thread pool and completing via the standard callback mechanism. This essentially eliminates the need for synchronisation, since all object state updates which rely on callbacks will execute atomically within the main reactor thread.

Although the core application code can take advantage of this clean concurrency model, complications are introduced the Reaction framework is used in conjunction with an independent GUI event loop. In order to address these problems, there are some common techniques which are useful when implementing the model's interface to the view and controller components.

9.2.1 Adding The Controller Interface

One of the main complicating factors when implementing the control interface on the model is synchronisation between the user interface and the model state. Since the controller part of the user interface will typically be running in an independent thread there is ample scope for introducing race conditions and potential deadlocks. In order to mitigate these problems it is advisable that the control interface on the model adopts the following rules:

- Always carry out model state changes within the context of the reactor thread.
- Assume controller actions can occur in the context of any thread.

It is relatively easy to achieve these objectives by making use of the Reaction framework's signal event functionality. A key property of signal events are that they can be triggered from any thread context and will then issue signal callbacks within the context of the main reactor thread. By way of example, consider a simple 'stopwatch' example that provides a control interface with timer *start*, *stop*, *reset* and *quit* options. The Java interface definition for such a control interface would look like the example shown in Listing 9.1.

In order to make the control interface safe to call from any thread context, some mechanism of handing the API call from the GUI thread to the reactor's main thread is required. To achieve this each API call can be mapped to a signal event, with the application logic being implemented in the signal event handlers. The use of Java generics allows the signal event data parameter to be strongly typed, and this will usually map to an immutable data type which

Listing 9.2 Example of Control Interface Command Identifiers

```
...
  enum CommandIds {
    TIMER_START,
    TIMER_STOP,
    TIMER_RESET,
    TIMER_QUIT
  };
...
```

Listing 9.3 Mapping Control Interface API Calls to Signal Events

```
...
  public void timerStart() {
    controlInterfaceSignal.signal(CommandIds.TIMER_START);
  }
  public void timerStop() {
    controlInterfaceSignal.signal(CommandIds.TIMER_STOP);
  }
  public void timerReset() {
    controlInterfaceSignal.signal(CommandIds.TIMER_RESET);
  }
  public void timerQuit() {
    controlInterfaceSignal.signal(CommandIds.TIMER_QUIT);
  }
...
```

wraps the API call parameters for propagation to the appropriate signal event handler. In the case of the example, a simple enumeration may be used to encode the nature of the API call, as shown in Listing 9.2.

Given a suitable signal event parameter for each API call, it is then possible to dispatch the calls via a signal callback, as illustrated in Listing 9.3. Note that this example is simplified by mapping all of the API calls to a single signal. In more complex applications, multiple different signal objects may be used in order to increase modularity.

Once the control interface API calls have been converted to signal events they can be handled via a suitable signalable object which is registered to receive the generated signal callbacks. The `onSignal` callback method of such a signalable object may be seen in Listing 9.4.

In this example, the `commandId` value which is passed as the signal parameter is used to differentiate between the different API calls. This allows the appropriate control action to be carried out from the context of the main reactor thread, eliminating the need for additional synchronisation. Note that a null `controlId` parameter is used to imply signal finalization on application shutdown.

Listing 9.4 Control Command Execution in Reactor Thread Context

```
...
  public void onSignal(Signal<CommandIds> signal, CommandIds commandId) {

    // Shutdown on signal finalization.
    if (commandId == null) {
      ...
    } else {
      switch (commandId) {

        // Start the timer.
        case TIMER_START:
          ...
        // Stop the timer.
        case TIMER_STOP:
          ...
        // Reset the timer.
        case TIMER_RESET:
          ...
        // Exit the timer stopwatch application.
        case TIMER_QUIT:
          ...
      }
    }
  }
...
```

This general pattern of using signal events to decouple control-side API calls from updates to the model state should be employed by the model component for all control interfaces. However, there is still considerable flexibility in terms of specifying more complex signal parameter data objects or using multiple signals to partition the control interface functionality.

9.2.2 Adding The View Interface

An approach to implementing a viewer interface using the *Publisher-Subscriber* pattern has already been introduced as part of the discussion of signal events in Section 4.4. In short, this relies on the application model component providing a suitable update notification signal and a range of getter methods which expose the current state of the published data. In the case of the simple stopwatch example shown in Listing 9.5, this implies a single getter method which returns the current elapsed time.

When implementing a viewer interface to the core application model, it is necessary to make certain assumptions about the types of viewer component which are supported. These assumptions lead to the following general rules which should be followed when implementing the viewer interface:

Listing 9.5 View Interface For Simple Stopwatch Example

```
public interface ApplicationView {
  public Signal<Integer> getTimerUpdateSignal();
  public int getTimerData();
}
```

- Any modifications to the published data should be carried out in the context of the main reactor thread.
- Callbacks which change the state of the published data should always leave the data in an internally consistent state.
- All calls to the view interface getter functions should be made in the context of the update notification signal callback.
- It is assumed that any data returned by the view interface getter functions may be cached by the view component, so copy on read should be used for non-primitive data.

A key aspect of these assumptions is that the viewer component carries out all rendering for a given update within the context of the update signal notification callback. The implication of this is that the model view cannot then be modified while the viewer is actively rendering the view data.

In certain circumstances it may not be viable to carry out all rendering within the update notification callback – for example, when rendering to remote client via a framework such as GWT. However, it is the responsibility of the viewer component to deal with such issues, not the core application model.

Typically, GUI frameworks which support delayed rendering will need to read back the view data in the context of the update notification callback and then cache it for future rendering. In order to support this, all data returned from view interface getter functions should be immutable, using 'copy on read' for non-native objects. Where this is not practicable, this should be explicitly stated in the interface contract.

9.3 GUI Implementation Using Swing

It is often the case that individual GUI components can aggregate the roles of both controller and viewer within the MVC pattern, and this is the case for the simple stopwatch example presented here. In the example, the aggregate viewer and controller class is called `ViewerController` and the full implementation is provided in the `swing` examples package as `SwingGuiExample`.

9.3.1 Swing Controller Implementation

For the controller functionality of the `ViewerController` component, a number of buttons are added to the GUI window for starting, stopping and resetting

Listing 9.6 Adding Swing Control Buttons to the GUI

```
...
  // Populate the content panel with the control buttons.
  JButton startButton = new JButton("Start");
  panel.add(startButton);
...
  // Add action listeners to the control buttons and wire
  // them up to the control interface.
  startButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
      appControl.timerStart();
    }
  });
...
```

Listing 9.7 Adapting Swing Window Controls to Model Interface

```
...
  // Create the basic frame, wiring up the window close event
  // to the application shutdown control method.
  frame = new JFrame();
  frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
  frame.setTitle("Reaction Stopwatch Example");
  frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      appControl.timerQuit();
    }
  });
...
```

the stopwatch. These correspond directly to the `timerStart`, `timerStop` and `timerReset` methods on the model's control interface. Since the control interface methods are thread safe, it is possible to call them directly from the context of the GUI thread using a standard Swing `ActionListener` object as shown in Figure 9.6.

When connecting control buttons to the model control interface, an anonymous instantiation of the `ActionListener` class is specialised by the addition of the `actionPerformed` method, which simply invokes the required method on the model control interface. In this example the control interface is named `appControl` and is an instance of the `ApplicationControl` interface given in Listing 9.1.

There is also the 'close' button on the window frame which corresponds to the `timerQuit` method on the control interface. Again, this may be called directly from the GUI thread using a standard Swing `WindowAdapter` object as shown in Listing 9.7.

Listing 9.8 Caching the Swing Stopwatch Data on Reaction Update

```
...
  // Local version of the current stopwatch time.
  private volatile String currentTime = "UNKNOWN";

  // Receive application view updates via the signal callback.
  public void onSignal(Signal<Integer> signal, Integer data) {
    if (data == null) {
      System.out.println("View interface shut down.");
    } else {
      int timeVal = appView.getTimerData();
      updateCurrentTime(timeVal);
    }
  }

  // Print out the current stopwatch time.
  private void updateCurrentTime(int timeVal) {
    DecimalFormat formatter = new DecimalFormat("00");
    currentTime = "" + formatter.format(timeVal / 3600000) + ":"
        + formatter.format((timeVal % 3600000) / 60000) + ":"
        + formatter.format((timeVal % 60000) / 1000) + "."
        + formatter.format((timeVal % 1000) / 10);
    panel.repaint();
  }
...
```

9.3.2 Swing Viewer Implementation

When implementing the viewer functionality using the Swing framework, it is important to bear in mind the issues described in Section 9.2.2 with respect to delayed rendering. In Swing, the process of painting the GUI components is carried out in an independent thread which accepts asynchronous notifications of changes to the presentational data. Therefore, the presentational data must be cached on receiving update notifications from the main reactor thread.

In the example given here, the viewer component accesses the current model state using the `appView` instance of the view interface previously shown in Listing 9.5. This provides access to a single item of presentational data, in the form of the current stopwatch timer value. In this case the raw timer data is not cached directly, but pre-formatted within the context of the Reaction view update callback. This is illustrated by the code fragment shown in Listing 9.8.

Once the cached presentational data has been updated, the Swing framework is notified of the changes using the panel repaint request method. This causes the displayed view of the data to be repainted at an arbitrary point in the future, within the context of an arbitrary Swing framework redraw thread.

The procedure used to repaint the displayed view from within the context of the Swing framework redraw thread is a standard text rendering routine, as

Listing 9.9 Rendering the Stopwatch Data on Swing Paint Request

```
...
  // Create a panel with a dedicated 'paintComponent' method
  // which prints the current stopwatch time.
  panel = new JPanel() {
    private Font f = new Font("Sans", Font.BOLD, 24);
    public void paintComponent(Graphics g) {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;
      g2.setFont(f);
      Rectangle2D bounds = f.getStringBounds
          (currentTime, g2.getFontRenderContext());
      g2.drawString(currentTime,
          (int) (getWidth() - bounds.getWidth()) / 2,
          2 * (int) (getHeight() - bounds.getHeight()) / 3);
    }
  };
...
```

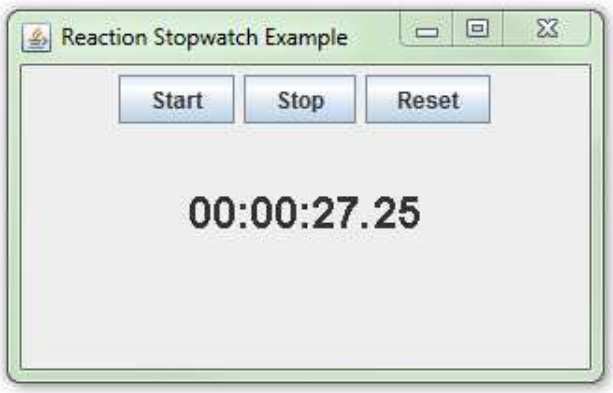

Figure 9.1: Running the Swing Based Stopwatch Example

shown in Listing 9.9. This takes the currently cached string representation of the stopwatch time and renders it to the screen at a specific position in the displayed window.

The combination of the buttons added to the GUI window in Section 9.3.1 and the rendered version of the current stopwatch time yields the window display shown in Figure 9.1. This looks and behaves just like a conventional Swing based application, even though the underlying Reaction based model is completely independent of the GUI toolkit being used.

Listing 9.10 Adding JavaFX Control Buttons to the GUI

```
...
  // Create the horizontal row of control buttons.
  TilePane buttonBox = new TilePane(Orientation.HORIZONTAL);
  buttonBox.getStyleClass().add("control-box");

  // Create the start button and connect it to the timer start hook.
  Button startButton = new Button("Start");
  startButton.getStyleClass().add("control-button");
  startButton.setMaxWidth(Double.MAX_VALUE);
  startButton.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
      appControl.timerStart();
    }
  });
  buttonBox.getChildren().add(startButton);
...
```

9.4 GUI Implementation Using JavaFX

Recent Java releases now include a new user interface framework in the form of JavaFX [8]. This is huge improvement over the ageing Swing/AWT framework and provides a much more suitable platform for implementing the GUI portion of Reaction based applications.

The JavaFX example presented here makes use of the same underlying application model as the previous Swing based example, replacing only the viewer and controller parts of the application. As for the Swing based example, the viewer and controller roles are combined into a single class called `ViewerController` and the full implementation is included in the `javafx` examples package as `JavaFxExample`.

9.4.1 JavaFX Controller Implementation

As for the previous Swing based example, a number of buttons are added to the GUI window for starting, stopping and resetting the stopwatch. These correspond directly to the `timerStart`, `timerStop` and `timerReset` methods on the model's control interface. Since the control interface methods are thread safe, it is possible to call them directly from the context of the GUI thread using a standard action event handler object as shown in Listing 9.10.

In the example, the control buttons are all added to a horizontal box, which will place them in the `TilePane` widget container. The appearance of the enclosing box and the control buttons themselves is configured using the specified style classes from the `JavaFxExample.css` stylesheet

There is also the 'close' button on the window frame which corresponds

Listing 9.11 Adapting JavaFX Window Controls to Model Interface

```
...
  // Handle window close events.
  stage.setOnCloseRequest(new EventHandler<WindowEvent>() {
    public void handle(WindowEvent windowEvent) {
      if (windowEvent.getEventType() ==
          WindowEvent.WINDOW_CLOSE_REQUEST) {
        appControl.timerQuit();
      }
    }
  });
...
```

Listing 9.12 Binding a JavaFX String Property to Viewer Text

```
...
  // Create a pane to hold the current timer string and bind the string
  // value to the observable timer value.
  StackPane timerValuePane = new StackPane();
  timerValuePane.getStyleClass().add("timer-pane");
  Text timerValueText = new Text();
  timerValueText.getStyleClass().add("timer-text");
  timerValueText.textProperty().bind(
      observableTimer.getObservableTimerString());
  timerValuePane.getChildren().add(timerValueText);
...
```

to the `timerQuit` method on the control interface. Again, this may be called directly from the GUI thread using a standard JavaFX window event handler object as shown in Listing 9.11.

9.4.2 JavaFX Viewer Implementation

The implementation of viewer functionality in JavaFX is made much easier by the support for value binding to user interface elements. This means that the application developer does not need to explicitly manage the process of redrawing the user interface view. However, JavaFX still imposes the restriction that updates to bound property values must take place in the context of the main Java FX thread.

In the example, the signalable interface to the application view and the observable timer property value used by the JavaFX framework are wrapped in a separate class called `ObservableTimer`. This allows the observable timer value to be bound to the JavaFX user interface element as shown in Listing 9.12. Once this binding has been established, any changes to the observable timer string value will automatically cause the displayed timer text to be redrawn.

Listing 9.13 Caching the JavaFX Stopwatch Data on Reaction Update

```
...
  // Receive application view updates via the signal callback.
  public synchronized void onSignal(Signal<Integer> signalId,
      Integer timerUpdateSequence) {
    if (timerUpdateSequence == null) {
      System.out.println("View interface shut down.");
    } else {
      int timeVal = appView.getTimerData();
      DecimalFormat formatter = new DecimalFormat("00");
      timerString = "" + formatter.format(timeVal / 3600000) + ":"
          + formatter.format((timeVal % 3600000) / 60000) + ":"
          + formatter.format((timeVal % 60000) / 1000) + "."
          + formatter.format((timeVal % 1000) / 10);
      Platform.runLater(this);
    }
  }
...
```

Listing 9.14 Updating the JavaFX Observable Stopwatch String Value

```
...
  // Updates the value held by the observable string.
  public synchronized void run() {
    stringProperty.set(timerString);
  }
...
```

The observable timer class manages the forwarding of updates from the model to the JavaFX observable property updates which will trigger the user interface to be updated. The first task is to convert the numeric timer value to a string which is suitable for display, followed by a request to the JavaFX framework to apply this change to the observable string property value from the context of the main JavaFX thread.

The process of converting the integer timer value to a displayable string is shown in Listing 9.13. After the display string has been formatted, the `Platform.runLater` call is made in order to schedule the `Runnable.run` method for subsequent execution in the context of the JavaFX main thread.

The procedure used to update the observable string value from within the context of the JavaFX main thread is shown in Listing 9.14. This takes the currently cached string representation of the stopwatch time and updates the observable string value which is bound to the user interface display.

The use of property binding in JavaFX provides for a much more elegant viewer implementation than the equivalent Swing based code. Furthermore, while all the styling associated with the Swing view has to be hardcoded into

Listing 9.15 Applying JavaFX Styles to the Displayed Timer String

```
...
.timer-text {
  -fx-fill: white;
  -fx-font-family: "Arial Black";
  -fx-font-size: 32;
  -fx-font-weight: bold;
  -fx-effect: innershadow(three-pass-box, rgba(0,0,0,0.7), 6, 0.0, 0, 2);
}
...
```

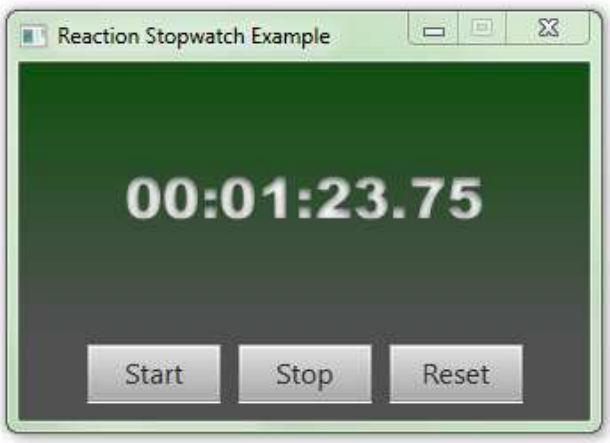

Figure 9.2: Running the JavaFX Based Stopwatch Example

the painting callback, the JavaFX implementation does not need to contain any formatting information at all.

In the JavaFX implementation, all of the text styling associated with the viewer functionality is assigned using the `styleClass` method calls made when declaring the text elements in Listing 9.12. These identify the text styling to be used as defined in the associated stylesheet. For the purposes of the example, some simple effects can be applied to the displayed string as shown in Listing 9.15.

The combination of the buttons added to the GUI window in Section 9.4.1 and the rendered version of the current stopwatch time yields the window display shown in Figure 9.2. This has a much more polished look than the corresponding Swing application, despite the fact that much less code is required to produce the desired result. However, both applications still share the same core application code - further emphasising the independence of the underlying Reaction based model from the GUI toolkit being used.

References

[1] The Bnd OSGi bundle tool website. `http://bnd.bndtools.org`.

[2] The main OSGi Alliance website. `http://www.osgi.org/`.

[3] The main Twisted Matrix Labs website. `http://twistedmatrix.com/`.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture : Volume 1.* John Wiley and Sons, Chichester, England, 1996.

[5] Doug Lea and Contributors. The JSR-166 Concurrency Utility Library. `http://jcp.org/en/jsr/detail?id=166`.

[6] Sheng Liang. *The Java Native Interface : Programmer's Guide and Specification.* Addison-Wesley, Reading, Massachusetts, United States, 1999.

[7] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture : Volume 2.* John Wiley and Sons, Chichester, England, 2000.

[8] James L. Weaver, Weiqi Gao, Stephen Chin, Dean Iverson, and Johan Vos. *Pro JavaFX 2 : A Definitive Guide to Rich Clients with Java Technology.* Apress / Springer, New York, United States, 2012.