http://kieker.sourceforge.net

Kieker 1.3-dev User Guide

Nils Ehmke, André van Hoorn,* and Reiner Jung

April 28, 2011

Software Engineering Group, http://se.informatik.uni-kiel.de
Dept. Computer Science, Christian Albrechts University of Kiel, Germany

*Corresponding author; e-mail: avh@informatik.uni-kiel.de

Contents

1 Introduction

Modern software applications are often complex and have to fulfill a large set of functional and non-functional properties. The internal behavior of such large systems cannot easily be determined on the basis of the source code. Furthermore, existing applications often lack sufficient documentation which makes it cumbersome to extend and change them for future needs. A solution to these problems can be monitoring, which allows to log the behavior of the application and to discover the application-internal control flows and response times of method executions.

The monitoring of the behavior can help in detecting performance problems and faulty behavior, capacity planning, and many other areas. The Kieker framework provides the necessary monitoring capabilities and comes with tools and libraries for the analysis of monitored data. Kieker was designed for continuous monitoring in production systems inducing only a very low overhead.

1.1 What is Kieker?

The present version of Kieker is a monitoring and analysis framework for Java applications. Support for other platforms, such as .NET, is currently under development. Figure 1.1 shows the framework's composition based on the two main components Kieker.-Monitoring and Kieker.Analysis.

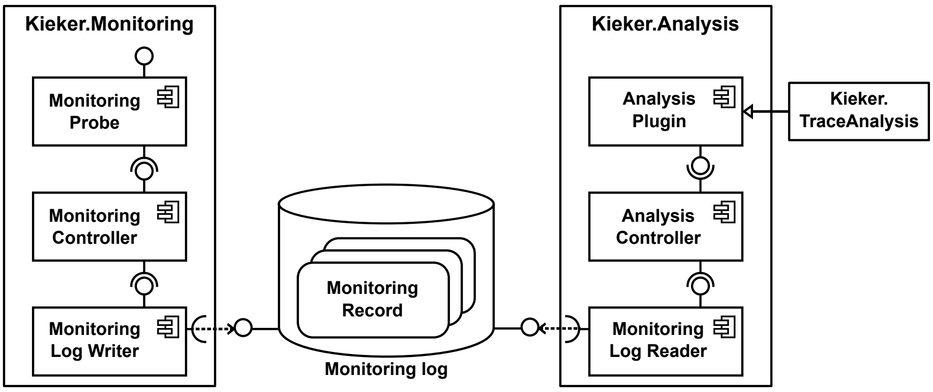


Figure 1.1: Overview of the framework components

The Kieker.Monitoring component is responsible for program instrumentation, data collection, and logging. Its core is the `MonitoringController`. The component Kieker.Analysis is responsible for reading, analyzing, and visualizing the monitoring data. Its core is the `AnalysisController` which manages the life-cycle of the monitoring reader and all analysis plugins.

The monitoring and analysis parts of the Kieker framework are composed of subcomponents which represent the different functionalities of the monitoring and analysis tasks. The important interaction pattern among the components is illustrated in Figure 1.2 but will be explained furthermore throughout the course of this user guide.

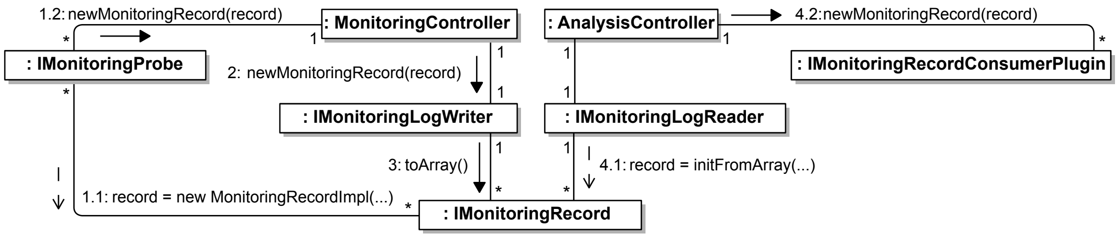


Figure 1.2: Communication among Kieker framework components

The monitoring probes create the monitoring records containing the monitoring data and deliver them to the monitoring controller. The monitoring controller employs the monitoring writers to write these monitoring records to a monitoring log or stream. For analyzing purposes, a monitoring reader reads the records from the monitoring log/stream. These records can then be further processed by the analysis plugins.

Kieker includes monitoring writers and readers for filesystems, SQL databases, and the Java Messaging Service (JMS) [2]. A special feature of Kieker is the ability to monitor and analyze (distributed) traces of method executions and corresponding timing information. For monitoring this data, Kieker includes monitoring probes employing AspectJ [7], Java EE Servlet [3], Spring [4], and Apache CXF [5] technology. The Kieker.TraceAnalysis tool, itself implemented as a Kieker.Analysis plugin (Figure 1.1), allows to reconstruct and visualize architectural models of the monitored systems, e.g., as sequence and dependency diagrams.

1.2 Structure of this User Guide

Based on a simple example, Chapter 2 demonstrates how to manually instrument Java programs with Kieker.Monitoring in order to monitor timing information of method executions, and how to use Kieker.Analysis to analyze the monitored data. Chapter 3 provides a more detailed description of Kieker.Monitoring and shows how to implement and use custom monitoring records, monitoring probes, and monitoring writers. A more detailed description of Kieker.Analysis and how to implement and use custom monitoring readers, and analysis plugins follows in Chapter 4. Chapter 5 demonstrates how to use Kieker.TraceAnalysis for monitoring, analyzing, and visualizing trace information. Additional resources are included in the appendix.

☞ The Java sources presented in this user guide are included in the `examples/userguide/` directory of the Kieker distribution (see Section 2.1).

2 Quick Start Example

This chapter provides a brief introduction to Kieker based on a simple Bookstore example application. Section 2.1 explains how to download and install Kieker. The Bookstore application itself is introduced in Section 2.2, while the following sections demonstrate how to use Kieker for monitoring (Section 2.3) and analyzing (Section 2.4) the resulting monitoring data.

2.1 Download and Installation

The Kieker download site[1] provides archives of the binary and source distribution, the Javadoc API, as well as additional examples. For this quick start guide Kieker's binary distribution, e.g., `kieker-1.3-dev_binaries.zip`, is required and must be downloaded. After having extracted the archive, you'll find the directory structure and contents shown in Figure 2.1.

```
kieker-1.3-dev/
├── bin/ ........................................ Call scripts for Kieker tools
│   └── ...
├── doc/ ..........................................................................
│   └── kieker-1.3-dev_userguide.pdf ............... PDF file of this document
├── dist/ ................................... The Kieker framework libraries
│   ├── kieker-1.3-dev.jar
│   └── kieker-monitoring-servlet-1.3-dev.war
├── examples/
│   └── userguide/ .............. Source code of the examples in this document
│       └── ...
├── lib/ ......................................... Libraries required by Kieker
│   ├── commons-logging-1.1.1.jar
│   └── ...
└── META-INF/ ................................... Example configuration files
    ├── kieker.monitoring.properties
    └── ...
```

Figure 2.1: Directory structure and contents of Kieker's binary distribution

[1] `http://sourceforge.net/projects/kieker/files`

The Java sources presented in this user guide are included in the `examples/user-guide/` directory. The file `kieker-1.3-dev.jar` contains the Kieker.Monitoring and Kieker.Analysis components, as well as the Kieker.TraceAnalysis tool. A Servlet-based Web application, provided in `kieker-monitoring-servlet-1.3-dev.war`, can be used to control the status of Kieker.Monitoring in Java EE environments. The file `kieker.monitoring.properties` is a sample configuration file for Kieker.Monitoring, as detailed in Chapter 3. Since Kieker uses the Apache Commons library [6] as a logging interface, the file `commons-logging-1.1.1.jar` is the only dependency to a third-party library which is needed to execute Kieker in any case.

2.2 Bookstore Example Application

The Bookstore application is a small sample application resembling a simple bookstore with a market-place facility where users can search for books in an online catalog, and subsequently get offers from different book sellers. Figure 2.2 shows a class diagram describing the structure of the bookstore and a sequence diagram illustrating the dynamics of the application.

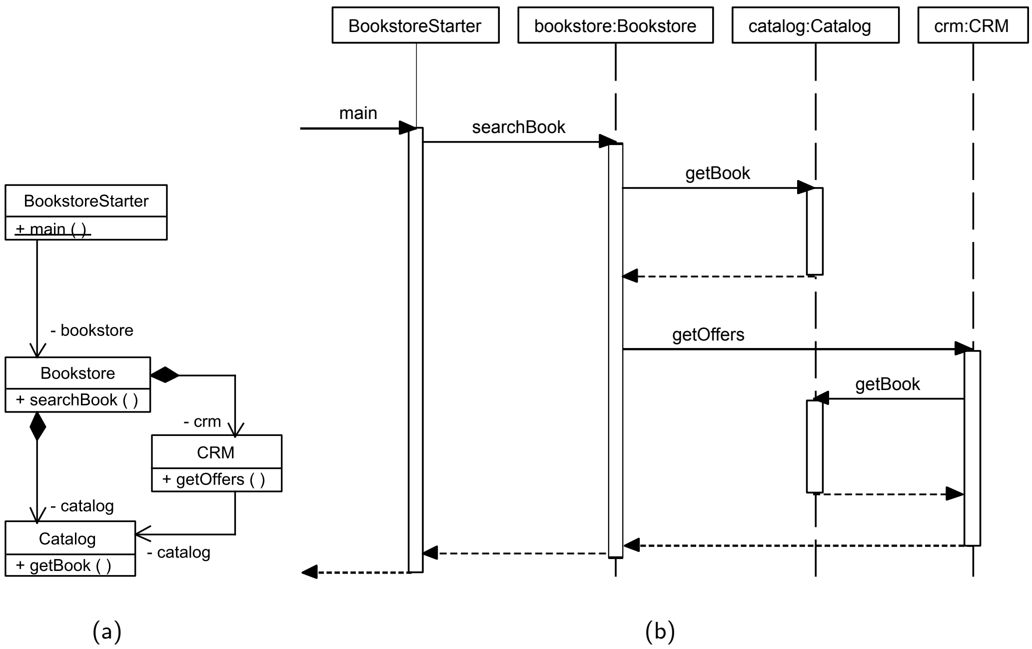


Figure 2.2: UML class diagram (a) and sequence diagram (b) of the Bookstore application

The bookstore contains a catalog for books and a customer relationship management system (CRM) for the book sellers. To provide this service, the different classes provide operations to initialize the application, search for books, and get offers or searched books. In this example, the methods implementing these operations are merely stubs. However, for the illustration of Kieker they are sufficient and the inclined reader may extend the application into a real bookstore.

The directory structure of the Bookstore example is shown in Figure 2.3 and comprises four Java classes in its source directory `src/bookstoreApplication/` which are explained in detail below.

```
examples/
└── userguide/
    └── ch2–bookstore-application/
        └── src/ ........................................ The directory for the source code files
        ├── bookstoreApplication/
        │   ├── Bookstore.java
        │   ├── BookstoreStarter.java
        │   ├── Catalog.java
        │   └── CRM.java
        ├── build.properties
        └── build.xml ................................ Optional built script for the application
```

Figure 2.3: The directory structure of the Bookstore application

The Java sources of the not instrumented Bookstore application can be found in the `examples/userguide/ch2-bookstore-application/` directory.

The class `BookstoreStarter`, shown in Listing 2.1, contains the application's *main* method, i.e., the program start routine. It initializes the `Bookstore` and issues five search requests by calling the *searchBook* method of the `bookstore` object.

```java
package bookstoreApplication;

public class BookstoreStarter {

    public static void main(String[] args) {
        Bookstore bookstore = new Bookstore();
        for (int i = 0; i < 5; i++) {
            System.out.println("Bookstore.main: Starting request " + i);
            bookstore.searchBook();
        }
    }
```

```
}
```

Listing 2.1: BookstoreStarter.java

The `Bookstore`, shown in Listing 2.2, contains a catalog and a CRM object, representing the catalog of the bookstore and a customer relationship management system which can provide offers for books out of the catalog. The business method of the bookstore is *searchBook()* which will first check the catalog for books and then check for offers.

In a real application these methods would pass objects to ensure the results of the catalog search will be available to the offer collecting method. However, for our example we omitted such code.

```java
package bookstoreApplication;

public class Bookstore {

    private final Catalog catalog = new Catalog();
    private final CRM crm = new CRM(catalog);

    public void searchBook() {
        catalog.getBook(false);
        crm.getOffers();
    }
}
```

Listing 2.2: Bookstore.java

The customer relationship management for this application is modeled in the `CRM` class shown in Listing 2.3. It only provides a business method to collect offers which uses the catalog for some lookup. The additional catalog lookup is later used to illustrate different traces in the application.

```java
package bookstoreApplication;

public class CRM {
    private final Catalog catalog;

    public CRM(final Catalog catalog) {
        this.catalog = catalog;
    }

    public void getOffers() {
        catalog.getBook(false);
    }
}
```

Listing 2.3: CRM.java

The final class is `Catalog` shown in Listing 2.4. It resembles the catalog component in the application.

```
package bookstoreApplication;

public class Catalog {

    public void getBook(final boolean complexQuery) { }
}
```

Listing 2.4: Catalog.java

After this brief introduction of the application and its implementation, the next step is to see the example running. To compile and run the example, the commands in Listing 2.5 can be executed. This document assumes that the reader enters the commands in the example directory. For this first example this is `examples/userquide/ch2-bookstore-application/`.

> Windows comes with two commando interpreters called `cmd.exe` and `command.com`. Only the first one is able to handle wildcards correctly. So we recommend using `cmd.exe` for these examples.

```
> mkdir build
> javac src/bookstoreApplication/*.java -d build

> java -classpath build bookstoreApplication.BookstoreStarter
```

Listing 2.5: Commands to compile and run the Bookstore application

The first command compiles the application and places the resulting four class files in the `build/` directory. To verify the build process, the `build/` directory can be inspected. The second command loads the bookstore application and produces the output shown in Listing 2.6.

```
Bookstore.main: Starting request 0
Bookstore.main: Starting request 1
Bookstore.main: Starting request 2
Bookstore.main: Starting request 3
Bookstore.main: Starting request 4
```

Listing 2.6: Example run of the Bookstore application

In this section, the Kieker example application was introduced and when everything went well, the bookstore is a runnable program. Furthermore, the composition of the application and its function should now be present. The next Section 2.3 will demonstrate how to monitor this example application employing Kieker.Monitoring using manual instrumentation.

2.3 Monitoring with Kieker.Monitoring

In the previous Sections 2.1 and 2.2, the Kieker installation and the example application have been introduced. In this section, the preparations for application monitoring, the instrumentation of the application, and the actual monitoring are explained.

> In this example, the instrumentation is done manually. This means that the monitoring probe is implemented by mixing monitoring logic with business logic, which is often not desired since the resulting code is hardly maintainable. Kieker includes probes based on AOP (aspect-oriented programming, [1]) technology, as covered by Chapter 5. However, to illustrate the instrumentation in detail, the quick start example uses manual instrumentation.

The first step is to copy the Kieker jar-file `kieker-1.3-dev.jar` to the `lib/` directory of the example directory (see Section 2.2). The file is located in the `kieker-1.3-dev/dist/` directory of the extracted Kieker archive, as described in Section 2.1. The file `commons-logging-1.1.1.jar` is located in the `kieker-1.3-dev/lib/` directory and has to be copied to the `lib/` directory of the example application. The final layout of the example directories is illustrated in Figure 2.4.

```
examples/
└── userguide/
    └── ch2–manual-instrumentation/
        ├── build/ ............ Directory for the Java class files
        ├── lib/ .............. Directory for the required libraries
        │   ├── kieker-1.3-dev.jar
        │   └── commons-logging-1.1.1.jar
        └── src/ .............. The directory for the source code files
            └── ...
```

Figure 2.4: The directory structure of the Bookstore application with Kieker libraries

> The Java sources of the manually instrumented Bookstore application described in this section can be found in the `examples/userguide/ch2-manual-instrumentation/` directory.

Kieker maintains monitoring data as so-called monitoring records. Section 3.3 describes how to define and use custom monitoring record types. The monitoring record type used in this example is an *operation execution record* which is included in the Kieker distribution. Figure 2.5 shows the attributes which are relevant to this example. The record type will be detailed in Chapter 5 .

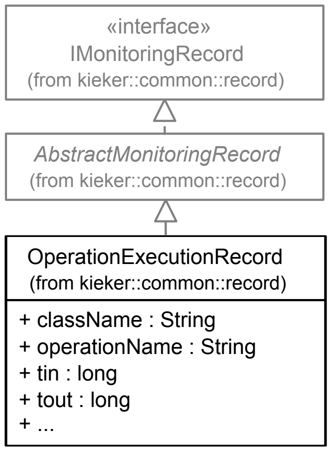


Figure 2.5: The class diagram of the operation execution record

The attributes relevant to this part are *className* and *operationName* for the class and method name, as well as *tin* and *tout* for the timestamp before and after the call of the instrumented method.

Listing 2.7 shows the instrumentation of the `Bookstore` class and its method *searchBook()*. In the lines 12 and 13, the monitoring controller is instantiated. It provides the monitoring service for the instrumentation.

```
12     private final static IMonitoringController MONITORING_CONTROLLER =
13             MonitoringController.getInstance();
14
15     public void searchBook() {
16         /* 1.) Call the Catalog component's getBook() method
17          *     and log its entry and exit timestamp using Kieker. */
18         final long tin = Bookstore.MONITORING_CONTROLLER.getTimeSource().getTime();
19         this.catalog.getBook(false);
20         final long tout = Bookstore.MONITORING_CONTROLLER.getTimeSource().getTime
               ();
21         final OperationExecutionRecord e =
22                 new OperationExecutionRecord(
23                 Catalog.class.getName(), "getBook(..)",
24                 tin, tout);
25         Bookstore.MONITORING_CONTROLLER.newMonitoringRecord(e);
26
27         /* 2.) Call the CRM catalog's getOffers() method
28          *     (without monitoring). */
29         this.crm.getOffers();
30     }
```

Listing 2.7: Instrumentation of the *getBook()* call in Bookstore.java

The lines 18 and 20 are used to determine the current time in nanoseconds before and after the *getBook()* call. In lines 21 to 24, a monitoring record for this measurement is created and initialized with the two time values. Additionally, the record has an attribute for the involved class `Catalog` and the called method *getBook()*. Finally the record is handed over to the monitoring controller which calls a monitoring writer to persist the record. In this example, the filesystem writer is used—Kieker uses this writer by default when no other writer is specified, as detailed in Section 3.5.

In addition to the instrumentation in the `Bookstore` class, the *getOffers()* method of the `CRM` class is instrumented as well. Similar to Listing 2.7, measurements are taken before and after the call of the `catalog`'s *getBook()* method, as shown in lines 19 and 21 of Listing 2.8. Not shown in the listing is the instantiation of the monitoring controller. However, it is done in the same way as illustrated in Listing 2.7. Finally, a record is created (see lines 23–26) and stored by calling the monitoring controller (see line 27).

```
17    public void getOffers() {
18        /* Call the Catalog component's getBook() method
19         * and log its entry and exit timestamp using Kieker. */
20        final long tin = CRM.MONITORING_CONTROLLER.getTimeSource().getTime();
21        this.catalog.getBook(false);
22        final long tout = CRM.MONITORING_CONTROLLER.getTimeSource().getTime();
23        final OperationExecutionRecord e =
24                new OperationExecutionRecord(
25                Catalog.class.getName(), "getBook()",
26                tin, tout);
27        CRM.MONITORING_CONTROLLER.newMonitoringRecord(e);
28    }
```

Listing 2.8: Instrumentation of the *getBook()* call in CRM.java

The next step after instrumenting the code is running the instrumented application. Listing 2.9 shows the two commands to compile and run the application under UNIX-like systems. Listing 2.10 shows the same commands for Windows.

```
▷ mkdir build
▷ javac src/bookstoreApplication/*.java -classpath lib/kieker-1.3-dev.jar -d build/

▷ java -classpath build/:
        lib/kieker-1.3-dev.jar:
        lib/commons-logging-1.1.1.jar
        bookstoreApplication.BookstoreStarter
```

Listing 2.9: Commands to compile and run the instrumented Bookstore under UNIX-like systems

```
▷ mkdir build
▷ javac src\bookstoreApplication\*.java -classpath lib\kieker-1.3-dev.jar -d build\

▷ java -classpath build\;
        lib\kieker-1.3-dev.jar;
        lib\commons-logging-1.1.1.jar
```

```
bookstoreApplication.BookstoreStarter
```

Listing 2.10: Commands to compile and run the instrumented Bookstore under Windows

> STOP Under Windows it is necessary to separate the classpath elements by a semicolon instead of a colon.

If everything worked correctly, a new directory for the monitoring data with a name similar to `kieker-20110427-142244899-UTC-Kaapstad-KIEKER-SINGLETON/` is created (see Listing 2.6). The numbers in the directory name represent the time and date of the monitoring. In Kieker's default configuration, the log directory can be found in the default temporary directory: under UNIX-like systems, this is typically `/tmp/`; check the environment variable `%temp%` for the location under Windows. The monitoring directory contains two types of files: `.dat` files containing text representations of the monitoring records and a file named `kieker.map` which contains information on the types of monitoring records used.

```
/tmp/
└── kieker-20110427-142244899-UTC-Kaapstad-KIEKER-SINGLETON/
    ├── kieker.map
    └── kieker-20110427-142244920-UTC-Thread-1.dat
```

Figure 2.6: Directory structure after a monitoring run

The Listings 2.11 and 2.12 show example file contents. The `.dat`-file is saved in CSV format (**C**omma **S**eparated **V**alues)—in this case, the values of a monitoring record are separated by semicolons. To understand the `.dat`-file structure the semantics have to be explained. For this quick start example only some of the values are relevant. The first value `$1` indicates the record type. The forth value indicates the class and method which has been called. And the seventh and eighth value are the start and end time of the execution of the called method.

```
$1;1303914164918306848;−1;bookstoreApplication.Catalog.getBook(..);N/A;−1
    ;1303914164915013739;1303914164917124977;N/A;−1;−1
$1;1303914164920544967;−1;bookstoreApplication.Catalog.getBook();N/A;−1
    ;1303914164918412684;1303914164920506469;N/A;−1;−1
```

Listing 2.11: kieker-20110427-142244920-UTC-Thread-1.dat (excerpt)

The second file is a simple mapping file referencing keys to monitoring record types. In Listing 2.12 the key `$1` is mapped to the type of operation execution records which were used in the monitoring. The key value corresponds to the key values in the `.dat`-file.

```
$1=kieker.common.record.OperationExecutionRecord
```

Listing 2.12: kieker.map

By the end of this section, two Java classes of the Bookstore application have been manually instrumented using Kieker.Monitoring and at least one run of the instrumented application has been performed. The resulting monitoring log, written to the `.dat`-file in CSV format, could already be used for analysis or visualization by any spreadsheet or statistical tool. The following Section 2.4 will show how to process this monitoring data with Kieker.Analysis.

2.4 Analysis with Kieker.Analysis

In this section, the monitoring data recorded in the previous section is analyzed with Kieker.Analysis. For this quick example guide, the analysis tool is very simple and does not show the full potential of Kieker. For more detail read Chapter 5 which uses Kieker.TraceAnalysis.

- **examples/**
  - **userguide/**
    - **ch2–manual-instrumentation/**
      - **build/** .... *Directory for the Java class files*
      - **lib/** .... *Directory for the required libraries*
        - ...
        - kieker-1.3-dev.jar
      - **src/** .... *The directory for the source code files*
        - **bookstoreApplication/**
          - ...
          - BookstoreAnalysisStarter.java
          - Consumer.java

Figure 2.7: Directory layout of the example application with the analysis files highlighted

The analysis application developed in this section comprises the files `Consumer.java` and `BookstoreAnalysisStarter.java`, as shown in Figure 2.7. These files can also be found in the directory `examples/userguide/ch2-manual-instrumentation/`.

Listing 2.13 on page 16 shows the content of the `Consumer.java` file which implements the `IMonitoringRecordConsumerPlugin` interface. This is the standard interface for the consumer of Kieker monitoring records. The consumer is part of the Kieker.Analysis component. It processes records provided by the monitoring readers (see Chapter 1). The consumer checks if the response time of each method call, in this case *getBook()*, is below a specified threshold value. This threshold is set during construction of the `Consumer` class (see lines 13–15).

For the data analysis, the method *newMonitoringRecord()* (see lines 23–40) is used.

```java
package bookstoreApplication;

import java.util.Collection;

import kieker.analysis.plugin.IMonitoringRecordConsumerPlugin;
import kieker.common.record.IMonitoringRecord;
import kieker.common.record.OperationExecutionRecord;

public class Consumer implements IMonitoringRecordConsumerPlugin {

  private long maxResponseTime;

  public Consumer(long maxResponseTime) {
    this.maxResponseTime = maxResponseTime;
  }

  @Override
  public Collection<Class<? extends IMonitoringRecord>> getRecordTypeSubscriptionList() {
    return null;
  }

  @Override
  public boolean newMonitoringRecord(IMonitoringRecord record) {
    if (!(record instanceof OperationExecutionRecord)) {
      return true;
    }
    OperationExecutionRecord rec = (OperationExecutionRecord) record;
    /* Derive response time from the record. */
    long responseTime = rec.tout - rec.tin;
    /* Now compare with the response time threshold: */
    if (responseTime > maxResponseTime) {
      System.err.println("maximum response time exceeded by "
          + (responseTime - maxResponseTime) + " ns: " + rec.className
          + "." + rec.operationName);
    } else {
      System.out.println("response time accepted: " + rec.className
          + "." + rec.operationName);
    }
    return true;
  }

  @Override
  public boolean execute() { return true; }

  @Override
  public void terminate(boolean error) { }

}
```

Listing 2.13: Consumer.java

This method is called for every monitoring record. At first, the method tests if the monitoring record are really `OperationExecutionRecord` instances, as these are the only record types it can process. Then the methods calculates the execution time of one recorded *getBook()* call. If the method call takes longer than specified, a message is written directly to the error stream.

The framework methods *terminate* and *execute* don't do anything in this example, due to the fact that this consumer does not need any initialization. If the consumer would, for example, use threads then these methods would be the correct location to start and stop them.

After implementing a consumer, the application's main class has to be created. In this case the main program is located in the `BookstoreAnalysisStarter.java` file shown in Listing 2.14.

```
1  package bookstoreApplication;
2
3  import kieker.analysis.AnalysisController;
4  import kieker.analysis.plugin.MonitoringRecordConsumerException;
5  import kieker.analysis.reader.MonitoringReaderException;
6  import kieker.analysis.reader.filesystem.FSReader;
7
8  public class BookstoreAnalysisStarter {
9
10     public static void main(final String[] args)
11             throws MonitoringReaderException, MonitoringRecordConsumerException {
12
13     if (args.length == 0) {
14       return;
15     }
16
17         /* Create Kieker.Analysis instance */
18         final AnalysisController analysisInstance = new AnalysisController();
19         /* Register our own consumer; set the max. response time to 1.9 ms */
20         analysisInstance.registerPlugin(new Consumer(1900000));
21
22         /* Set filesystem monitoring log input directory for our analysis */
23         final String inputDirs[] = {args[0]};
24         analysisInstance.setLogReader(new FSReader(inputDirs));
25
26         /* Start the analysis */
27         analysisInstance.run();
28     }
29 }
```

Listing 2.14: BookstoreAnalysisStarter.java

The `BookstoreAnalysisStarter` follows a simple scheme. Each analysis tool has to create at least one `AnalysisController` which can be seen in Listing 2.14 in line 18. Then the consumers are registered with the analysis instance. In this case, the previously described `Consumer` is instantiated and the maximum response time is set to 1.9 milliseconds. Line 24 sets the file system monitoring log reader which is initialized with the first command-line argument value as the input directory. The application expects the output directory from the earlier monitoring run (see Section 2.3) as the only argument value, which must be passed manually. The analysis started by calling its *run* method (line 27).

The Listings 2.15 and 2.16 describe how the analysis application can be compiled and executed under UNIX-like systems and Windows.

```
▷ mkdir build
▷ javac src/bookstoreApplication/*.java
         -classpath lib/kieker-1.3-dev.jar
         -d build/

▷ java -classpath
        build/:
        lib/kieker-1.3-dev.jar:
        lib/commons-logging-1.1.1.jar
        bookstoreApplication.BookstoreAnalysisStarter
        /tmp/kieker-20110427-142244899-UTC-Kaapstad-KIEKER-SINGLETON
```

Listing 2.15: Commands to compile and run the analysis under UNIX-like systems

```
▷ mkdir build
▷ javac src\bookstoreApplication\*.java
         -classpath lib\kieker-1.3-dev.jar
         -d build\

▷ java -classpath
        build\;
        lib\kieker-1.3-dev.jar;
        lib\commons-logging-1.1.1.jar
        bookstoreApplication.BookstoreAnalysisStarter
        C:\Temp\kieker-20110427-142244899-UTC-Kaapstad-KIEKER-SINGLETON
```

Listing 2.16: Commands to compile and run the analysis under Windows

You need to make sure that the application gets the correct path from the monitoring run. The consumer prints an output message for each record received. An example output can be found in Appendix D.1.

3 Kieker.Monitoring Component

☞ The Java sources of this chapter can be found in the `examples/userguide/ch3-4-custom-components/` directory of the binary release.

3.1 Kieker.Monitoring Configuration

Kieker.Monitoring is being configured by a properties file. A sample configuration file, which can be used as a template for custom configurations, is provided by the file `kieker.monitoring.properties` in the directory `kieker-1.3-dev/META-INF/` of the binary release (see Section 2.1).

In order to use a custom configuration file, its location needs to be passed to the JVM using the parameter *kieker.monitoring.configuration* as follows:

```
▷ java -Dkieker.monitoring.configuration=<ANY-DIR>/my.kieker.monitoring.properties [...]
```

Appendix B lists the template file with a documentation of all available properties. If no configuration file is passed to the JVM, a default configuration—according to this sample file—is being used by Kieker.Monitoring.

3.2 Monitoring Controller

The `MonitoringController` constructs and controls a Kieker.Monitoring instance and provides methods to, among others, log monitoring records (*newMonitoringRecord*) employing the configured monitoring writer and to retrieve the current timestamp (*currentTimeNanos*). The method *currentTimeNanos* returns the number of nanoseconds elapsed since 1 Jan 1970 00:00 UTC.

The class `MonitoringController` is implemented employing the singleton pattern. The singleton instance can be retrieved by calling the static method *getInstance*. Figure 3.1 shows a class diagram of the class `MonitoringController` including the methods just mentioned. The `MonitoringController` reads the configuration file, as described in Section 3.1.

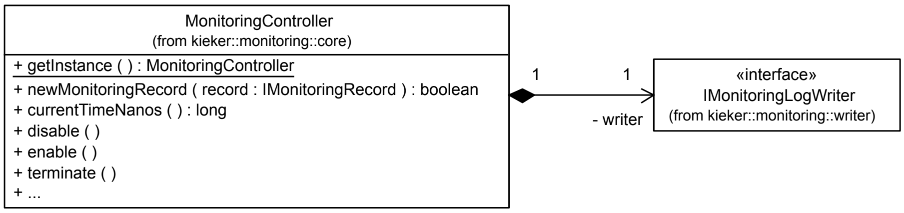


Figure 3.1: Class diagram of the `MonitoringController` (including selected methods)

3.3 Monitoring Records

Monitoring records are objects that contain the monitoring data, as mentioned in the previous chapters. Typically, an instance of a monitoring record is constructed in a monitoring probe (Section 3.4), passed to the monitoring controller (Section 3.2), serialized and deserialized by a monitoring writer (Section 3.5) and a monitoring reader (Section 4.2), and provided to the analyis plugins (Section 4.4) by the analysis controller (Section 4.1). Figure 1.2 illustrates this life cycle of a monitoring record.

In Chapter 2, we've already introduced and used the monitoring record type `OperationExecutionRecord`. Kieker allows to use custom monitoring record types. Corresponding classes must implement the interface `IMonitoringRecord` shown in Figure 3.2. The methods *initFromArray*, *toArray*, *getValueTypes* are used for serialization and deserialization of the monitoring data contained in the record. The method *setLoggingTimestamp* is used by the monitoring controller to store the date and time when a record is received by the controller. The method *getLoggingTimestamp* can be used during analysis to retrieve this value. Kieker.Monitoring provides the abstract class `AbstractMonitoringRecord` (Figure 3.2) which already implements the methods to maintain the logging timestamp.

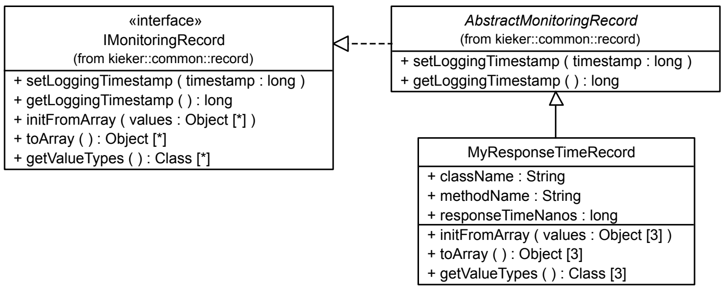


Figure 3.2: Class diagram with the `IMonitoringRecord` interface, the abstract class `AbstractMonitoringRecord`, and a custom monitoring record type `MyResponseTimeRecord`

Employing the abstract class for implementing your own monitoring record type, you need to:

1. Create a class that extends `AbstractMonitoringRecord` and
2. Override the methods *initFromArray*, *toArray*, *getValueTypes*.

The class `MyResponseTimeRecord`, shown in the class diagram in Figure 3.2 and in Listing 3.1, is an example of a custom monitoring record type that can be used to monitor response times of method executions.

```java
package bookstoreApplication;

import kieker.common.record.AbstractMonitoringRecord;

public class MyResponseTimeRecord extends AbstractMonitoringRecord {

    private static final long serialVersionUID = 1775L;
    private final static String NA_VAL = "N/A";

    /* Attributes storing the actual monitoring data: */
    public String className = MyResponseTimeRecord.NA_VAL;
    public String methodName = MyResponseTimeRecord.NA_VAL;
    public long responseTimeNanos = -1;

    @Override
    public final void initFromArray(final Object[] values) {
            this.className = (String) values[0];
            this.methodName = (String) values[1];
            this.responseTimeNanos = (Long) values[2];
    }

    @Override
    public final Object[] toArray() {
        return new Object[]{this.className, this.methodName, this.responseTimeNanos};
    }

    @Override
    public Class<?>[] getValueTypes() {
        return new Class[]{String.class, String.class, long.class};
    }
}
```

Listing 3.1: MyResponseTimeRecord.java

3.4 Monitoring Probes

The probes are responsible for collecting the monitoring data and passing this monitoring data to the monitoring controller. In Chapter 2.3, we have already demonstrated how to manually instrument a Java application. Listing 3.2 shows a similar manual monitoring probe which uses the monitoring record type `MyResponseTimeRecord` defined in the previous Section 3.3.

```
    { /* 1. Invoke catalog.getBook() and monitor response time */
      final long tin =
          Bookstore.MONITORING_CONTROLLER.getTimeSource().getTime();
      this.catalog.getBook(false);
      final long tout =
          Bookstore.MONITORING_CONTROLLER.getTimeSource().getTime();
      /* Create a new record and set values */
      final MyResponseTimeRecord e = new MyResponseTimeRecord();
      e.className = "mySimpleKiekerExample.bookstoreTracing.Catalog";
      e.methodName = "getBook(..)";
```

Listing 3.2: Excerpt from Bookstore.java

In order to avoid multiple calls to the *getInstance* method of the `MonitoringController` class, the singleton instance should be stored in a final static variable, as shown in Listing 3.3.

```
  private final CRM crm = new CRM(this.catalog);
  private final static IMonitoringController MONITORING_CONTROLLER =
```

Listing 3.3: Singleton instance of the monitoring controller stored in a final static variable (excerpt from Bookstore.java)

When manually instrumenting an application, the monitoring probe is implemented by mixing monitoring logic with business logic, which is often not desired since the resulting code is hardly maintainable. Many middleware technologies, such as Java EE Servlet [3], Spring [4], and Apache CXF [5] provide interception/AOP [1] interfaces which are well-suited to implement monitoring probes. AspectJ [7] allows to instrument Java applications without source code modifications. Chapter 5 describes the Kieker probes based on these technologies allowing to monitor trace information in distributed applications.

3.5 Monitoring Writers

Monitoring log writers serialize monitoring records to the monitoring log and must implement the interface `IMonitoringWriter`. The monitoring controller passes the received records to the writer by calling the method *newMonitoringRecord*. Writers can use the methods to serialize the record contents, as described in Section 3.3.

Figure 3.3 shows the monitoring writers already implemented in Kieker.Monitoring. The writers `AsyncFsWriter`, `SyncFsWriter`, `AsyncDbWriter`, and `SyncDbWriter` can be used to store monitoring records to filesystems and databases respectively. The variants with the prefix `Async` are implemented using asynchronous threads that decouple the I/O operations from the control flow of the instrumented application. The `AsyncFsWriter` is the default writer which has already been used in Section 2.3. Currently, the database writer only supports the record type `OperationExecutionRecord`.

The `AsyncJMSWriter` writes records to a JMS (Java Messaging Service [2]) queue. This allows to implement on-the-fly analysis in distributed systems, i.e., analysis while continuously receiving new monitoring data from an instrumented application potentially running on another machine. A brief description of how to use the `AsyncJMSWriter` can be found in Appendix G.

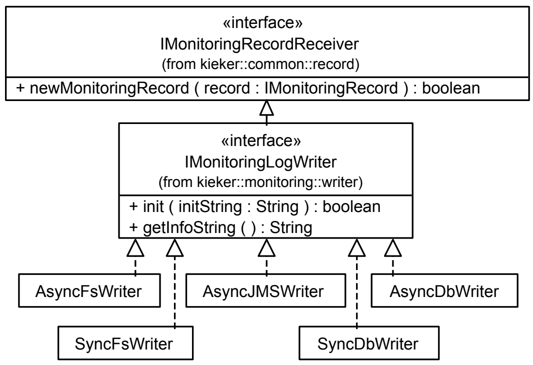


Figure 3.3: Interface `IMonitoringLogWriter` and the implementing classes

Listing 3.4 on page 25 shows a custom writer `MyPipeWriter` which uses a named pipe to write the given records into a buffer located in the memory. The source code of the class `MyPipe` is listed in Appendix C.1.
The monitoring writer to be used is selected and configured by the Kieker.Monitoring configuration properties (Section 3) *monitoringDataWriter* and *monitoringDataWriterInitString*. Listing 3.5 demonstrates how to use the custom writer `MyPipeWriter` defined above. In this example, the pipe name is passed as the property value *monitoringDataWriterInitString*.

```
monitoringDataWriter=bookstoreApplication.MyPipeWriter
monitoringDataWriterInitString =pipeName=somePipe
```

Note that we decided to use `Object` arrays as the data structure of the monitoring log in order to demonstrate the use of the *toArray* and *initFromArray* (in Section 4.2) methods. Alternatively, we could have used `IMonitoringRecord` as the data structure used by the pipe.

```
package bookstoreApplication;

import java.util.Properties;

import kieker.common.record.IMonitoringRecord;
import kieker.monitoring.core.configuration.Configuration;
import kieker.monitoring.writer.AbstractMonitoringWriter;

public class MyPipeWriter extends AbstractMonitoringWriter {
  private static final String PREFIX = MyPipeWriter.class.getName() + ".";
  private static final String PROPERTY_PIPE_NAME = MyPipeWriter.PREFIX
      + "pipeName";
  private volatile String pipeName;
  private volatile MyPipe pipe;

  public MyPipeWriter(final Configuration configuration) {
    super(configuration);
  }

  @Override
  public boolean newMonitoringRecord(final IMonitoringRecord record) {
    try {
      /* Just write the content of the record into the pipe. */
      this.pipe.put(new PipeData(record.getLoggingTimestamp(), record.toArray()));
    } catch (final InterruptedException e) {
      return false; // signal error
    }
    return true;
  }

  @Override
  protected Properties getDefaultProperties() {
    final Properties properties = new Properties(super.getDefaultProperties());
    properties.setProperty(MyPipeWriter.PROPERTY_PIPE_NAME, "myPipeName");
    return properties;
  }

  @Override
  protected void init() throws Exception {
    this.pipeName =
        this.configuration
            .getStringProperty(MyPipeWriter.PROPERTY_PIPE_NAME);
    this.pipe = MyNamedPipeManager.getInstance().acquirePipe(this.pipeName);
  }

  @Override
  public void terminate() { }
}
```

Listing 3.4: MyWriter.java

4 Kieker.Analysis Component

> ☞ The Java sources of this chapter can also be found in the `examples/userguide/ch3-4-custom-components/` directory of the binary release.

4.1 Analysis Controller

An analysis with Kieker.Analysis is set up and executed employing the class `AnalysisController`. Kieker.Analysis requires a monitoring reader (Section 4.2) and at least one monitoring record consumer plugin (Section 4.4). In addition to the monitoring record consumer plugin, other analysis plugins can be registered. Figure 4.1 shows the class diagram with the important Kieker.Analysis classes and their relationship.

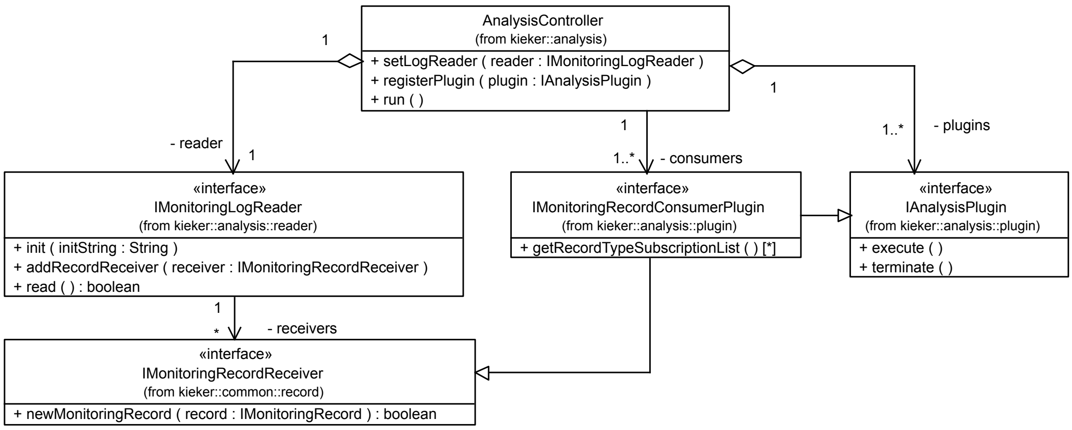


Figure 4.1: Class diagram showing important Kieker.Analysis classes and their relationship

Setting up and running an analysis with Kieker.Analysis requires the following steps to be performed, as described in Section 2.4 already:

1. Creating an instance of the `AnalysisController` class
2. Creating and registering the monitoring reader (*setLogReader*) as well as the monitoring record consumers and other analysis plugins (*registerPlugin*).
3. Starting the analysis instance (*run*).

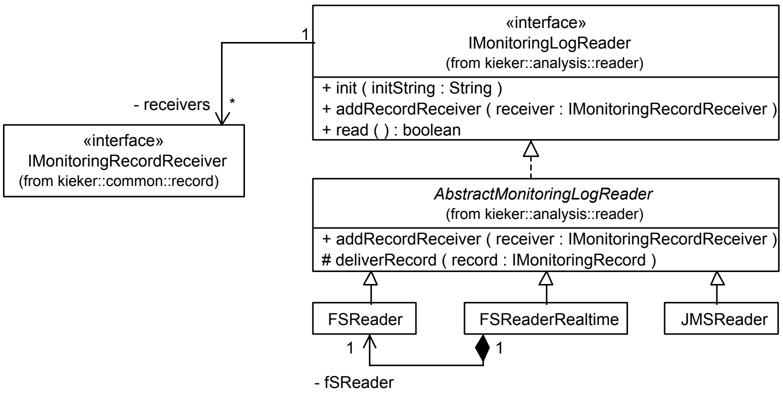


Figure 4.2: Interface `IMonitoringLogReader` and implementing classes

In the following Sections 4.2 and 4.4, we will create a custom monitoring reader `MyPipeReader` and a monitoring record consumer plugin `MyResponseTimeConsumer`. The following Listing 4.1 shows how to create and run an analysis with these custom components:

```
/* Start an analysis of the response times */
final AnalysisController analyisController = new AnalysisController();
final IMonitoringReader reader =
        new MyPipeReader("somePipe");
final IMonitoringRecordConsumerPlugin consumer =
        new MyResponseTimeConsumer();
analyisController.setLogReader(reader);
analyisController.registerPlugin(consumer);
```

Listing 4.1: Code snippet setting up and running a Kieker.Analysis instance (Starter.java)

On invocation of the *run* method, the `Analysis Controller` calls the *execute* method of all analysis plugins allowing them to initialize. Then, it starts the configured monitoring reader by calling its *read* method. Monitoring record consumers receive the monitoring records provided by the reader. As soon as the reader returns from the execution of its *read* method, the method *terminate* of each registered plugin is called by the `Analysis Controller`.

4.2 Monitoring Readers

The monitoring readers are the direct counterpart to the monitoring writers. While writers receive records and write them into files or other kinds of monitoring logs/streams, readers deserialize monitoring data and provide it as `IMonitoringRecord` instances.

There are already some readers implemented in Kieker as shown in the class diagram in Figure 4.2. The `FSReader` has already been used in Section 2.4. The `FSReaderRealtime` can be used to simulate continuous monitoring of a production system. It adds delays between the delivery of the monitoring records to its consumers according to the original delays reconstructed from the logging timestamps (Section 3.3). A brief description of how to use the `JMSReader` can be found in Appendix G.
The implementation of a custom reader is similar to implementating a monitoring writer. Custom reader should extend the class `AbstractKiekerMonitoringReader` which already provides an implementation of the observer pattern. By invoking the method *deliverRecord*, the delivery of records is then delegated to the super class.

Listing 4.2 on page 29 shows a simple reader which polls records from the named pipe introduced in the previous Chapter 3.

4.3 Analysis Plugins

Any analysis or visualization component used with Kieker.Analysis must implement the interface `IAnalysisPlugin` (Figure 4.1). As described in Section 4.1, the life-cycle of each registered plugin is controlled by the `Analysis Controller` instance employing the methods *execute* and *terminate*. Analysis plugins must implement these methods for initialization and termination.

The monitoring record consumer plugins described in the following Section 4.4, are special analysis plugins that receive the monitoring records provided by the monitoring log reader. Starting with these monitoring record plugins, analysis plugins can be connected in a pipe-and-filter style to implement more complex analyses. Kieker provides input and output port interface and implementing classes to implement such analyses. See the documentation of the classes `AbstractInputPort` and `OutputPort` for details. Kieker.TraceAnalysis is implemented based on this pattern.

4.4 Monitoring Record Consumer Plugins

As just mentioned, consumer plugins are special analysis plugins which receive the records provided by the monitoring log reader and implement analyses or visualizations based on these records. Consumer plugins must implement the interface `IMonitoringRecordConsumerPlugin` (see Figure 4.1). By implementing the *getRecordTypeSubscriptionList* method, a consumer plugin can specify the desired types of monitoring records to be received via the method *newMonitoringRecord*.

The custom consumer in Listing 4.3 on page 30 simply writes the content of the received response time records to the standard output stream.

```java
package bookstoreApplication;

import kieker.analysis.reader.AbstractMonitoringReader;
import kieker.analysis.util.PropertyMap;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class MyPipeReader extends AbstractMonitoringReader {

  private static final Log log = LogFactory.getLog(MyPipeReader.class);

    private static final String PROPERTY_PIPE_NAME = "pipeName";

    private volatile MyPipe pipe;

    public MyPipeReader () {}

    public MyPipeReader (final String pipeName) {
        this.init (MyPipeReader.PROPERTY_PIPE_NAME+"="+pipeName);
    }

    @Override
    public boolean init( final String initString ) throws IllegalArgumentException {
      try {
          final PropertyMap propertyMap = new PropertyMap(initString, "|", "=");
          final String pipeName = propertyMap.getProperty(MyPipeReader.
              PROPERTY_PIPE_NAME);
          this.pipe = MyNamedPipeManager.getInstance().acquirePipe(pipeName);
    } catch (final Exception exc) {
      MyPipeReader.log.error("Failed to parse initString '" + initString
          + "': " + exc.getMessage());
      return false;
    }
    return true;
    }

    @Override
    public boolean read() {
        try {
          PipeData data;
            /* Wait max. 4 seconds for the next data. */
            while ((data = this.pipe.poll(4)) != null) {
                /* Create new record, init from received array ... */
                final MyResponseTimeRecord record = new MyResponseTimeRecord();
                record.initFromArray(data.getRecordData());
                record.setLoggingTimestamp(data.getLoggingTimestamp());
                /* ... and delegate the task of delivering to the super class. */
                this.deliverRecord(record);
            }
        } catch (final InterruptedException e) {
            return false; // signal error
        }
        return true;
    }
}
```

Listing 4.2: MyPipeReader.java

```
package bookstoreApplication;

import java.util.Collection;

import kieker.analysis.plugin.IMonitoringRecordConsumerPlugin;
import kieker.common.record.IMonitoringRecord;

public class MyResponseTimeConsumer implements IMonitoringRecordConsumerPlugin {

    @Override
    public Collection<Class<? extends IMonitoringRecord>> getRecordTypeSubscriptionList()
        {
        return null;
    }

    @Override
    public boolean newMonitoringRecord(IMonitoringRecord record) {
        if (record instanceof MyResponseTimeRecord) {
            /* Write the content to the standard output stream. */
            MyResponseTimeRecord myRecord = (MyResponseTimeRecord) record;
            System.out.println("[Consumer] " + myRecord.getLoggingTimestamp()
                    + ": " + myRecord.className + ", " + myRecord.methodName
                    + ", " + myRecord.responseTimeNanos);
        }
        return true;
    }

    @Override
    public boolean execute() {
        return true;
    }

    @Override
    public void terminate(boolean error) {
    }
}
```

Listing 4.3: MyReponseTimeConsumer.java

5 Kieker.TraceAnalysis Tool

Kieker.TraceAnalysis implements the special feature of Kieker allowing to monitor, analyze and visualize (distributed) traces of method executions and corresponding timing information. For this purpose, it includes monitoring probes employing AspectJ [7], Java EE Servlet [3], Spring [4], and Apache CXF [5] technology. Moreover, it allows to reconstruct and visualize architectural models of the monitored systems, e.g., as sequence and dependency diagrams.

Section 2 already introduced parts of the monitoring record type `OperationExecutionRecord`. Kieker.TraceAnalysis uses this record type to represent monitored executions and associated trace and session information. Figure 5.1 shows a class diagram with all attributes of the record type `OperationExecutionRecord`. The attributes *className*, *operationName*, *tin*, and *tout* have been introduced before. The attributes *traceId* and *sessionId* are used to store trace and session information; *eoi* and *ess* contain control-flow information needed to reconstruct traces from monitoring data. For details on this, we refer to our technical report [**?**].

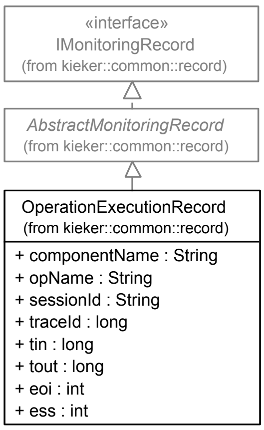


Figure 5.1: The class diagram of the operation execution record

Section 5.1 describes how to instrument Java applications for monitoring trace information. It presents the technology-specific probes provided by Kieker for this purpose—with a focus on AspectJ. Additional technology-specific probes can be implemented based on the existing probes. Section 5.2 presents the tool which can be used to analyze and visualize the recorded trace data. Examples for the available analysis and visualization outputs provided by Kieker.TraceAnalysis are presented in Section 5.3.

5.1 Monitoring Trace Information

The following Sections describe how to use the monitoring probes based on AspectJ (Section 5.1.1), the Java Servlet API (Section 5.1.2), the Spring Framework (Section 5.1.3), and Apache CXF (Section 5.1.4) provided by Kieker.

5.1.1 AspectJ-Based Instrumentation

AspectJ [7] allows to weave code into the byte code of Java applications and libraries without requiring manual modifications of the source code. Kieker includes the AspectJ-based monitoring probes `OperationExecutionAspectAnnotation`, `OperationExecutionAspectAnnotationServlet`, `OperationExecutionAspectFull`, and `OperationExecutionAspectFullServlet` which can be woven into Java applications at compile time and load time. These probes monitor method executions and corresponding trace and timing information. The probes with the postfix `Servlet` additionally store a session identifier within the `OperationExecutionRecord`. When the probes with name element `Annotation` are used, methods to be monitored must be annotated by the Kieker annotation `@OperationExecutionMonitoringProbe`. This section demonstrates how to use the AspectJ-based probes to monitor traces based on the Bookstore application from Chapter 2.

☞ The Java sources of the AspectJ example presented in this section can be found in the `examples/userguide/ch5-trace-monitoring-aspectj/` directory of the binary release.

```
examples/
└── userguide/
    └── ch5–trace-monitoring-aspectj/
        ├── build/ ........................ Directory for the Java class files
        │   └── META-INF/
        ├── META-INF/ ..................... Directory for the configuration files
        │   └── aop.xml
        ├── lib/ .......................... Directory for the needed libraries
        │   ├── kieker-1.3-dev.jar
        │   ├── commons-logging-1.1.1.jar
        │   └── aspectjweaver-1.6.11.jar
        └── src/ .......................... Directory for the source code files
            └── bookstoreTracing/
                ├── Bookstore.java
                ├── BookstoreStarter.java
                ├── Catalog.java
                └── CRM.java
```

Figure 5.2: The new directory structure of the Bookstore application

Figure 5.2 shows the directory used by the example of this section. The jar-file `aspectjweaver-1.6.11.jar` is included in the `lib/` directory of the Kieker binary release. Its `META-INF/` directory contains an example `aop.xml`. The library `aspectjweaver-1.6.11.jar` contains the *AspectJ weaver* which is registered with the JVM and weaves the monitoring instrumentation into the Java classes. It will be configured based on the configuration file `aop.xml`.

Once the necessary files have been copied to the example directory, the source code can be instrumented with the annotation `OperationExecutionMonitoringProbe`. Listing 5.1 shows how the annotation is used.

```java
package bookstoreTracing;

import kieker.monitoring.annotation.OperationExecutionMonitoringProbe;

public class Bookstore {

    private final Catalog catalog = new Catalog();
    private final CRM crm = new CRM(catalog);

    @OperationExecutionMonitoringProbe
    public void searchBook() {
        catalog.getBook(false);
        crm.getOffers();
    }
```

```
}
```

Listing 5.1: Bookstore.java

As a first example, each method of the Bookstore application will be annotated. The annotation can be used to instrumented all methods except for constructors.

The `aop.xml` file has to be modified to specify the classes to be considered for instrumentation by the AspectJ weaver. Listing 5.2 shows the modified configuration file.

```
1  <!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "http://www.aspectj.org/dtd/
       aspectj_1_5_0.dtd">
2
3  <aspectj>
4      <weaver options="">
5          <include within="bookstoreTracing..*"/>
6      </weaver>
7
8      <aspects>
9        <aspect name="kieker.monitoring.probe.aspectJ.operationExecution.
             OperationExecutionAspectAnnotation"/>
10     </aspects>
11 </aspectj>
```

Listing 5.2: aop.xml

Line 5 tells the AspectJ weaver to consider all classes inside the package `bookstore-Tracing`. AspectJ allows to use wild-cards for the definition of classes to include—e.g., <include within="bookstoreTracing.Bookstore*"/> to weave all classes in this package with the prefix `Bookstore`.

Line 9 specifies the aspect to be woven into the classes. In this case, the Kieker probe `OperationExecutionAspectAnnotation` is used. It requires that method intended to be instrumented are annotated by @OperationExecutionMonitoringProbe, as mentioned before.

Listings 5.4 and 5.3 show how to compile and run the annotated Bookstore application. The `aop.xml` must be located in a `META-INF/` directory in the classpath—in this case the `build/` directory. The AspectJ weaver has to be loaded as a so-called Java-agent. It weaves the monitoring aspect into the byte code of the Bookstore application.

```
▷ mkdir build
▷ mkdir build/META-INF
▷ javac src/bookstoreTracing/*.java
        -d build/
        -classpath lib/kieker-1.3-dev.jar:lib/commons-logging-1.1.1.jar

▷ cp META-INF/aop.xml build/META-INF/aop.xml

▷ java -javaagent:lib/aspectjweaver-1.6.11.jar
       -classpath build/:lib/kieker-1.3-dev.jar:lib/commons-logging-1.1.1.jar
        bookstoreTracing.BookstoreStarter
```

Listing 5.3: Commands to compile and run the Bookstore under UNIX-like systems

```
▷ mkdir build
▷ mkdir build\META−INF
▷ javac src\bookstoreTracing\∗.java
        −d build\
        −classpath lib\kieker-1.3-dev.jar; lib \commons-logging-1.1.1.jar

▷ copy META−INF\aop.xml build\META−INF\

▷ java −javaagent:lib\aspectjweaver-1.6.11.jar
       −classpath build\; lib /kieker-1.3-dev.jar; lib \commons-logging-1.1.1.jar
        bookstoreTracing.BookstoreStarter
```

Listing 5.4: Commands to compile and run the annotated Bookstore under Windows

After a complete run of the application, the monitoring files should appear in the same way as mentioned in Section 2.3 including the additional trace information. An example log of a complete run can be found in Appendix D.2.

Instrumentation without annoations AspectJ-based instrumentation without using annotations is quite simple. It is only necessary to modify the file `aop.xml`, as shown in Listing 5.5.

```
1  <!DOCTYPE aspectj PUBLIC "−//AspectJ//DTD//EN" "http://www.aspectj.org/dtd/
       aspectj_1_5_0.dtd">
2
3  <aspectj>
4      <weaver options="">
5          <include within="bookstoreTracing.BookstoreStarter"/>
6      </weaver>
7
8      <aspects>
9        <aspect name="kieker.monitoring.probe.aspectJ.executions.OperationExecutionAspectFull
             "/>
10     </aspects>
11 </aspectj>
```

Listing 5.5: aop.xml

The alternative aspect `OperationExecutionAspectFull` is being activated in line 9. As indicated by its name, this aspect makes sure that every method within the included classes/packages will be instrumented and monitored. Listing 5.5 demonstrates how to limit the instrumented methods to those of the class `BookstoreStarter`.

The commands shown in the Listings 5.4 and 5.3 can again be used to compile and execute the example. Note that the annotations within the source code have no effect when using this aspect.

When using a custom aspect, it can be necessary to specify its class-name in the include directives of the aop.xml.

5.1.2 Servlet Filters

The Java Servlet API [3] includes the `javax.servlet.Filter` interface. It can be used to implement interceptors for incoming HTTP requests. Kieker includes the probes `OperationExecutionRegistrationFilter` and `OperationExecutionRegistrationAndLoggingFilter` which implement the `javax.servlet.Filter` interface. Both initialize the session and trace information for incoming requests. The latter additionally creates an `OperationExecutionRecord` for each invocation of the filter and passes it to the `MonitoringController`.

Listing 5.6 demonstrates how to integrate the `OperationExecutionRegistrationAndLoggingFilter` in the `web.xml` file of a web application.

```
<filter>
 <filter-name>sessionRegistrationFilter</filter-name>
 <filter-class>kieker.monitoring.probe.servlet.OperationExecutionRegistrationAndLoggingFilter</
     filter-class>
</filter>
<filter-mapping>
 <filter-name>sessionRegistrationFilter</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

Listing 5.6: `OperationExecutionRegistrationAndLoggingFilter` in a `web.xml` file

The Java EE Servlet container example described in Appendix employs the the `OperationExecutionRegistrationAndLoggingFilter`.

5.1.3 Spring

The Spring framework [4] provides interfaces for intercepting Spring services and web requests. Kieker includes the probes `OperationExecutionMethodInvocationInterceptor` and `OperationExecutionWebRequestRegistrationInterceptor`. The `OperationExecutionMethodInvocationInterceptor` is similar to the AspectJ-based probes described in the previous section and monitors method executions as well as corresponding trace and session information. The `OperationExecutionWebRequestRegistrationInterceptor` intercepts incoming Web requests and initializes the trace and session data for this trace.

See the Spring documentation for instructions how to add the interceptors to the server configuration.

5.1.4 CXF SOAP Interceptors

The Apache CXF framework [5] allows to implement and interceptors for service calls, for example, based on the SOAP web service protocol. Kieker includes the probes `OperationExecutionSOAPRequestOutInterceptor`, `OperationExecutionSOAPRequestInInterceptor`, `OperationExecutionSOAPResponseOutInterceptor`, and `OperationExecutionSOAPResponseInInterceptor` which can be used to monitor SOAP-based web service calls. Session and trace information is written to and read from the SOAP header of service requests and responses allowing to monitor distributed traces. See the CXF documentation for instructions how to add the interceptors to the server configuration.

5.2 Trace Analysis and Visualization

Monitoring data including trace information can be analyzed and visualized with the Kieker.TraceAnalysis tool which is included in the Kieker binary as well.

> STOP
>
> In order to use this tool, it is necessary to install two other programs:
>
> 1. **Graphviz** A graph visualization software which can be downloaded from `http://www.graphviz.org/`.
> 2. **GNU PlotUtils** A set of tools for generating 2D plot graphics which can be downloaded from `http://www.gnu.org/software/plotutils/` (for Linux) and from `http://gnuwin32.sourceforge.net/packages/plotutils.htm` (for Windows).
>
> Under Windows it is recommended to add the `bin/` directories of both tools to the "path" environment variable.

Once both have been installed, the Kieker.TraceAnalysis tool can be used. It can be accessed via the wrapper-script `trace-analysis.sh` or `trace-analysis.bat` (Windows) in the `bin/` directory. Non-parameterized calls of the scripts print all possible options on the screen, as listed in Appendix A.3.

The commands shown in Listings 5.7 and 5.8 generate a sequence diagram as well as a call tree to an existing directory named `out/`. The monitoring data is assumed to be located in the directory `/tmp/kieker-20110428-142829399-UTC-Kaapstad-KIEKER/` or `%temp%\kieker-20100813-121041532-UTC-virus-KIEKER` under Windows.

```
▷ ./trace-analysis.sh --inputdirs /tmp/kieker-20110428-142829399-UTC-Kaapstad-KIEKER
                      --outputdir out/
                      --plot-Allocation-Sequence-Diagrams
                      --plot-Call-Trees
```

Listing 5.7: Commands to produce the diagrams under UNIX-like systems

```
▷ ./trace-analysis.bat --inputdirs %temp%\kieker-20100813-121041532-UTC-virus-KIEKER
                       --outputdir out\
                       --plot-Allocation-Sequence-Diagrams
                       --plot-Call-Trees
```

Listing 5.8: Commands to produce the diagrams under Windows

The resulting contents of the `out/` directory should be similar to the following tree:

```
out/
├── allocationSequenceDiagram-6120391893596504065.pic
├── callTree-6120391893596504065.dot
└── system-entities.html
```

The `.pic` and `.dot` files can be converted into other formats, such as `.pdf`, by using the *Graphviz* and *PlotUtils* tools `dot` and `pic2plot`. The following Listing 5.9 demonstrates this.

```
▷ dot callTree−6120391893596504065.dot -Tpng -o callTree.png
▷ pic2plot allocationSequenceDiagram−6120391893596504065.pic -Tpng > sequenceDiagram.png
```

Listing 5.9: Commands to convert the diagrams

☞ The scripts `dotPic-fileConverter.sh` and `dotPic-fileConverter.bat` convert all `.pic` and `.dot` in a specified directory. See Appendix A.4 for details.

Examples of all available visualization are presented in the following Section 5.3.

5.3 Example Kieker.TraceAnalysis Outputs

The examples presented in this section were generated based on the monitoring data which can be found in the directory `examples/userguide/ch5-trace-monitoring-aspectj/testdata/tpmon-20100830-082225522-UTC/`. It consists of 1635 traces of the Bookstore application with AspectJ-based instrumentation, as described in Section 5.1.1. In order to illustrate the visualization of distributed traces, the hostname of the `Catalog`'s method *getBook* was probabilistically changed to a second hostname. For a more detailed description of the underlying formalisms, we refer to our technical report [**?**].

5.3.1 Textual Trace and Equivalence Class Representations

Execution Traces

Textual execution trace representations of valid/invalid traces are written to an output file using the command-line options `--print-Execution-Traces` and `--print-invalid-Execution-Traces`. Listing 5.10 shows the execution trace representation for the valid trace . . . 6129.

```
TraceId 6488138950668976129 (minTin=1283156498770302094 (Mo, 30 Aug 2010 08:21:38 +0000 (UTC));
      maxTout=1283156498820012272 (Mo, 30 Aug 2010 08:21:38 +0000 (UTC)); maxEss=2):
<6488138950668976129[0,0] 1283156498770302094−1283156498820012272 SRV0::@3:bookstoreTracing.
      Bookstore.searchBook N/A>
<6488138950668976129[1,1] 1283156498770900902−1283156498773404399 SRV1::@1:bookstoreTracing.
      Catalog.getBook N/A>
<6488138950668976129[2,1] 1283156498817823953−1283156498820007367 SRV0::@2:bookstoreTracing.CRM.
      getOffers N/A>
<6488138950668976129[3,2] 1283156498817855493−1283156498819999771 SRV1::@1:bookstoreTracing.
      Catalog.getBook N/A>
```

Listing 5.10: Textual output of trace 6488138950668976129's execution trace representation

Message Traces

Textual message trace representations of valid traces are written to an output file using the command-line option `--print-Message-Traces`. Listing 5.11 shows the message trace representation for the valid trace . . . 6129.

```
Trace 6488138950668976129:
<SYNC−CALL 1283156498770302094 $ −−> 6488138950668976129[0,0]
      1283156498770302094−1283156498820012272 SRV0::@3:bookstoreTracing.Bookstore.searchBook N/A>
<SYNC−CALL 1283156498770900902 6488138950668976129[0,0]
      1283156498770302094−1283156498820012272 SRV0::@3:bookstoreTracing.Bookstore.searchBook N/A
      −−> 6488138950668976129[1,1] 1283156498770900902−1283156498773404399 SRV1::@1:
      bookstoreTracing.Catalog.getBook N/A>
<SYNC−RPLY 1283156498773404399 6488138950668976129[1,1]
      1283156498770900902−1283156498773404399 SRV1::@1:bookstoreTracing.Catalog.getBook N/A −−>
      6488138950668976129[0,0] 1283156498770302094−1283156498820012272 SRV0::@3:bookstoreTracing.
      Bookstore.searchBook N/A>
```

```
<SYNC-CALL 1283156498817823953 6488138950668976129[0,0]
    1283156498770302094-1283156498820012272 SRV0::@3:bookstoreTracing.Bookstore.searchBook N/A
    --> 6488138950668976129[2,1] 1283156498817823953-1283156498820007367 SRV0::@2:
    bookstoreTracing.CRM.getOffers N/A>
<SYNC-CALL 1283156498817855493 6488138950668976129[2,1]
    1283156498817823953-1283156498820007367 SRV0::@2:bookstoreTracing.CRM.getOffers N/A -->
    6488138950668976129[3,2] 1283156498817855493-1283156498819999771 SRV1::@1:bookstoreTracing.
    Catalog.getBook N/A>
<SYNC-RPLY 1283156498819999771 6488138950668976129[3,2]
    1283156498817855493-1283156498819999771 SRV1::@1:bookstoreTracing.Catalog.getBook N/A -->
    6488138950668976129[2,1] 1283156498817823953-1283156498820007367 SRV0::@2:bookstoreTracing.
    CRM.getOffers N/A>
<SYNC-RPLY 1283156498820007367 6488138950668976129[2,1]
    1283156498817823953-1283156498820007367 SRV0::@2:bookstoreTracing.CRM.getOffers N/A -->
    6488138950668976129[0,0] 1283156498770302094-1283156498820012272 SRV0::@3:bookstoreTracing.
    Bookstore.searchBook N/A>
<SYNC-RPLY 1283156498820012272 6488138950668976129[0,0]
    1283156498770302094-1283156498820012272 SRV0::@3:bookstoreTracing.Bookstore.searchBook N/A
    --> $>
```

Listing 5.11: Textual output of trace 6488138950668976129's message trace representation

Trace Equivalence Classes

Deployment/assembly-level trace equivalence classes are computed and written to output files using the command-line options `--print-Deployment-Equivalence-Classes` and `--print-Assembly-Equivalence-Classes`. Listings 5.12 and 5.13 show the output generated for the monitoring data used in this section.

```
Class 0 ; cardinality : 187; # executions: 4; representative : 6488138950668976141; max. stack depth: 2
Class 1 ; cardinality : 706; # executions: 4; representative : 6488138950668976129; max. stack depth: 2
Class 2 ; cardinality : 386; # executions: 4; representative : 6488138950668976130; max. stack depth: 2
Class 3 ; cardinality : 356; # executions: 4; representative : 6488138950668976131; max. stack depth: 2
```

Listing 5.12: Textual output of information on the *deployment-level* trace equivalence classes

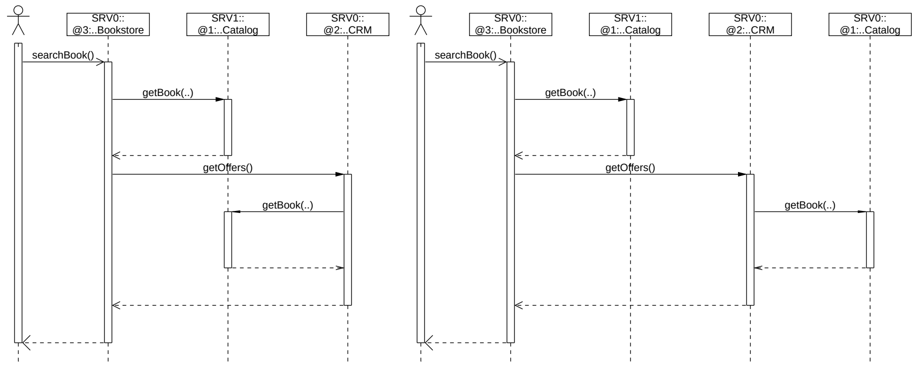
```
Class 0 ; cardinality : 1635; # executions: 4; representative : 6488138950668976129; max. stack depth: 2
```

Listing 5.13: Textual output of information on the *assembly-level* trace equivalence class

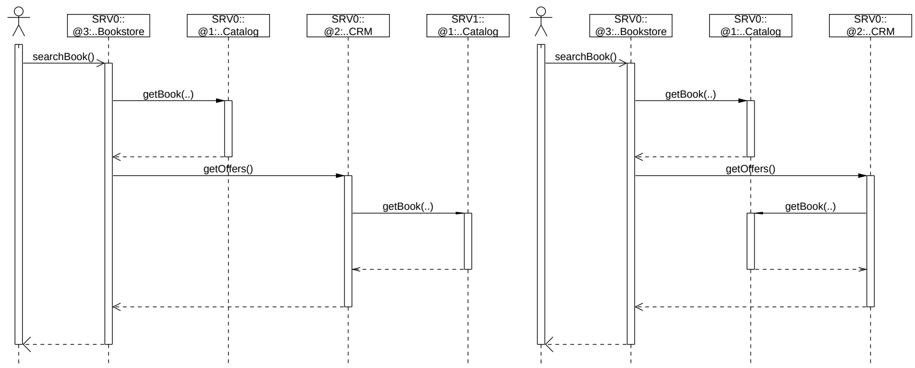
5.3.2 Sequence Diagrams

Deployment-Level Sequence Diagrams

Deployment-level sequence diagrams are generated using the command-line option `--plot-Deployment-Sequence-Diagrams`. Figures 5.3(a)–5.3(d) show these sequence diagrams for each deployment-level trace equivalence representative (Section 5.3.1).



(a) Trace . . . 6129 (b) Trace . . . 6130



(c) Trace . . . 6131 (d) Trace . . . 6141

Figure 5.3: *Deployment-level* sequence diagrams of the trace equivalence class representatives (Listing 5.13)

Assembly-Level Sequence Diagrams

Assembly-level sequence diagrams are generated using the command-line option `--plot-Assembly-Sequence-Diagrams`. Figure 5.4 show the sequence diagram for the assembly-level trace equivalence representative (Section 5.3.1).

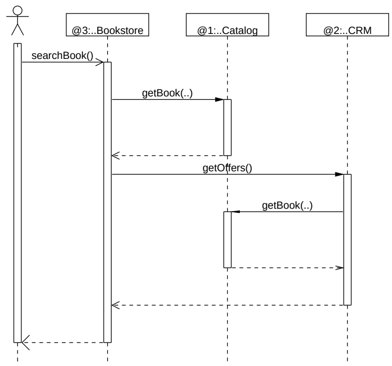
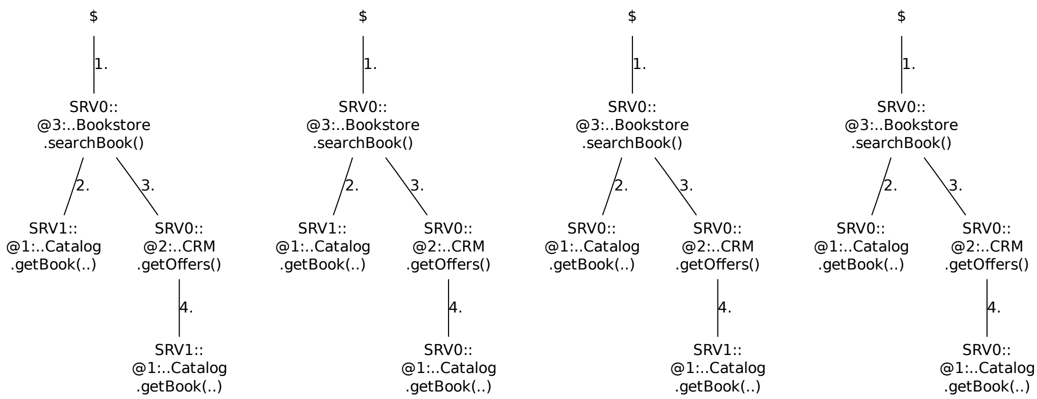


Figure 5.4: *Assembly-level* sequence diagram of trace . . . 6129

5.3.3 Call Trees

Trace Call Trees

Trace call trees are generated using the command-line option `--plotCallTrees`. Figures 5.5(a)–5.5(d) show these call trees for each deployment-level trace equivalence representative (Section 5.3.1).

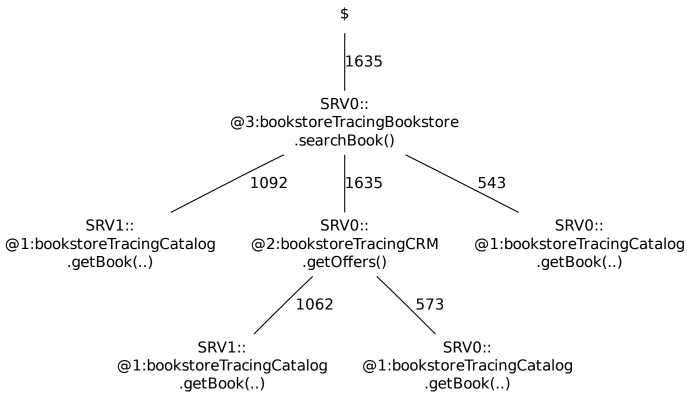


(a) Trace . . . 6129 (b) Trace . . . 6130 (c) Trace . . . 6131 (d) Trace . . . 6141

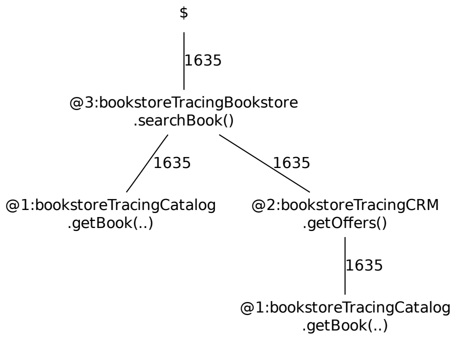
Figure 5.5: Calls trees of the trace equivalence class representatives (Listing 5.13)

Aggregated Call Trees

Aggregated deployment/assembly-level call trees are generated using the command-line options `--plot-Aggregated-Deployment-Call-Tree` and `--plot-Aggregated-Assembly-Call-Tree`. Figures 5.6(a) and 5.6(b) show these aggregated call trees for the traces contained in the monitoring data used in this section.



(a) deployment-level



(b) assembly-level

Figure 5.6: Aggregated call trees generated from the 1635 traces

5.3.4 Dependency Graphs

Container Dependency Graphs

A container dependency graph is generated using the command-line option `--plot-Container-Dependency-Graph`. Figure 5.7 shows the container dependency graph for the monitoring data used in this section.

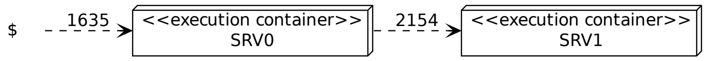
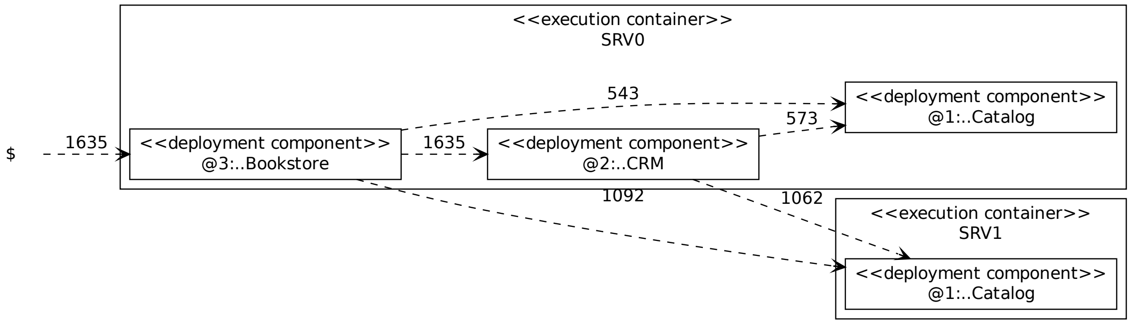


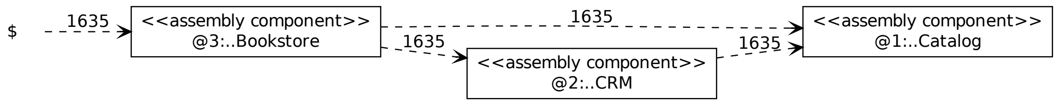
Figure 5.7: Container dependency graph

Component Dependency Graphs

Deployment/assembly-level component dependency graphs are generated using the command-line options `--plot-Deployment-Component-Dependency-Graph` and `--plot-Assembly-Component-Dependency-Graph`. Figures 5.8(a) and 5.8(b) show the component dependency graphs for the monitoring data used in this section.



(a) deployment-level

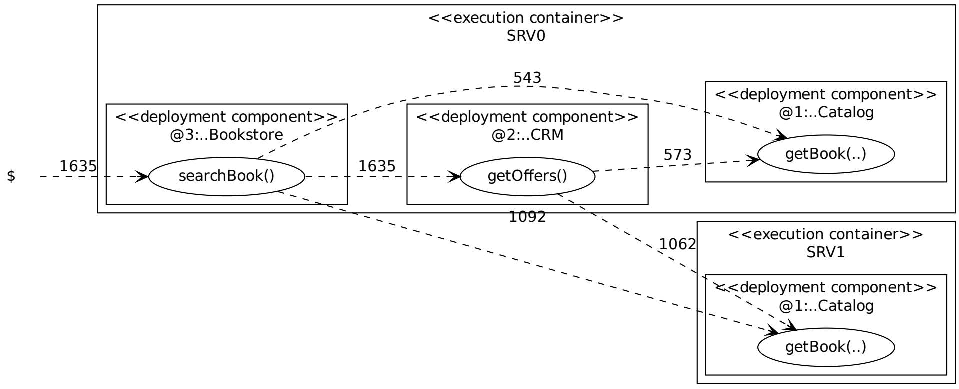


(b) assembly-level

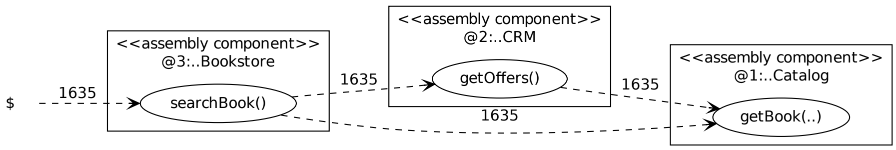
Figure 5.8: Component dependency graphs

Operation Dependency Graphs

Deployment/assembly-level operation dependency graphs are generated using the command-line options `--plot-Deployment-Operation-Dependency-Graph` and `--plot-Assembly-Operation-Dependency-Graph`. Figures 5.9(a) and 5.9(b) show the operation dependency graphs for the monitoring data used in this section.



(a) deployment-level



(b) assembly-level

Figure 5.9: Operation dependency graphs

5.3.5 HTML Output of the System Model

Kieker.TraceAnalysis writes an HTML representation of the system model reconstructed from the trace data to a file `system-entities.html`. Figure 5.10 shows a screenshot of this file rendered by a web browser.

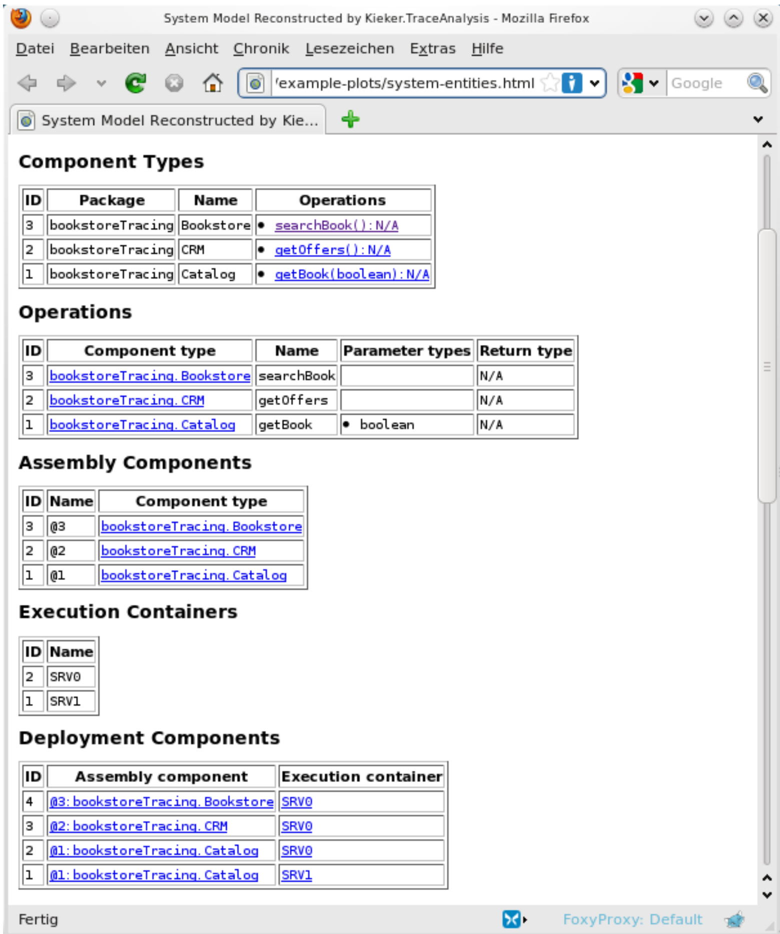


Figure 5.10: HTML output of the system model reconstructed from the traces

A Wrapper scripts

The `bin/` directory of Kieker's binary release contains some `.sh` and `.bat` scripts to invoke tools included in `kieker-1.3-dev.jar`. The following sections give a short description of their functionality and list their usage outputs as printed to the standard output stream when called without command-line parameters. In addition to the standard output stream, the file `kieker.log` is used for logging output during execution.

A.1 Script `convertLoggingTimestamp.sh|bat`

The script converts Kieker.Monitoring logging timestamps, representing the number of nanoseconds since 1 Jan 1970 00:00 UTC, to a human-readable textual representation in the UTC and local timezones.

Main-class: `kieker.tools.loggingTimestampConverter.LoggingTimestampConverterTool`

Usage

```
usage: kieker.tools.loggingTimestampConverter.LoggingTimestampConverterTool -t
       <timestamp1 ... timestampN>
 -t,--timestamps <timestamp1 ... timestampN> List of timestamps (UTC timezone)
                                              to convert
```

Example

The following listing shows the command to convert two logging timestamps as well as the resulting output.

```
▷ bin/convertLoggingTimestamp.sh --timestamps 1283156545581511026 1283156546127117246
1283156545581511026: Mo, 30 Aug 2010 08:22:25 +0000 (UTC) (Mo, 30 Aug 2010 10:22:25 +0200 (local
    time))
1283156546127117246: Mo, 30 Aug 2010 08:22:26 +0000 (UTC) (Mo, 30 Aug 2010 10:22:26 +0200 (local
    time))
```

A.2 Script `logReplay.sh|bat`

Replays filesystem monitoring logs created by Kieker.Monitoring. Example applications are:

- Merging multiple directories containing monitoring data into a single output directory.
- Importing a filesystem monitoring log to another monitoring log, e.g., a database. Therefore, an appropriate Kieker.Monitoring configuration file must be passed to the script (see Section 3.1).
- Replaying a recorded filesystem monitoring log in real-time in order to simulate incoming monitoring data from a running system, e.g., via JMS (see also Appendix G).

Main-class: `kieker.tools.logReplayer.FilesystemLogReplayerStarter`

Usage

```
usage: kieker.tools.logReplayer.FilesystemLogReplayerStarter
 -i,--inputdirs <dir1 ... dirN>               Log directories to read data
                                              from
 -k,--keep-logging-timestamps <true|false>    Replay the original logging
                                              timestamps (defaults to
                                              true)?)
 -n,--realtime-worker-threads <num>           Number of worker threads used
                                              in realtime mode (defaults to
                                              1).
 -r,--realtime <true|false>                   Replay log data in realtime?
```

Example

The following command replays the monitoring testdata included in the binary release to another directory:

```
▷ bin/logReplay.sh
  --inputdirs examples/userguide/ch5-trace-monitoring-aspectj/testdata/tpmon-20100830-082225522-UTC
  --keep-logging-timestamps true
  --realtime false
```

A.3 Script `trace-analysis.sh|bat`

Calls Kieker.TraceAnalysis to analyze and visualize monitored trace data, as described in Chapter 5.

Main-class: `kieker.tools.traceAnalysis.TraceAnalysisTool`

Usage

```
usage: kieker . tools . traceAnalysis . TraceAnalysisTool −i <dir1 ... dirN> −o <dir>
          [−p <prefix>] [−−plot−Deployment−Sequence−Diagrams]
          [−−plot−Assembly−Sequence−Diagrams]
          [−−plot−Deployment−Component−Dependency−Graph]
          [−−plot−Assembly−Component−Dependency−Graph]
          [−−plot−Container−Dependency−Graph]
          [−−plot−Deployment−Operation−Dependency−Graph]
          [−−plot−Assembly−Operation−Dependency−Graph]
          [−−plot−Aggregated−Deployment−Call−Tree]
          [−−plot−Aggregated−Assembly−Call−Tree] [−−plot−Call−Trees]
          [−−print−Message−Traces] [−−print−Execution−Traces]
          [−−print−invalid−Execution−Traces]
          [−−print−Deployment−Equivalence−Classes]
          [−−print−Assembly−Equivalence−Classes] [−−select−traces <id0 ... idn>]
          [−−ignore−invalid−traces] [−−max−trace−duration <duration in ms>]
          [−−ignore−executions−before−date <yyyyMMdd−HHmmss>]
          [−−ignore−executions−after−date <yyyyMMdd−HHmmss>] [−−short−labels]
 −i,−−inputdirs <dir1 ... dirN>                    Log directories to read
                                                   data from
 −o,−−outputdir <dir>                              Directory for the
                                                   generated file (s)
 −p,−−output−filename−prefix <prefix>              Prefix for output
                                                   filenames
    −−plot−Deployment−Sequence−Diagrams            Generate and store
                                                   deployment−level
                                                   sequence diagrams (. pic
                                                    files )
    −−plot−Assembly−Sequence−Diagrams              Generate and store
                                                   assembly−level sequence
                                                   diagrams (. pic files )
    −−plot−Deployment−Component−Dependency−Graph Generate and store a
                                                   deployment−level
                                                   component dependency
                                                   graph (.dot file )
    −−plot−Assembly−Component−Dependency−Graph     Generate and store an
                                                   assembly−level component
                                                   dependency graph (.dot
                                                    file )
    −−plot−Container−Dependency−Graph              Generate and store a
                                                   container dependency
                                                   graph (.dot file )
    −−plot−Deployment−Operation−Dependency−Graph   Generate and store a
                                                   deployment−level
                                                   operation dependency
                                                   graph (.dot file )
    −−plot−Assembly−Operation−Dependency−Graph     Generate and store an
                                                   assembly−level operation
                                                   dependency graph (.dot
                                                    file )
    −−plot−Aggregated−Deployment−Call−Tree         Generate and store an
                                                   aggregated
                                                   deployment−level call
                                                   tree (.dot files )
    −−plot−Aggregated−Assembly−Call−Tree           Generate and store an
                                                   aggregated
                                                   assembly−level call tree
```

```
                                                  (.dot  files )
  --plot-Call-Trees                               Generate and store  call
                                                  trees  for the  selected
                                                  traces  (.dot  files )
  --print-Message-Traces                          Save message trace
                                                  representations  of  valid
                                                  traces  (.txt  files )
  --print-Execution-Traces                        Save execution  trace
                                                  representations  of  valid
                                                  traces  (.txt  files )
  --print-invalid-Execution-Traces                Save a  execution  trace
                                                  representations  of
                                                  invalid  trace  artifacts
                                                  (.txt  files )
  --print-Deployment-Equivalence-Classes          Output an overview about
                                                  the deployment-level
                                                  trace  equivalence
                                                  classes
  --print-Assembly-Equivalence-Classes            Output an overview about
                                                  the assembly-level  trace
                                                  equivalence  classes
  --select-traces <id0 ... idn>                   Consider  only  the  traces
                                                  identified  by the  list
                                                  of  trace  IDs.  Defaults
                                                  to  all  traces.
  --ignore-invalid-traces                         If  selected,  the
                                                  execution  aborts  on the
                                                  occurence  of  an  invalid
                                                  trace.
  --max-trace-duration <duration in ms>           Threshold (in
                                                  milliseconds)  after
                                                  which an incomplete
                                                  trace  becomes invalid.
                                                  Defaults  to  infinity .
  --ignore-executions-before-date <yyyyMMdd-HHmmss> Executions starting
                                                  before  this  date (UTC
                                                  timezone)  are  ignored.
  --ignore-executions-after-date <yyyyMMdd-HHmmss> Executions ending after
                                                  this  date (UTC timezone)
                                                  are  ignored.
  --short-labels                                  If  selected,  abbreviated
                                                  labels  (e.g.,  package
                                                  names) are used  in  the
                                                  visualizations .
```

Example

The following commands generate a deployment-level operation dependency graph and convert it to pdf format:

```
▷ bin/trace-analysis.sh
  --inputdirs examples/userguide/ch5–trace-monitoring-aspectj/testdata/tpmon-20100830-082225522-UTC
  --outputdir .
  --plot-Deployment-Operation-Dependency-Graph
▷ dot -T pdf  deploymentOperationDependencyGraph.dot > deploymentOperationDependencyGraph.pdf
```

Additional examples can be found in Chapter 5.

A.4 Script `dotPic-fileConverter.sh|bat`

Converts each `.dot` and `.pic` file, e.g., diagrams generated by Kieker.TraceAnalysis (Section 5), located in a directory into desired grahpic output formats. This scripts simply calls the *Graphviz* and *PlotUtils* tools `dot` and `pic2plot`.

Usage

```
Usage: dotPic-fileConverter.sh <output-directory> <file-type-1 ... file-type-N>
```

Example

The following command converts each `.dot` and `.pic` file located in the directory `out/` to files in `.pdf` and `.png` format:

```
▷ bin/dotPic-fileConverter.sh out/ pdf png
```

B Kieker.Monitoring Default Configuration

This is the file `kieker.monitoring.properties` from the binary release and constitutes Kieker.Monitoring's default configuration. Section 3.1 describes how to use a custom configuration.

```
## In order to use a custom Kieker.Monitoring configuration, create a copy of
## this file and modify it according to your needs.
##
## The location of the file is passed to Kieker.Monitoring via the JVM parameter
## kieker.monitoring.configuration. For example, with a configuration file named
## my.kieker.monitoring.properties in the folder META-INF you would pass this location
## to the JVM when starting your application:
##
## java -Dkieker.monitoring.configuration=META-INF/my.kieker.monitoring.properties [...]
##
## If no configuration file is passed, Kieker tries to use a configuration file in
## META-INF/kieker.monitoring.properties
## If this also fails, a default configuration is being used according to the values in
## this default file.

# The name of the Kieker instance.
kieker.monitoring.name=KIEKER

# The name of the VM running Kieker. If empty the name will be determined
# automatically, else it will be set to the given value.
kieker.monitoring.hostname=

# The initial ID associated with all experiments.
kieker.monitoring.initialExperimentId=1

# Whether the MonitoringController will be available as an MBean.
kieker.monitoring.MBean=false

# These two properties are only evaluated if the MBean is activated.
# They define the ObjectName used to access the MBean (usually you
# don't have to change them).
kieker.monitoring.MBean.domain=kieker.monitoring
kieker.monitoring.MBean.name=MonitoringController

# Whether Kieker runs in replay or realtime mode.
# You usually don't want to change this value.
kieker.monitoring.replayMode=false
```

```
# Enable/disable monitoring after startup (true|false; default: true)
# If monitoring is disabled, the MonitoringController simply pauses.
# Furthermore, probes should stop collecting new data and monitoring
# writers stop should stop writing existing data.
kieker.monitoring.enabled=true

# The Timer used by Kieker. You usually don't want to change the value.
kieker.monitoring.timer=kieker.monitoring.timer.DefaultSystemTimer
# You can specify additional parameters send to the Timer, e.g.
#kieker.monitoring.timer.DefaultSystemTimer.KEY=VALUE

# The size of the thread pool used to execute registered periodic sensor jobs.
kieker.monitoring.periodicSensorsExecutorPoolSize=1

# Enables/disable the automatic assignment of each record's logging timestamp
# (true|false; default: true)
kieker.monitoring.setLoggingTimestamp=true

###########################
####### WRITER #######
###########################
## Selection of monitoring data writer (classname)
## The value must be a fully-qualified classname of a class implementing
## kieker.monitoring.IMonitoringWriter and providing a constructor that
## accepts an IMonitoringController and a single Configuration.
kieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncFsWriter

#####
#kieker.monitoring.writer=kieker.monitoring.writer.DummyWriter
#
## Configuration Properties of the DummyWriter
kieker.monitoring.writer.DummyWriter.key=value


#####
#kieker.monitoring.writer=kieker.monitoring.writer.filesystem.SyncFsWriter
#
## In order to use the default temporary directory, set the property value of
## storeInJavaIoTmpdir to true.
kieker.monitoring.writer.filesystem.SyncFsWriter.storeInJavaIoTmpdir=true
#
## In order to use a custom directory, set storeInJavaIoTmpdir=false
## and set customStoragePath as desired. Examples:
## /var/kieker or "C:\KiekerData" (ensure the folder exists).
kieker.monitoring.writer.filesystem.SyncFsWriter.customStoragePath=


#####
#kieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncFsWriter
```

```
#
## In order to use the default temporary directory, set the property value of
## storeInJavaIoTmpdir to true.
kieker.monitoring.writer.filesystem.AsyncFsWriter.storeInJavaIoTmpdir=true
#
## In order to use a custom directory, set storeInJavaIoTmpdir=false
## and set customStoragePath as desired. Examples:
## /var/kieker or "C:\KiekerData" (ensure the folder exists).
kieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath=
#
## Asynchronous writers need to store monitoring records in an internal buffer.
## This parameter defines its capacity in terms of the number of records.
kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueSize=10000
#
## Behavior of the asynchronous writer when the internal queue is full:
## 0: terminate Monitoring with an error (default)
## 1: writer blocks until queue capacity is available
## 2: writer discards new records until space is available
## Be careful when using the value '1' since then, the asynchronous writer
## is no longer decoupled from the monitored application.
kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueFullBehavior=0


#####
#kieker.monitoring.writer=kieker.monitoring.writer.namedRecordPipe.PipeWriter
#
## The name of the pipe used (must not be empty).
kieker.monitoring.writer.namedRecordPipe.PipeWriter.pipeName=kieker-pipe


#####
#kieker.monitoring.writer=kieker.monitoring.writer.jms.AsyncJMSWriter
#
## The url of the jndi provider that knows the jms service
kieker.monitoring.writer.jms.AsyncJMSWriter.ProviderUrl=tcp://127.0.0.1:3035/
#
## The topic at the jms server which is used in the publisher/subscribe communication.
kieker.monitoring.writer.jms.AsyncJMSWriter.Topic=queue1
#
## The type of the jms factory implementation, e.g.
## "org.exolab.jms.jndi.InitialContextFactory" for openjms 0.7.7
kieker.monitoring.writer.jms.AsyncJMSWriter.ContextFactoryType=org.exolab.jms.jndi.
    InitialContextFactory
#
## The service name for the jms connection factory.
kieker.monitoring.writer.jms.AsyncJMSWriter.FactoryLookupName=ConnectionFactory
#
## The time that a jms message will be kept alive at the jms server before
## it is automatically deleted.
kieker.monitoring.writer.jms.AsyncJMSWriter.MessageTimeToLive=10000
```

```
#
## Asynchronous writers need to store monitoring records in an internal buffer.
## This parameter defines its capacity in terms of the number of records.
kieker.monitoring.writer.jms.AsyncJMSWriter.QueueSize=10000
#
## Behavior of the asynchronous writer when the internal queue is full:
## 0: terminate Monitoring with an error (default)
## 1: writer blocks until queue capacity is available
## 2: writer discards new records until space is available
## Be careful when using the value '1' since then, the asynchronous writer
## is no longer decoupled from the monitored application.
kieker.monitoring.writer.jms.AsyncJMSWriter.QueueFullBehavior=0


#####
#kieker.monitoring.writer=kieker.monitoring.writer.database.SyncDbWriter
#
## Database driver classname
## Examples: MySQL -> com.mysql.jdbc.Driver
kieker.monitoring.writer.database.SyncDbWriter.DriverClassname=com.mysql.jdbc.Driver
#
## Connection string
## Examples:
##  MySQL: jdbc:mysql://HOSTNAME/DBNAME?user=DBUSER&password=DBPASS
##  DerbyDB: jdbc:derby:DBNAME;user=DBUSER;password=DBPASS
kieker.monitoring.writer.database.SyncDbWriter.ConnectionString=jdbc:mysql://HOSTNAME/
    DBNAME?user=DBUSER&password=DBPASS
#
## Name of the database table
## (can be generated using the file table-for-monitoring.sql)
kieker.monitoring.writer.database.SyncDbWriter.TableName=kiekerdata


#####
#kieker.monitoring.writer=kieker.monitoring.writer.database.AsyncDbWriter
#
## Database driver classname
## Examples: MySQL -> com.mysql.jdbc.Driver
kieker.monitoring.writer.database.AsyncDbWriter.DriverClassname=com.mysql.jdbc.Driver
#
## Connection string
## Examples:
##  MySQL: jdbc:mysql://HOSTNAME/DBNAME?user=DBUSER&password=DBPASS
##  DerbyDB: jdbc:derby:DBNAME;user=DBUSER;password=DBPASS
kieker.monitoring.writer.database.AsyncDbWriter.ConnectionString=jdbc:mysql://HOSTNAME/
    DBNAME?user=DBUSER&password=DBPASS
#
## Name of the database table
## (can be generated using the file table-for-monitoring.sql)
kieker.monitoring.writer.database.AsyncDbWriter.TableName=kiekerdata
```

```
#
## The number of concurrent Database connections.
kieker.monitoring.writer.database.AsyncDbWriter.numberOfConnections=4
#
## Load the initialExperimentId from the DB and increase it by 1
## instead of using the value from the configuration.
## (Currently not implemented!)
kieker.monitoring.writer.database.AsyncDbWriter.loadInitialExperimentId=false
```

Listing B.1: kieker.monitoring.properties

C Additional Source Code Listings

C.1 MyNamedPipeManager and MyPipe

```
package bookstoreApplication;

import java.util.HashMap;

public class MyNamedPipeManager {

    private static final MyNamedPipeManager PIPE_MGR_INSTANCE = new
          MyNamedPipeManager();

    /* Not synchronized! */
    private final HashMap<String, MyPipe> pipeMap = new HashMap<String, MyPipe>();

    public static MyNamedPipeManager getInstance() {
        return MyNamedPipeManager.PIPE_MGR_INSTANCE;
    }

    /**
     * Returns a pipe with name pipeName. If a pipe with this name does not
     * exist prior to the call, it will be created.
     *
     * @param pipeName name of the (new) pipe.
     * @return the pipe
     * @throws IllegalArgumentException
     *             if the given name is null or has length zero.
     */
    public synchronized MyPipe acquirePipe(final String pipeName)
            throws IllegalArgumentException {
        if ((pipeName == null) || (pipeName.length() == 0)) {
            throw new IllegalArgumentException("Invalid connection name: '" + pipeName + "
                '");
        }
        MyPipe conn = this.pipeMap.get(pipeName);
        if (conn == null) {
            conn = new MyPipe(pipeName);
            this.pipeMap.put(pipeName, conn);
        }
        return conn;
    }
```

```
}
```

Listing C.1: MyNamedPipeManager.java

```java
package bookstoreApplication;

import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.TimeUnit;

public class MyPipe {
  private final String pipeName;
  private final LinkedBlockingQueue<PipeData> buffer =
                new LinkedBlockingQueue<PipeData>();

  public MyPipe(final String pipeName) {
    this.pipeName = pipeName;
  }

  public String getPipeName() {
    return this.pipeName;
  }

  public void put(final PipeData data) throws InterruptedException {
    this.buffer.put(data);
  }

  public PipeData poll(final long timeout) throws InterruptedException {
    return this.buffer.poll(timeout, TimeUnit.SECONDS);
  }

}
```

Listing C.2: MyPipe.java

D Example Console Outputs

D.1 Quick Start Example (Chapter 2)

```
14.08.2010 12:39:54 kieker.monitoring.core.MonitoringController <init>
INFO: The VM has the name Laptop Thread:1
14.08.2010 12:39:54 kieker.monitoring.core.MonitoringController <init>
INFO: Virtual Machine start time 1281782393977
14.08.2010 12:39:54 kieker.monitoring.core.MonitoringController loadPropertiesFile
INFO: Loading properties from Kieker.Monitoring library jar!META-INF/kieker.monitoring.properties.
    default
14.08.2010 12:39:54 kieker.monitoring.core.MonitoringController loadPropertiesFile
INFO: You can specify an alternative properties file using the property 'kieker.monitoring.configuration
    '
14.08.2010 12:39:54 kieker.monitoring.core.MonitoringController loadPropertiesFile
INFO: Debug mode disabled
14.08.2010 12:39:54 kieker.monitoring.writer.filesystem.AsyncFsWriter init
INFO: Directory for monitoring data: C:\Temp\/tpmon-20100814-103954167-UTC/
14.08.2010 12:39:54 kieker.monitoring.writer.filesystem.FsWriterThread <init>
INFO: New FsWriter thread created
14.08.2010 12:39:54 kieker.monitoring.writer.filesystem.AsyncFsWriter init
INFO: (1 threads) will write to the file system
14.08.2010 12:39:54 kieker.monitoring.writer.filesystem.FsWriterThread run
INFO: FsWriter thread running
14.08.2010 12:39:54 kieker.monitoring.core.MonitoringController <init>
INFO: Initialization completed.
 Writer Info: monitoringDataWriter : kieker.monitoring.writer.filesystem.AsyncFsWriter,
     monitoringDataWriter config : (below), filenamePrefix :C:\Temp\, outputDirectory :C:\Temp\/tpmon
     -20100814-103954167-UTC/, version :1.2-SNAPSHOT-20100814, debug :false, enabled :true,
     terminated :false, experimentID :0, vmname:Laptop
Bookstore.main: Starting request 0
Bookstore.main: Starting request 1
14.08.2010 12:39:54 kieker.monitoring.writer.filesystem.MappingFileWriter writeMapping
INFO: Registered monitoring record type with id '1': kieker.common.record.OperationExecutionRecord
Bookstore.main: Starting request 2
Bookstore.main: Starting request 3
Bookstore.main: Starting request 4
```

Listing D.1: Execution of the manually instrumented Bookstore application (Section 2.3)

```
Apr 28, 2011 3:46:50 PM kieker.common.record.MonitoringRecordTypeRegistry registerRecordTypeIdMapping
INFO: Registered record type mapping 1/kieker.common.record.OperationExecutionRecord
Apr 28, 2011 3:46:50 PM kieker.analysis.reader.filesystem.FSDirectoryReader processInputFile
INFO: < Loading /tmp/kieker-20110427-142244899-UTC-Kaapstad-KIEKER-SINGLETON/kieker
    -20110427-142244920-UTC-Thread-1.dat
maximum response time exceeded by 211238 ns: bookstoreApplication.Catalog.getBook(..)
maximum response time exceeded by 193785 ns: bookstoreApplication.Catalog.getBook()
maximum response time exceeded by 197205 ns: bookstoreApplication.Catalog.getBook(..)
maximum response time exceeded by 226278 ns: bookstoreApplication.Catalog.getBook()
maximum response time exceeded by 191895 ns: bookstoreApplication.Catalog.getBook(..)
maximum response time exceeded by 229681 ns: bookstoreApplication.Catalog.getBook()
maximum response time exceeded by 228173 ns: bookstoreApplication.Catalog.getBook(..)
maximum response time exceeded by 196394 ns: bookstoreApplication.Catalog.getBook()
maximum response time exceeded by 227399 ns: bookstoreApplication.Catalog.getBook(..)
maximum response time exceeded by 227027 ns: bookstoreApplication.Catalog.getBook()
Apr 28, 2011 3:46:50 PM kieker.analysis.reader.filesystem.FSReaderCons execute
INFO: All reader threads provided FS_READER_TERMINATION_MARKER
```

Listing D.2: Execution of the example analysis (Section 2.4)

D.2 Trace Monitoring, Analysis & Visualization (Chapter 5)

```
Bookstore.main: Starting request 0
Apr 28, 2011 4:28:29 PM kieker.monitoring.core.configuration.Configuration createSingletonConfiguration
INFO: Loading properties from properties file in classpath: 'META-INF/kieker.monitoring.properties'
Apr 28, 2011 4:28:29 PM kieker.monitoring.core.controller.MonitoringController createInstance
INFO: Current State of kieker.monitoring (1.3-dev-20110427) Status: 'enabled'
  Name: 'KIEKER'; Hostname: 'Kaapstad'; experimentID: '1'
 WriterController:
  Number of Inserts: '0'
  Automatic assignment of logging timestamps: 'true'
Writer: 'kieker.monitoring.writer.filesystem.AsyncFsWriter'
   Configuration:
     kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueFullBehavior='0'
     kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueSize='10000'
     kieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath=''
     kieker.monitoring.writer.filesystem.AsyncFsWriter.storeInJavaIoTmpdir='true'
  Writer Threads (1):
     Finished: 'false'; Writing to Directory: '/tmp/kieker-20110428-142829399-UTC-Kaapstad-
          KIEKER'
Sampling Controller: Periodic Sensor available: Current Poolsize: '0'; Scheduled Tasks: '0'
Apr 28, 2011 4:28:29 PM kieker.monitoring.core.registry.ControlFlowRegistry <init>
INFO: First threadId will be 7752665283541598209
Apr 28, 2011 4:28:29 PM kieker.monitoring.writer.filesystem.MappingFileWriter writeMapping
INFO: Registered monitoring record type with id '1': kieker.common.record.OperationExecutionRecord
```

Listing D.3: Execution of the Bookstore with AspectJ trace instrumentation (Section 5.1.1)

E Ant Scripts

E.1 Quick Start Example (Chapter 2)

The following `build.xml` and `build.properties` files can be used for compiling and executing the manually instrumentated Bookstore application and the analysis, as described in Chapter 2. The files are included in the directory `examples/userguide/ch2-manual-instrumentation/`.

In order to run the analysis of the application, it is necessary to pass the location of the monitoring log directory. This is done via the parameter *analysis.directory*, e.g.:

```
▷ ant run-analysis -Danalysis.directory tmp/tpmon-20100812-122906207-UTC
```

Listing E.1: Command to compile and run the instrumented Bookstore via ant

```
build.dir=build
src.dir=src
lib.dir=lib

jar.file.monitoring=BookstoreMonitoring.jar
jar.file.analysis=BookstoreAnalysis.jar

jar.file.commons-logging=commons-logging-1.1.1.jar
jar.file.kieker=kieker-1.3-dev.jar

main-class-monitoring=bookstoreApplication.BookstoreStarter
main-class-analysis=bookstoreApplication.BookstoreAnalysisStarter

meta.dir=META-INF

msg.filesNotFound=One or more of the required libraries could not be found. Please add the
    following files to the ${lib.dir} directory: ${jar.file.commons-logging}, ${jar.file.kieker}.
```

Listing E.2: build.properties

```
<project name="Bookstore-Application" basedir="." default="run-monitoring">

  <property file="build.properties" />

  <path id="classpath">
    <fileset dir="${lib.dir}" includes="**/*.jar" />
  </path>
```

```
<condition property="not-all-files-available">
  <not><and>
    <available file="${lib.dir}/${jar.file.commons-logging}" />
    <available file="${lib.dir}/${jar.file.kieker}" />
  </and></not>
</condition>

<target name="-check-files" if="not-all-files-available">
  <fail message="${msg.filesNotFound}" />
</target>

<target name="run-monitoring" depends="-check-files, -build-jar-monitoring">
  <java fork="true" classname="${main-class-monitoring}">
    <classpath>
      <path refid="classpath" />
      <path location="${jar.file.monitoring}" />
    </classpath>
  </java>
</target>

<target name="run-analysis" depends="-check-files, -build-jar-analysis">
  <java fork="true" classname="${main-class-analysis}">
    <arg line="${analysis.directory}" />
    <classpath>
      <path refid="classpath" />
      <path location="${jar.file.analysis}" />
    </classpath>
  </java>
</target>

<target name="-build-jar-monitoring" depends="-compile">
  <jar destfile="${jar.file.monitoring}" basedir="${build.dir}">
    <manifest>
      <attribute name="Main-Class" value="${main-class-monitoring}" />
    </manifest>
  </jar>
  <delete dir="${build.dir}" />
</target>

<target name="-build-jar-analysis" depends="-compile">
  <jar destfile="${jar.file.analysis}" basedir="${build.dir}">
    <manifest>
      <attribute name="Main-Class" value="${main-class-analysis}" />
    </manifest>
  </jar>
  <delete dir="${build.dir}" />
</target>

<target name="-compile" depends="-init">
  <javac srcdir="${src.dir}" destdir="${build.dir}" classpathref="classpath" />
```

```
    <mkdir dir="${build.dir}/${meta.dir}" />
  </target>

  <target name="-init">
    <delete dir="${build.dir}" />
    <mkdir dir="${build.dir}" />
  </target>
</project>
```

Listing E.3: build.xml

E.2 Custom Components (Chapters 3 and 4)

The following `build.xml` and `build.properties` files can be used for compiling and executing the manually instrumentated Bookstore application with the custom components, as described in Chapters 3 and 4. The files are included in the directory `examples/userguide/ch3-4-custom-components/`.

```
build.dir=build
src.dir=src
lib.dir=lib
meta.dir=META-INF

jar.file=BookstoreApplication.jar
properties.file=kieker.monitoring.properties

jar.file.commons-logging=commons-logging-1.1.1.jar
jar.file.kieker=kieker-1.3-dev.jar

main-class=bookstoreApplication.Starter

meta.dir=META-INF

msg.filesNotFound=One or more of the required libraries could not be found. Please add the
    following files to the ${lib.dir} directory: ${jar.file.commons-logging}, ${jar.file.kieker}
```

Listing E.4: build.properties

```
<project name="Bookstore-Application" basedir="." default="run">

  <property file="build.properties" />

  <path id="classpath">
    <fileset dir="${lib.dir}" includes="**/*.jar" />
  </path>

  <condition property="not-all-files-available">
    <not>
      <and>
        <available file="${lib.dir}/${jar.file.commons-logging}" />
        <available file="${lib.dir}/${jar.file.kieker}" />
      </and>
    </not>
  </condition>

  <target name="-check-files" if="not-all-files-available">
    <fail message="${msg.filesNotFound}" />
  </target>

  <target name="run" depends="-check-files, -build-jar">
    <java fork="true" classname="${main-class}">
```

```
      <classpath>
        <path refid="classpath" />
        <path location="${jar.file}" />
      </classpath>
      <jvmarg value="-Dkieker.monitoring.properties=${meta.dir}/${properties.file}" />
    </java>
  </target>

  <target name="-build-jar" depends="-compile">
    <jar destfile="${jar.file}" basedir="${build.dir}">
      <manifest>
        <attribute name="Main-Class" value="${main-class}" />
      </manifest>
    </jar>
    <delete dir="${build.dir}" />
  </target>

  <target name="-compile" depends="-init">
    <javac srcdir="${src.dir}" destdir="${build.dir}" classpathref="classpath" />
    <mkdir dir="${build.dir}/${meta.dir}" />
    <copy file="${meta.dir}/${properties.file}" tofile="${build.dir}/${meta.dir}/${properties.file
        }" />
  </target>

  <target name="-init">
    <delete dir="${build.dir}" />
    <mkdir dir="${build.dir}" />
  </target>

</project>
```

Listing E.5: build.xml

E.3 AspectJ-based Trace Monitoring (Chapter 5)

The following `build.xml` and `build.properties` files can be used for compiling and executing the Bookstore application instrumentated with AspectJ, as described in Chapter 5.

The files are included in the directory `examples/userguide/ch5-trace-monitoring-aspectj/`.

```
build.dir=build
src.dir=src
lib.dir=lib
meta.dir=META-INF

jar.file=BookstoreApplication.jar
properties.file=kieker.monitoring.properties

jar.file.commonslogging=commons-logging-1.1.1.jar
jar.file.aspectjweaver=aspectjweaver-1.6.11.jar
jar.file.kieker=kieker-1.3-dev.jar

main-class=bookstoreTracing.BookstoreStarter
main-class-hostname-rewriter=bookstoreTracing.BookstoreHostnameRewriter

meta.dir=META-INF

msg.filesNotFound=One or more of the required libraries could not be found. Please add the
    following files to the ${lib.dir} directory: ${jar.file.commonslogging}, ${jar.file.
    aspectjweaver}, ${jar.file.kieker}
```

Listing E.6: build.properties

```
<project name="Tutorial-Example" basedir="." default="run">

  <property file="build.properties" />

  <path id="classpath">
    <fileset dir="${lib.dir}" includes="**/*.jar" />
  </path>

  <condition property="not-all-files-available">
    <not>
      <and>
        <available file="${lib.dir}/${jar.file.commonslogging}" />
        <available file="${lib.dir}/${jar.file.aspectjweaver}" />
        <available file="${lib.dir}/${jar.file.kieker}" />
      </and>
    </not>
  </condition>

  <target name="-check-files" if="not-all-files-available">
```

```
    <fail message="${msg.filesNotFound}" />
  </target>

  <target name="run" depends="-check-files, -build-jar">
    <java fork="true" classname="${main-class}">
      <classpath>
        <path refid="classpath" />
        <path location="${jar. file }" />
      </classpath>
                          <arg line="${num.requests}" />
      <jvmarg value="-javaagent:${lib.dir}/${jar . file . aspectjweaver}" />
      <jvmarg value="-Dorg.aspectj.weaver.showWeaveInfo=true" />
      <jvmarg value="-Daj.weaving.verbose=true" />
    </java>
  </target>

  <target name="run-hostname-rewriter" depends="-check-files, -build-jar">
    <java fork="true" classname="${main-class-hostname-rewriter}">
      <arg line="${analysis . directory }" />
      <classpath>
        <path refid="classpath" />
        <path location="${jar. file }" />
      </classpath>
    </java>
  </target>


  <target name="-build-jar" depends="-compile">
    <copy file="META-INF/aop.xml" tofile="${build.dir}/META-INF/aop.xml" />
    <jar destfile ="${jar. file }" basedir="${build. dir }">
      <manifest>
        <attribute name="Main-Class" value="${main-class}" />
      </manifest>
    </jar>
    <delete dir="${build. dir }" />
  </target>

  <target name="-compile" depends="-init">
    <javac srcdir="${src. dir }" destdir ="${build. dir }" classpathref ="classpath" />
  </target>

  <target name="-init">
    <delete dir="${build. dir }" />
    <mkdir dir="${build.dir}" />
  </target>

</project>
```

Listing E.7: build.xml

F Java EE Servlet Container Example

The Kieker download site[1] includes an additional example `JavaEEServletContainerExample`. Using the sample Java Web application iBATIS JPetStore[2], this example demonstrates how to employ Kieker.Monitoring for monitoring a Java application running in a Java EE container—in this case, Apache Tomcat[3]. Monitoring probes based on the Java EE Servlet API and AspectJ are used to monitor execution, trace, and session data (see Section 5).

> The example is currently only prepared for UNIX-like systems. However, based on the descriptions below, it shouldn't be too difficult to run it under Windows

F.1 Preparation of the Tomcat Servlet Container

1. Copy the files `kieker-1.3-dev.jar`, `commons-logging-1.1.1.jar`, and `aspectjweaver-1.6.11.jar` from Kieker's binary distribution to the Tomcat's `lib/` directory.
2. Copy the file `kieker-monitoring-servlet-1.3-dev.war` from Kieker's binary distribution to the Tomcat's `webapps/` directory.
3. Tomcat's `lib/` directory contains the files `kieker.monitoring.properties` and `aop.xml` — the configuration of Kieker.Monitoring and the AspectJ-based instrumentation.
4. Tomcat's `bin/catalina.sh` file was modified to add the location of the `kieker.monitoring.properties` and the AspectJ agent to the argument list of the JVM call:

```
JAVA_OPTS="-javaagent:lib/aspectjweaver-1.6.11.jar -Dorg.aspectj.weaver.showWeaveInfo=false -Daj.weaving.verbose=false"
JAVA_OPTS="${JAVA_OPTS} -Dkieker.monitoring.configuration=$(dirname $0)/../lib/META-INF/kieker.monitoring.properties"
```

Listing F.1: Excerpt from catalina.sh

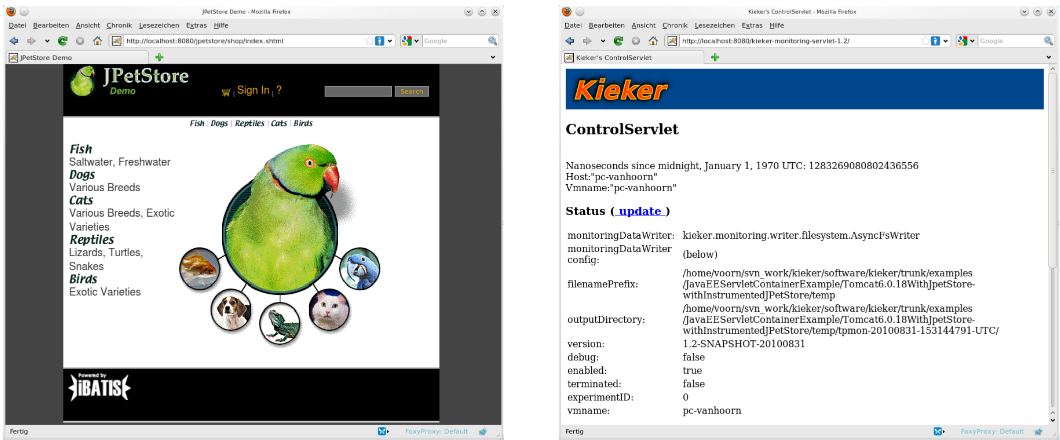
[1] `http://sourceforge.net/projects/kieker/files`
[2] `http://ibatis.apache.org/`
[3] `http://tomcat.apache.org/`

5. A prepared `jpetstore.war` file is already located in the Tomcat's `webapps/` directory. If you want to rebuild the sources, for example to modify the instrumentation, see Section F.3.

F.2 JPetStore and Kieker.Monitoring Control Servlet

We will now start the Tomcat server and generate some monitoring data by manually accessing the JPetStore web application.

1. Start the Tomcat server using the `bin/startup.sh` script in the Tomcat's `bin/` directory (you may have to execute "`chmod +x bin/*.sh`" in order to set the executable flags of the shell scripts). You should make sure, that the Tomcat started properly, by taking a look at the `logs/catalina.out` file. On error, the file `logs/localhost.<date>.log` may contain details to resolve the issue.
2. Now, you can access the JPetStore application by opening the URL `http://localhost:8080/jpetstore/` (Figure F.1(a)). Kieker intialization messages should appear in `logs/catalina.out`. The output includes the information where the monitoring data is written to (should be Tomcat's `temp/tpmon-<DATE-TIME>/` directory).
3. Browse through the application to generate some monitoring data. This data can be analyzed and visualized using Kieker.TraceAnalysis, as described in Chapter 5.
4. Kieker includes a servlet to control the status of Kieker.Monitoring. It can be accessed via `http://localhost:8080/kieker-monitoring-servlet-1.3-dev/` (Figure F.1(b)).



(a) iBATIS JPetStore

(b) Kieker.Monitoring control servlet

Figure F.1

F.3 Rebuilding the JPetStore Application

In order to rebuild the JPetStore sources (located in `JPetStore-5.0-instrumented/`), the following steps are required:

1. Copy the `kieker-1.3-dev.jar` from Kieker's binary distribution to the JPetStore's `devlib/` directory. It is required for the annotation-based instrumentation (`@OperationExecutionMonitoringProbe`), as described in Chapter 5
2. Build the JPetStore with the `build.xml` by calling `ant` from within `build/` directory.
3. You'll find the packaged JPetStore `.war`-file in `build/wars/`.
4. Copy the file to the Tomcat's `webapps/` directory.

G Using the JMS Writer and Reader

This chapter gives a brief description on how to use the `AsyncJMSWriter` and `JMSReader` classes. The directory `examples/userguide/appendix-JMS/` contains the sources, ant scripts etc. used in this example. It is based on the Bookstore application with manual instrumentation presented in Chapter 2.

1. Copy the files `kieker-1.3-dev.jar` and `commons-logging-1.1.1.jar` from the binary distribution to the example's `lib/` directory.
2. The file `examples/userguide/appendix-JMS/META-INF/kieker.monitoring.properties` is already configured to use the `AsyncJMSWriter`:

```
# If monitoring is disabled, the MonitoringController simply pauses.
# Furthermore, probes should stop collecting new data and monitoring
# writers stop should stop writing existing data.
```

Listing G.1: Excerpt from `kieker.monitoring.properties` configuring the JMS writer

3. Download an OpenJMS install archive from `http://openjms.sourceforge.net` and decompress it to the root directory of the example.
4. Copy the following files from the OpenJMS `lib/` folder to the `lib/` directory of this example:
   a) `openjms-<version>.jar`
   b) `openjms-common-<version>.jar`
   c) `openjms-net-<version>.jar`
   d) `jms-<version>.jar`
   e) `concurrent-<version>.jar`
   f) `spice-jndikit-<version>.jar`

The execution of the example is performed by the following three steps:

1. Start the JMS server (you may have to set your `JAVA_HOME` variable first):

```
▷ openjms-<version>/bin/startup.sh
```

2. Start the analysis part (in a new terminal):

```
▷ ant run-analysis
```

3. Start the instrumented Bookstore (in a new terminal):

```
▷ ant run-monitoring
```

H Libraries

The following table shows all libraries which are used by Kieker and explains them briefly. These libraries are included in the `lib/` directory of both the Kieker binary and source distributions.

The Apache Commons [6] library (`commons-logging-1.1.1.jar`) is the only third-party library always needed when using Kieker. The need to provide the additional libraries in the classpath depends on the specific configuration. For example, the AspectJ libraries are only required when using AspectJ-based monitoring probes.

Filename	Description
aspectjrt-1.6.11.jar	This jar-file contains the runtime library for AspectJ programs.
aspectjtools-1.6.11.jar	This package contains the tools (the AspectJ Compiler and Browser) for AspectJ.
aspectjweaver-1.6.11.jar	This jar contains the weaver-agent for the aspect-oriented-extension for Java named AspectJ.
commons-cli-1.2.jar	Apache Commons CLI provides a simple API for working with the command line arguments and options.
commons-io-1.2.jar	Apache Commons-IO contains utility classes, stream implementations, file filters, and endian classes.
commons-logging-1.1.1.jar	Apache Commons Logging is a thin adapter allowing configurable bridging to other, well known logging systems.
commons-pool-1.2.jar	Apache Commons Pool is an Object-pooling API supplying different interfaces and classes to create modular object pools.
concurrent-1.3.4.jar	This library supplies different thread-safe classes for the enhanced development of multithreaded Java applications.
cxf-api-2.2.10.jar	Apache CXF is an open source services framework.

cxf-common-utilities-2.2.10.jar	This package contains different classes for Apache CXF.
cxf-rt-bindings-soap-2.2.10.jar	This package contains necessary files to use Apache CXF as well with the Simple Object Access Protocol (SOAP).
cxf-rt-core-2.2.10.jar	This library contains the Apache CXF Runtime Core.
derby.jar	Apache Derby is a lightweight database written in Java which can also be used as an embedded database. This library contains the necessary drivers for the database as well as the database management system itself.
jmc.jar	This library contains the Java Media Components which can be used for example for playing video content in Swing applications.
jms-1.1.jar	Java Message Service is an API to send and receive messages within a client and to control so called Message Oriented Middleware (MOM).
jndi-1.2.1.jar	The Java Naming and Directory Interface is an API which provides methods for multiple naming and directory services. It can be used for example to register disposed files in a network and to allow other part of a Java program to use them for RMI calls.
junit-4.5.jar	This jar-file contains the necessary classes for the JUnit-tests, which can be used to test automatically Java classes.
log4j-1.2.15.jar	Apache log4j is a framework for the logging of messages, errors and exceptions in Java applications.
mysql-connector-java-5.1.5-bin.jar	This library contains the drivers to connect from a Java application to a MySQL database system.
openjms-0.7.7-beta-1.jar	OpenJMS is an open source implementation of Sun Microsystems's Java Message Service API 1.1 Specification

openjms-common-0.7.7-beta-1.jar	OpenJMS is an open source implementation of Sun Microsystems's Java Message Service API 1.1 Specification
openjms-net-0.7.7-beta-1.jar	OpenJMS is an open source implementation of Sun Microsystems's Java Message Service API 1.1 Specification
rabbitmq-client.jar	This library contains the client for the RabbitMQ messaging system.
Scenario.jar	This package provides scene graph functionality for Java.
servlet.jar	This package contains different classes for the work with servlets.
servlet-api.jar	The Java Servlet API supplies protocols to let applications respond for example to HTTP requests.
sigar-1.6.3.jar	n/a
spice-jndikit-1.2.jar	The JNDI Kit is a toolkit for the easy use of the so called Java Naming and Directory Interface.
spring.jar	The spring framework delivers different methods and classes to make the handling with Java/Java EE easier.
spring-web.jar	This library contains the web application context, multipart resolver, Struts support, JSF support and web utilities for the spring framework.

Bibliography

[1] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin (1997). Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97), Jyváskylá, Finland, 9-13 June, 1997*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Berlin: Springer.

[2] Oracle (2010). Java Messaging Service (JMS). `http://www.oracle.com/technetwork/java/index-jsp-142945.html`.

[3] Oracle (2010). Java Servlet Technology. `http://www.oracle.com/technetwork/java/index-jsp-135475.html`.

[4] SpringSource (2010). Spring. `http://www.springsource.org/`.

[5] The Apache Foundation (2010). Apache CXF. `http://cxf.apache.org/`.

[6] The Apache Foundation (2010). Commons Logging. `http://commons.apache.org/logging/`.

[7] The Eclipse Foundation (2010). The AspectJ Project. `http://www.eclipse.org/aspectj/`.