http://kieker-monitoring.net

Kieker 1.9 User Guide*

Kieker Project

April 15, 2014

Kiel University
Department of Computer Science
Software Engineering Group
Christian-Albrechts-Platz 4
24118 Kiel, Germany

University of Stuttgart
Institute of Software Technology
Reliable Software Systems Group
Universitätsstraße 38
70569 Stuttgart, Germany

**For guide lines on how to cite Kieker and this document, please see Section 1.4.*

Contents

1 Introduction

Modern software applications are often complex and have to fulfill a large set of functional and non-functional requirements. 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 dynamic analysis based on application-level monitoring, which allows to log the behavior of the application and to discover, for example, application-internal control flows, calling dependencies, and method response times.

Dynamic analysis 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 has been designed for continuous monitoring in production systems inducing only a very low overhead. Further information on the overhead caused by Kieker is provided at `http://kieker-monitoring.net/overhead-evaluation/`.

1.1 What is Kieker?

Kieker is a Java-based application performance monitoring and dynamic software analysis framework [12]. Monitoring adapters for other platforms, such as Visual Basic 6 (VB6), .NET, and COBOL, exist as well.[1] Figure 1.1 shows the framework's composition based on the two main components Kieker.Monitoring and Kieker.Analysis.

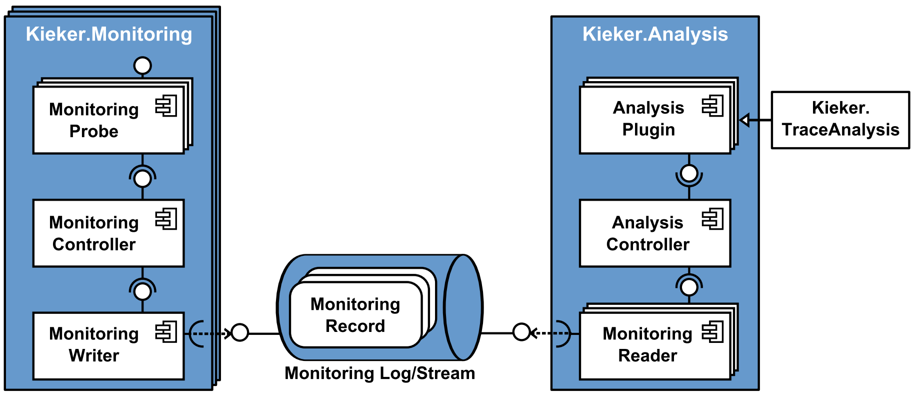


Figure 1.1: Overview of the framework components

[1]Contact us directly if you are interested in Kieker support for other platforms

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 pipe-and-filter architecture of analysis plugins, including monitoring readers and analysis filters.

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.

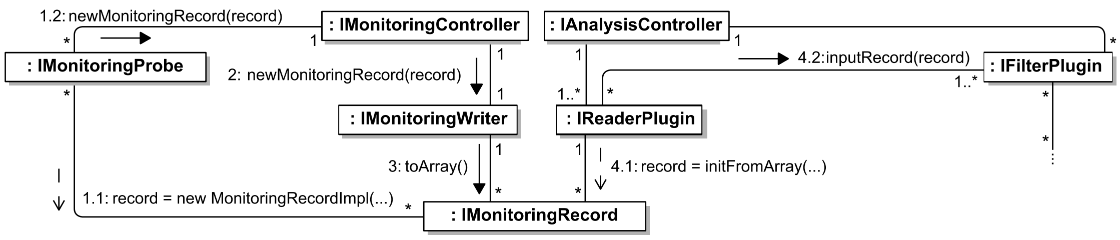


Figure 1.2: Communication among Kieker framework components

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, monitoring reader plugins read the records from the monitoring log/stream. These records can then be further processed by a configuration of additional filter and repository plugins, inter-connected via input and output ports.

1.2 Framework Components and Extension Points

Figure 1.3 depicts the possible extension points for custom components as well as the components which are already included in the Kieker distribution and detailed below.

- **Monitoring writers and corresponding readers** for file systems and SQL databases, for in-memory record streams (named pipes), as well writers and readers employing Java Management Extensions (JMX) [6] and Java Messaging Service (JMS) [5] technology. A special reader allows to replay existing persistent monitoring logs, for example to emulate incoming monitoring data—also in real-time.
- **Time sources** utilizing Java's *System.nanoTime()* (default) or *System.currentTimeMillis()* methods.

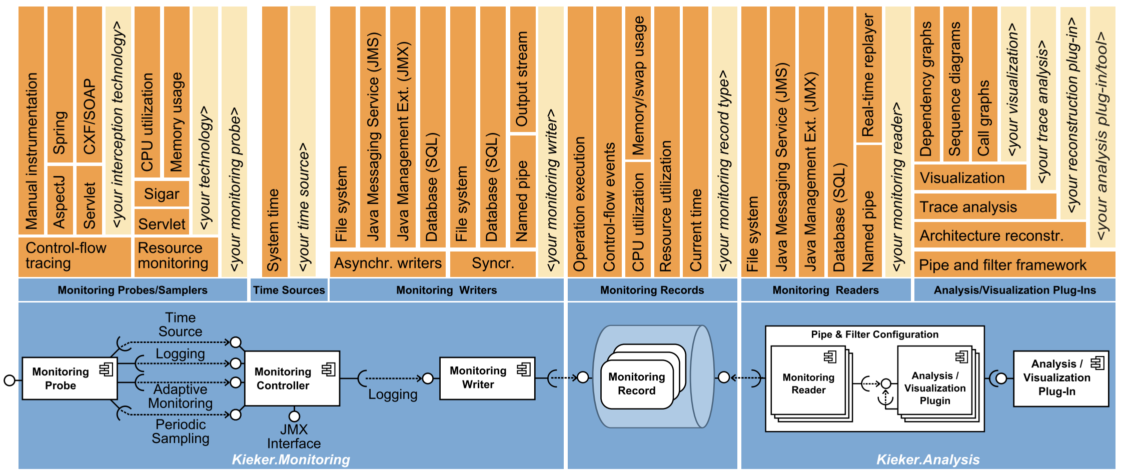


Figure 1.3: Kieker framework components and extension points for custom components

- **Monitoring record types** allowing to store monitoring data about operation executions (including timing, control-flow, and session information), CPU and resource utilization, memory/swap usage, as well as a record type which can be used to store the current time.
- **Monitoring probes**: A special feature of Kieker is the ability to monitor (distributed) traces of method executions and corresponding timing information. For monitoring this data, Kieker includes monitoring probes employing AspectJ [10], Java EE Servlet [7], Spring [8], and Apache CXF [9] technology. Additionally, Kieker includes probes for (periodic) system-level resource monitoring employing the Sigar library [1].
- **Analysis/Visualization plugins** can be assembled to pipe-and-filter architectures based on input and output ports. The Kieker.TraceAnalysis tool is itself implemented based on a re-usable set of Kieker.Analysis plugins allowing to reconstruct and visualize architectural models of the monitored systems, e.g., as dependency graphs, sequence diagrams, and call trees.

1.3 Licensing

Kieker is licensed under the Apache License, Version 2.0. You may obtain a copy of the license at `http://www.apache.org/licenses/LICENSE-2.0`.

The Kieker source and binary release archives include a number of third-party libraries. The `lib/` directory of the release archives contains a `.LICENSE` file for each third-party library, pointing to the respective license text.

1.4 Citing Kieker

When referencing Kieker resources in your publications, we would be happy if you respected the following guide lines:

- When referencing the Kieker project, please cite our ICPE 2012 [12] paper and/or our 2009 technical report [11]. Also, you might want to add a reference to our web site (`http://kieker-monitoring.net/`) like [4].
- When referencing this user guide, e.g., when reprinting contents, please use a citation like [3].

At `http://kieker-monitoring.net/research/publications/` we provide entries for BibTeX and other bibliography systems.

1.5 Kieker is Recommended by the SPEC Research Group

In 2011, Kieker was reviewed and accepted for distribution as part of the SPEC Research Group's repository of peer-reviewed tools for quantitative system evaluation and analysis. See `http://research.spec.org/projects/tools.html` for details.

1.6 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, e.g., analyzing Java EE systems, using the JMS writers and readers, as well as monitoring system-level measures (CPU, memory, etc.) with Sigar.

☞ The Java sources presented in this user guide, as well as pre-compiled binaries, are included in the `examples/userguide/` directory of the Kieker distribution (see Section 2.1). Also, the example directories can be imported as Eclipse projects.

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-1.9_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-1.9/
├── bin/ ........................................ Call scripts for Kieker tools
│   └── ...
├── dist/ ....................................... The Kieker framework libraries
│   ├── kieker-1.9.jar
│   └── ...
├── doc/ ...........................................................
│   └── kieker-1.9_userguide.pdf .................... PDF file of this document
├── examples/
│   └── userguide/ ........... Source code of the examples in this document
│       └── ...
├── lib/ ........................................ Libraries required by Kieker
│   └── ...
└── META-INF/ ................................... Example configuration files
    ├── kieker.monitoring.properties
    └── ...
```

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

[1] `http://kieker-monitoring.net/download/`

The Java sources presented in this user guide, as well as pre-compiled binaries, are included in the `examples/userguide/` directory. The file `kieker-1.9.jar` contains the Kieker.Monitoring and Kieker.Analysis components, as well as the Kieker.TraceAnalysis tool. The sample Kieker.Monitoring configuration file `kieker.monitoring.properties` will be detailed in Chapter 3. In addition to the `kieker-1.9.jar` file, the `dist/` directory includes variants of this `.jar` files with integrated third-party libraries. Additional information on these `.jar` files and when to use them will follow later in this document.

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.

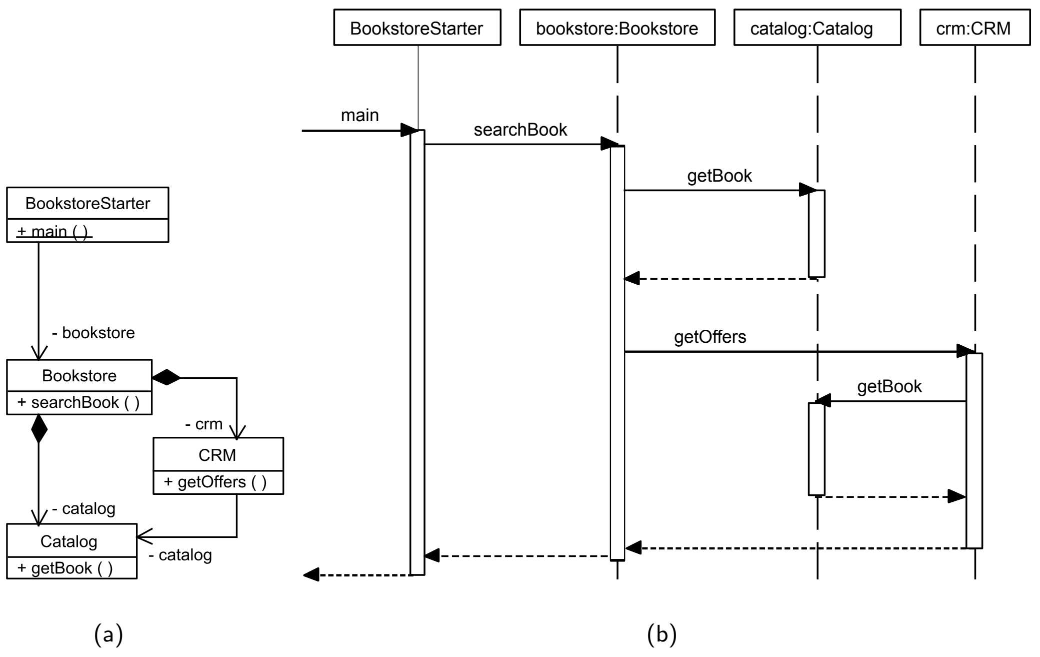


Figure 2.2: UML class diagram (a) and sequence diagram (b) of the Bookstore application

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/.../ch2bookstore/`, which are explained in detail below.

```
examples/
└─ userguide/
   └─ ch2–bookstore-application/
      └─ src/ ........................................ The directory for the source code files
         ├─ .../ch2bookstore/
         │  ├─ 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 and a pre-compiled binary of the uninstrumented Bookstore application can be found in the `examples/userguide/ch2-bookstore-application/` directory.

The class `BookstoreStarter` contains the application's *main* method (shown in Listing 2.1), 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
public static void main(final String[] args) {
  final Bookstore bookstore = new Bookstore();
  for (int i = 0; i < 5; i++) {
    System.out.println("Bookstore.main: Starting request " + i);
    bookstore.searchBook();
  }
}
```

Listing 2.1: *main* method from 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
public class Bookstore {

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

  public void searchBook() {
    this.catalog.getBook(false);
    this.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 provides only a business method to collect offers by using the catalog for some lookup. The additional catalog lookup is later used to illustrate different traces in the application.

```java
public class CRM {
  private final Catalog catalog;

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

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

Listing 2.3: CRM.java

Finally, the class `Catalog` is shown in Listing 2.4. It resembles the catalog component in the application.

```java
public class Catalog {

  public void getBook(final boolean complexQuery) {
    // nothing to do here
  }
}
```

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 command-line 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/kieker/examples/userguide/ch2bookstore/*.java -d build

> java -classpath build  kieker.examples.userguide.ch2bookstore.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

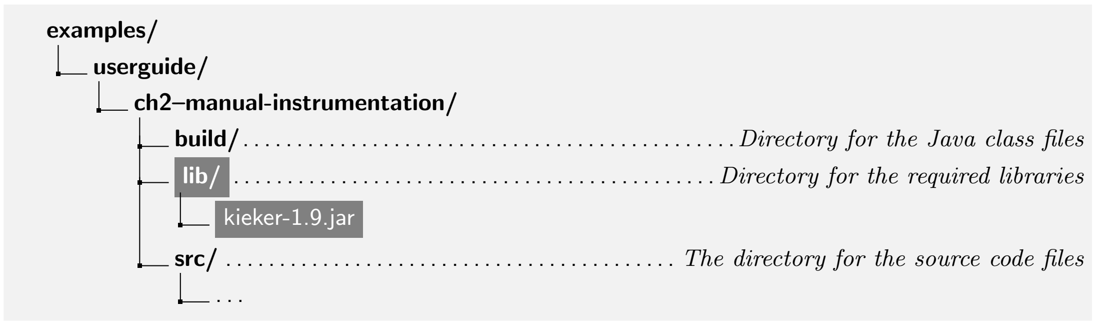
In this section, the Kieker example application was introduced and when everything went well, the bookstore is a runnable program. Furthermore, the composition of the application and its function should now be present. The next Section 2.3 will demonstrate how to monitor this example application employing 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, [2]) 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-1.9.jar` to the `lib/` directory of the example directory (see Section 2.2). The file is located in the `kieker-1.9/dist/` directory of the extracted Kieker archive, as described in Section 2.1. The final layout of the example directory 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-1.9.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 and pre-compiled binaries 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 .

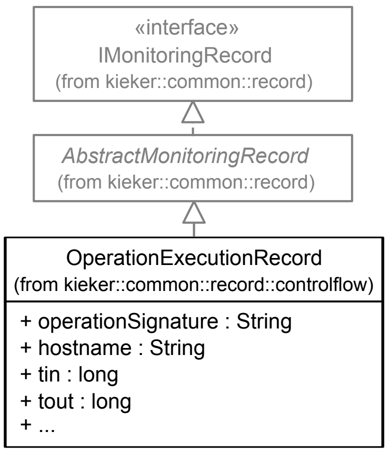

«interface»
IMonitoringRecord
(from kieker::common::record)
AbstractMonitoringRecord
(from kieker::common::record)
OperationExecutionRecord
(from kieker::common::record::controlflow)
+ operationSignature : String
+ hostname : String
+ tin : long
+ tout : long
+ ...

Figure 2.5: The class diagram of the operation execution record

The attributes relevant to this part are *operationSignature* and *hostname*, as well as *tin* and *tout* for the timestamps 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 25 and 26, the monitoring controller is instantiated. It provides the monitoring service for the instrumentation.

```
25  private static final IMonitoringController MONITORING_CONTROLLER =
26      MonitoringController.getInstance();
27  private final Catalog catalog = new Catalog();
28  private final CRM crm = new CRM(this.catalog);
29
30  public void searchBook() {
31    // 1.) Call Catalog.getBook() and log its entry and exit timestamps.
32    final long tin = MONITORING_CONTROLLER.getTimeSource().getTime();
33    this.catalog.getBook(false); // <-- the monitored execution
34    final long tout = MONITORING_CONTROLLER.getTimeSource().getTime();
35
36    final OperationExecutionRecord e = new OperationExecutionRecord(
37        "public void " + this.catalog.getClass().getName() + ".getBook(boolean
            )",
38        OperationExecutionRecord.NO_SESSION_ID,
39        OperationExecutionRecord.NO_TRACEID,
40        tin, tout, "myHost",
41        OperationExecutionRecord.NO_EOI_ESS,
42        OperationExecutionRecord.NO_EOI_ESS);
43    MONITORING_CONTROLLER.newMonitoringRecord(e);
44
45    // 2.) Call the CRM catalog's getOffers() method (without monitoring).
46    this.crm.getOffers();
47  }
48 }
```

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

The lines 32 and 34 are used to determine the current time in nanoseconds before and after the *getBook()* call. In lines 36 to 42, a monitoring record for this measurement is created and initialized, passing the method signature, the hostname, and the two time values as arguments. Finally the record is handed over to the monitoring controller (line 43) 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 36 and 38 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 40–46) and stored by calling the monitoring controller (see line 47).

```java
34  public void getOffers() {
35    // 1.) Call Catalog.getBook() and log its entry and exit timestamps.
36    final long tin = MONITORING_CONTROLLER.getTimeSource().getTime();
37    this.catalog.getBook(false); // <-- the monitored execution
38    final long tout = MONITORING_CONTROLLER.getTimeSource().getTime();
39
40    final OperationExecutionRecord e = new OperationExecutionRecord(
41        "public void " + this.catalog.getClass().getName() + ".getBook(boolean
            )",
42        OperationExecutionRecord.NO_SESSION_ID,
43        OperationExecutionRecord.NO_TRACEID,
44        tin, tout, "myHost",
45        OperationExecutionRecord.NO_EOI_ESS,
46        OperationExecutionRecord.NO_EOI_ESS);
47    MONITORING_CONTROLLER.newMonitoringRecord(e);
48  }
```

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 commands to compile and run the application under UNIX-like systems. Listing 2.10 shows the same commands for Windows. The expected working directory is the base directory of this example, i.e. `examples/userguide/ch2-manual-instrumentation/`.

```
▷ mkdir build
▷ javac src/kieker/examples/userguide/ch2bookstore/manual/*.java
        −classpath lib/kieker-1.9.jar −d build/

▷ java −classpath build/:lib/kieker-1.9.jar
        kieker.examples.userguide.ch2bookstore.manual.BookstoreStarter
```

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

Under Windows it is necessary to separate the classpath elements by a semicolon instead of a colon. Also, we recommend to use the Windows shell `cmd.exe` for this tutorial since problems have been reported for the Windows PowerShell.

```
▷ mkdir build
▷ javac src\kieker\examples\userguide\ch2bookstore\manual\*.java
        −classpath lib\kieker-1.9.jar −d build\

▷ java −classpath build\; lib\kieker-1.9.jar
        kieker.examples.userguide.ch2bookstore.manual.BookstoreStarter
```

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

If everything worked correctly, a new directory for the monitoring data with a name similar to `kieker-20120402-163314855-UTC-myHost-KIEKER-SINGLETON/` is created (see Figure 2.6). 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 variables `$TMPDIR` or `%temp%` for the location under Mac OS or Windows respectively. The exact location of the created monitoring log is reported in Kieker's console output (see for example Appendix G.1). 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-20130910-120352847-UTC-myHost-KIEKER-SINGLETON/
    ├── kieker.map
    └── kieker-20120402-163314882-UTC–000-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 fourth 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.

```
$0;1378814632852912850;1.8−SNAPSHOT;KIEKER−SINGLETON;myHost;1;false;0;NANOSECONDS;1
$1;1378814632852360525;public void kieker.examples.userguide.ch2bookstore.manual.Catalog.getBook (
    boolean);<no−session−id>;−1;1378814632849896821;1378814632852105483;myHost;−1;−1
```

Listing 2.11: kieker-20130910-120352862-UTC-000-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.

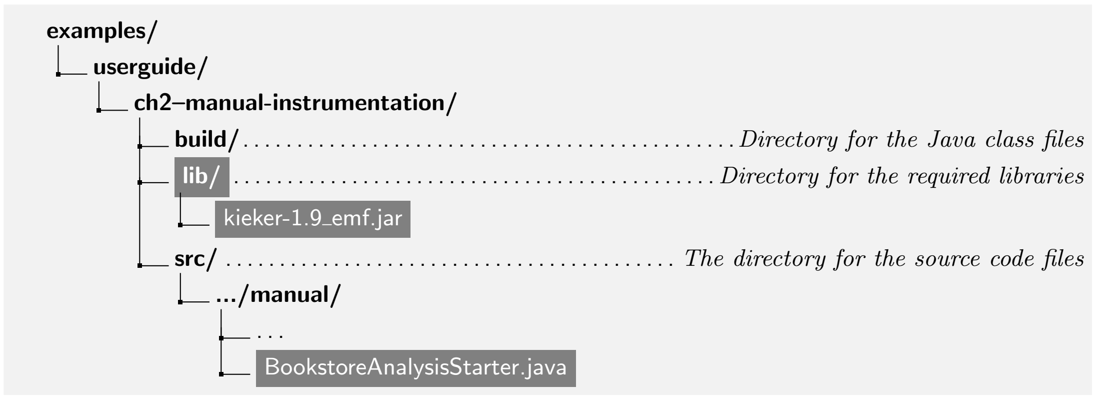
```
$0=kieker.common.record.misc.KiekerMetadataRecord
$1=kieker.common.record.controlflow.OperationExecutionRecord
```

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 4 to learn which plugins, i.e., readers and filters, are included in Kieker, how to use them, and how to develop custom plugins. Chapter 5 presents the Kieker.TraceAnalysis tool, which is also based on Kieker.Analysis. Kieker.Analysis has a dependency to the Eclipse Modeling Framework (EMF).[2] For this reason, we are using the `kieker-1.9_emf.jar` that is a variant of the `kieker-1.9.jar`, additionally including the required EMF dependencies. When using the `kieker-1.9.jar`, the `org.eclipse.emf.*.jar` files (to be found in Kieker's `lib/` directory) need to be added to the classpath.



```
examples/
└── userguide/
    └── ch2–manual-instrumentation/
        ├── build/ ........ Directory for the Java class files
        ├── lib/ .......... Directory for the required libraries
        │   └── kieker-1.9_emf.jar
        └── src/ .......... The directory for the source code files
            └── .../manual/
                ├── ...
                └── BookstoreAnalysisStarter.java
```

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

[2]http://www.eclipse.org/modeling/emf/

The analysis application developed in this section comprises the file `BookstoreAnalysisStarter.java`, as shown in Figure 2.7. This file can also be found in the directory `examples/userguide/ch2-manual-instrumentation/`. The file sets up the basic pipe-and-filter configuration depicted in Figure 2.8: Kieker's file system reader (`FSReader`) reads monitoring records from a file system monitoring log (as produced in the previous Section 2.3) and passes these to the `TeeFilter` plugin; the `TeeFilter` plugin reads events of arbitrary type (i.e., Java `Object`), prints them to a configured output stream, and also relays them to filters connected to the filter's output port *relayedEvents*.

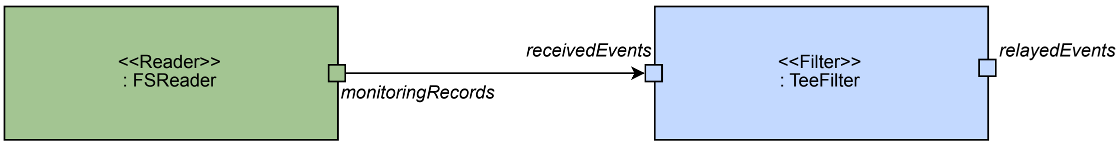


Figure 2.8: Example pipe-and-filter configuration

Kieker.Analysis pipe-and-filter configurations can be created programmatically, i.e., by configuring, instantiating, and connecting the plugins in a Java program.[3] For the example, this is demonstrated in Listing 2.13, which shows an excerpt from the `BookstoreAnalysisStarter`'s *main* method.

```java
        // Create Kieker.Analysis instance
        final IAnalysisController analysisInstance = new AnalysisController();

        // Set filesystem monitoring log input directory for our analysis
        final Configuration fsReaderConfig = new Configuration();
        fsReaderConfig.setProperty(FSReader.CONFIG_PROPERTY_NAME_INPUTDIRS, args
            [0]);
        final FSReader reader = new FSReader(fsReaderConfig, analysisInstance);

        // Create and register a simple output writer.
        final Configuration teeFilterConfig = new Configuration();
        teeFilterConfig.setProperty(TeeFilter.CONFIG_PROPERTY_NAME_STREAM,
            TeeFilter.CONFIG_PROPERTY_VALUE_STREAM_STDOUT);
        final TeeFilter teeFilter = new TeeFilter(teeFilterConfig,
            analysisInstance);

        // Connect the output of the reader with the input of the filter.
        analysisInstance.connect(reader, FSReader.OUTPUT_PORT_NAME_RECORDS,
            teeFilter, TeeFilter.INPUT_PORT_NAME_EVENTS);

        // Start the analysis
        analysisInstance.run();
```

Listing 2.13: BookstoreAnalysisStarter.java (excerpt from *main* method)

[3]As an alternative, a web-based user interface is available for Kieker [4]

The BookstoreAnalysisStarter follows a simple scheme. Each analysis tool has to create at least one AnalysisController which can be seen in Listing 2.13 in line 37. Then, the plugins, which may be readers or filters, are configured, and instantiated. The usage of the constructor ensures that the component is registered with the analysis instance. Lines 40–42 configure, instantiate, and register the file system monitoring log reader, which uses the 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. Lines 45–48 configure, instantiate, and register the TeeFilter, which outputs received events to the standard output. Lines 51 and 52 connect the TeeFilter's input port to the filesystem reader's output port. The analysis is started by calling its *run* method (line 55).

The Listings 2.14 and 2.15 describe how the analysis application can be compiled and executed under UNIX-like systems and Windows.

```
▷ mkdir build
▷ javac src/kieker/examples/userguide/ch2bookstore/manual/*.java
        −classpath lib/kieker-1.9_emf.jar −d build/

▷ java −classpath build/:lib/kieker-1.9_emf.jar
        kieker.examples.userguide.ch2bookstore.manual.BookstoreAnalysisStarter
       /tmp/kieker−20130910−120352847−UTC−myHost−KIEKER−SINGLETON
```

Listing 2.14: Commands to compile and run the analysis under UNIX-like systems

```
▷ mkdir build
▷ javac src\kieker\examples\userguide\ch2bookstore\manual\*.java
        −classpath lib\kieker-1.9_emf.jar −d build\

▷ java −classpath build\;lib\kieker-1.9_emf.jar
        kieker.examples.userguide.ch2bookstore.manual.BookstoreAnalysisStarter
       C:\Temp\kieker−20130910−120352847−UTC−myHost−KIEKER−SINGLETON
```

Listing 2.15: Commands to compile and run the analysis under Windows

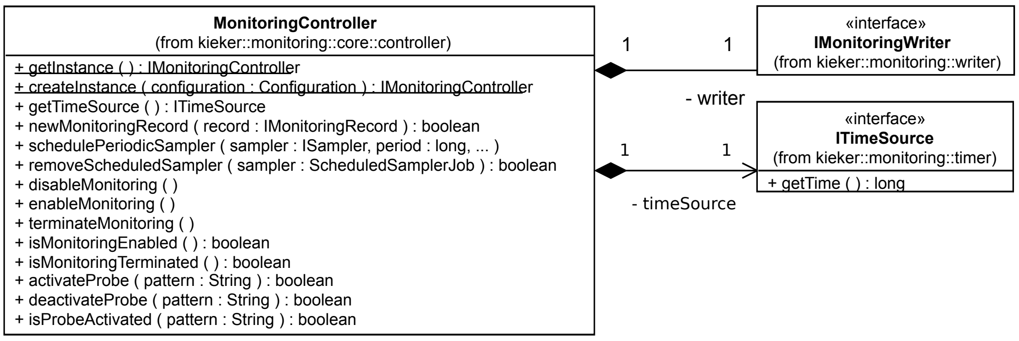
You need to make sure that the application gets the correct path from the monitoring run. The TeeFilter prints an output message for each record received. An example output can be found in Appendix G.1.

3 Kieker.Monitoring Component

The Java sources of this chapter, as well as a pre-compiled binary, can be found in the `examples/userguide/ch3-4-custom-components/` directory of the binary release.

3.1 Monitoring Controller

The `MonitoringController` constructs and controls a Kieker.Monitoring instance.



MonitoringController
(from kieker::monitoring::core::controller)
+ getInstance ( ) : IMonitoringController
+ createInstance (configuration : Configuration) : IMonitoringController
+ getTimeSource ( ) : ITimeSource
+ newMonitoringRecord (record : IMonitoringRecord) : boolean
+ schedulePeriodicSampler (sampler : ISampler, period : long, ...)
+ removeScheduledSampler (sampler : ScheduledSamplerJob) : boolean
+ disableMonitoring ( )
+ enableMonitoring ( )
+ terminateMonitoring ( )
+ isMonitoringEnabled ( ) : boolean
+ isMonitoringTerminated ( ) : boolean
+ activateProbe (pattern : String) : boolean
+ deactivateProbe (pattern : String) : boolean
+ isProbeActivated (pattern : String) : boolean
1
1
- writer
«interface»
IMonitoringWriter
(from kieker::monitoring::writer)
1
1
- timeSource
«interface»
ITimeSource
(from kieker::monitoring::timer)
+ getTime ( ) : long

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

As depicted by the class diagram in Figure 3.1, it provides methods for

- Creating `IMonitoringController` instances (Section 3.1.1),
- Logging monitoring records with the configured monitoring writer (Section 3.1.2),
- Retrieving the current time via the configured time source (Section 3.1.3),
- Scheduling and removing period samplers (Section 3.1.4),
- Controlling the monitoring state (Section 3.1.5), and
- Activating and deactivating probes at runtime 3.1.6.

3.1.1 Creating `MonitoringController` Instances

The `MonitoringController` provides two different static methods for retrieving instances of `IMonitoringController`:

1. The method *MonitoringController.getInstance()* returns a singleton instance. As described in Section 3.2, the configuration is read from a properties file that has

been passed to the JVM, is located in the classpath, or conforms to the default configuration (Appendix E).

2. The method *MonitoringController.createInstance(Configuration config)* can be utilized to create an instance that is configured according to the passed `Configuration` object, as described in Section 3.2.

3.1.2 Logging Monitoring Records

Monitoring records are sent to the configured monitoring writers by passing these records, in form of `IMonitoringRecord` objects, to the `MonitoringController`'s *newMonitoringRecord* method. Note, that this is not the case if monitoring is disabled or terminated (Section 3.1.5).

3.1.3 Retrieving the Current Time and Using Custom Time Sources

The current time is maintained by a so-called time source. The `MonitoringController`'s method *getTimeSource* returns an `ITimeSource` whose method *getTime* returns a timestamp in nanoseconds. Kieker's default time source, `SystemNanoTimer`, returns the current system time as the number of nanoseconds elapsed since 1 Jan 1970 00:00 UTC. The easiest way to use a custom time source is to extend the `AbstractTimeSource` and to implement the method *getTime()*. Custom time sources make sense, for example, in simulations where not the current system time but the current simulation time is relevant. The configuration needs to be adjusted to use a custom time source class.

3.1.4 Scheduling and Removing Periodic Samplers

For certain applications, it is required to monitor runtime data periodically, e.g., the utilization of system resources such as CPUs. For this purpose, Kieker supports special monitoring probes, called samplers. Samplers must implement the interface `ISampler` which includes a single method *sample(IMonitoringController monitoringController)*. This method is called in periodic time steps, as specified by the `MonitoringController`'s registration function *schedulePeriodicSampler*. Periodic samplers can be stopped by calling the `MonitoringController`'s method *removeScheduledSampler*.

Listing 3.1 shows the *sample* method of the `MemSwapUsageSampler` which can be used to monitor memory and swap usage employing the Sigar library [1]. Likewise to other monitoring probes described in this user guide (see for example Sections 3.4 and 2.3), it collects the data of interest (lines 61–62), creates a monitoring record (lines 63–66), and passes this monitoring record to the monitoring controller (line 67). The available Sigar-based samplers for monitoring system-level monitoring data, such as CPU and memory usage, are discussed in Appendix D.

```java
  public void sample(final IMonitoringController monitoringCtr) throws
      SigarException {
    if (!monitoringCtr.isMonitoringEnabled()) {
      return;
    }
    if (!monitoringCtr.isProbeActivated(SignatureFactory.
        createMemSwapSignature())) {
      return;
    }

    final Mem mem = this.sigar.getMem();
    final Swap swap = this.sigar.getSwap();
    final MemSwapUsageRecord r = new MemSwapUsageRecord(
        monitoringCtr.getTimeSource().getTime(), monitoringCtr.getHostname(),
        mem.getTotal(), mem.getActualUsed(), mem.getActualFree(),
        swap.getTotal(), swap.getUsed(), swap.getFree());
    monitoringCtr.newMonitoringRecord(r);
  }
}
```

Listing 3.1: Method *sample* from MemSwapUsageSampler.java

3.1.5 Controlling the Monitoring State

The `MonitoringController` provides methods to temporarily enable or disable monitoring (*enableMonitoring*/*disableMonitoring*), as well as to terminate monitoring permanently (*terminateMonitoring*). The current state can be requested by calling the methods *isMonitoringEnabled* and *isMonitoringTerminated*. If monitoring is not enabled (i.e., disabled or terminated), no monitoring records retrieved via the method *newMonitoringRecord* are passed to the monitoring writer. Also, probes should be passive or return immediately with respect to the return value of the method *isMonitoringEnabled*. Note, that once the `MonitoringController` is terminated, it cannot be enabled later on.

3.1.6 Adaptive Monitoring

The `MonitoringController` provides an API to activate and deactivate probes at runtime. By passing a method signature—e.g., `"public void Bookstore.getBook()"`—to the method *isProbeActivated*, probes can check whether or not monitoring for the method with the given signature is active. Monitoring can be (de)activated for single signature *patterns*—e.g., `"public void Bookstore.*(..)"`— via the methods *activateProbe* and *deactivateProbe*. The current list of (de)activated patterns can be obtained via the method *getProbePatternList*. The entire list can be replaced using the method *setProbePatternList*. Alternatively, a file with include and exclude patterns can be used.

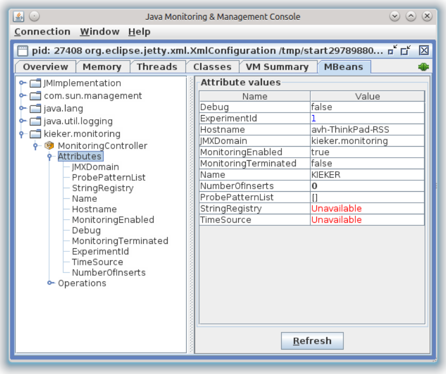
This file can be polled in regular intervals. A default configuration file, including a description of the pattern syntax, is provided by the file `kieker.monitoring.adaptiveMonitoring.conf` in the `META-INF/` directory of the binary release.

With the same mechanism arbitrary probes can be controlled. The syntax is also included in the above file. For example, Kieker's probes for CPU and memory make use of this mechanism.

By default, Kieker's adaptive monitoring feature is deactivated. It can be enabled by setting the value of the configuration property `kieker.monitoring.adaptiveMonitoring.enabled` in the `kieker.monitoring.properties` file to *true*. Additional properties to configure the adaptive monitoring are included in the file `kieker.monitoring.properties`, e.g., the location of the afore-mentioned file with include/exclude patterns and the polling interval for this file.

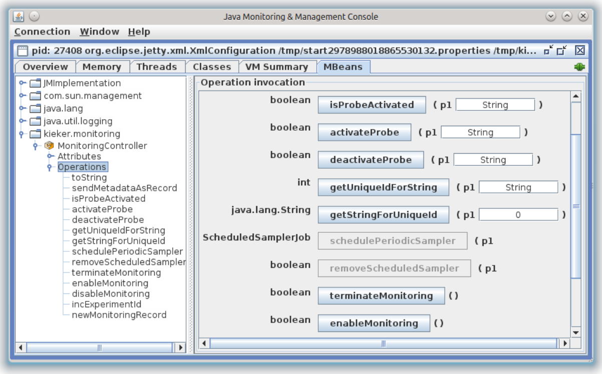
3.1.7 JMX MBean Access to `MonitoringController`

The `MonitoringController`'s interface methods (see Figure 3.1) can be accessed as a JMX MBean. For example, this allows to control the monitoring state using the methods described in the previous Section 3.1.5. As a JMX-compliant graphical client that is included in the JDK, `jconsole` is probably the easiest way to get started. Just keep in mind to add Kieker to the classpath when calling `jconsole` so that the MBean works correctly. Figure 3.2 shows two screenshots of the MBean access using `jconsole`.



Name	Value
Debug	false
ExperimentId	1
Hostname	avh-ThinkPad-RSS
JMXDomain	kieker.monitoring
MonitoringEnabled	true
MonitoringTerminated	false
Name	KIEKER
NumberOfInserts	0
ProbePatternList	[]
StringRegistry	Unavailable
TimeSource	Unavailable

(a) Attributes



```
boolean               isProbeActivated         ( p1 String )
boolean               activateProbe            ( p1 String )
boolean               deactivateProbe          ( p1 String )
int                   getUniqueIdForString     ( p1 String )
java.lang.String      getStringForUniqueId     ( p1 0 )
ScheduledSamplerJob   schedulePeriodicSampler  ( p1
boolean               removeScheduledSampler   ( p1
boolean               terminateMonitoring      ()
boolean               enableMonitoring         ()
```

(b) Operations

Figure 3.2: Screenshots of the `jconsole` JMX client accessing the `MonitoringController`'s attributes and operations via the MBean interface.

In order to enable JMX MBean access to the `MonitoringController`, the corresponding configuration properties must be set to *true* (listing below). The `kieker.monitoring.properties` includes additional JMX-related configuration properties.

```
## Whether any JMX functionality is available
kieker.monitoring.jmx=true
...

## Enable/Disable the MonitoringController MBean
kieker.monitoring.jmx.MonitoringController=true
...
```

For remote access to the server, set *kieker.monitoring.jmx.remote=true*. In this case it is recommended to set *com.sun.management.jmxremote.authenticate=true* as well. More information can be found on Oracle's JMX Technology Home Page [6].

3.2 Kieker.Monitoring Configuration

Kieker.Monitoring instances can be configured by properties files, `Configuration` objects, and by passing property values as JVM arguments. If no configuration is specified, a default configuration is being used. Appendix E lists this default configuration with a documentation of all available properties. The default configuration properties file, which can be used as a template for custom configurations, is provided by the file `kieker.monitoring.properties` in the directory `kieker-1.9/META-INF/` of the binary release (see Section 2.1).

Configurations for Singleton Instances

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 [...]
```

Alternatively, a file named `kieker.monitoring.properties` can be placed in a directory called `META-INF/` located in the classpath. The available configuration properties can also be passed as JVM arguments, e.g., `−Dkieker.monitoring.enabled=true`.

Configurations for Non-Singleton Instances

The class `Configuration` provides factory methods to create `Configuration` objects according to the default configuration or loaded from a specified properties file: *createDefaultConfiguration*, *createConfigurationFromFile*, and *createSingletonConfiguration*. Note, that JVM parameters are only evaluated when using the factory method *createSingletonConfiguration*. The returned `Configuration` objects can be adjusted by setting single property values using the method *setProperty*.

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 3.1), serialized and deserialized by a monitoring writer (Section 3.5) and a monitoring reader, and provided to analysis filters (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 3.3. The methods *initFromArray*, *toArray*, *getValueTypes* are used for serialization and deserialization of the monitoring data contained in the record. Alternatively—in order to support the definition of immutable record types—the marker interface `IMonitoringRecord.Factory` needs to be implemented, requiring the implementation of (i) the *toArray* method (as before), (ii) a constructor accepting a values array, and (iii) a public static *TYPES* field. 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 3.3) which already implements the methods to maintain the logging timestamp.

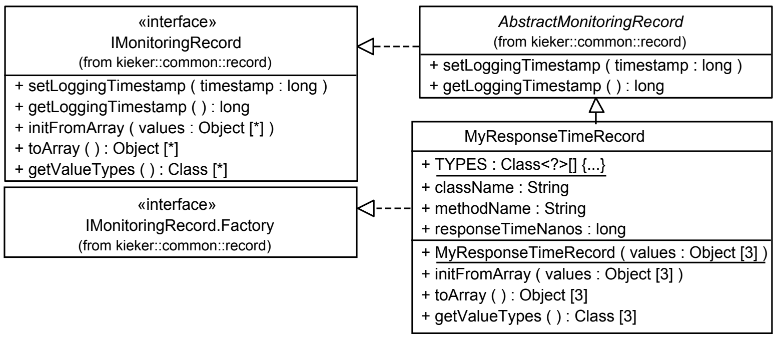

«interface»
IMonitoringRecord
(from kieker::common::record)
+ setLoggingTimestamp (timestamp : long)
+ getLoggingTimestamp ( ) : long
+ initFromArray (values : Object [*])
+ toArray ( ) : Object [*]
+ getValueTypes ( ) : Class [*]
«interface»
IMonitoringRecord.Factory
(from kieker::common::record)
AbstractMonitoringRecord
(from kieker::common::record)
+ setLoggingTimestamp (timestamp : long)
+ getLoggingTimestamp ( ) : long
MyResponseTimeRecord
+ TYPES : Class<?>[] {...}
+ className : String
+ methodName : String
+ responseTimeNanos : long
+ MyResponseTimeRecord (values : Object [3])
+ initFromArray (values : Object [3])
+ toArray ( ) : Object [3]
+ getValueTypes ( ) : Class [3]

Figure 3.3: Class diagram with the `IMonitoringRecord` and `IMonitoringRecord.Factory` interfaces, the abstract class `AbstractMonitoringRecord`, and a custom monitoring record type `MyResponseTimeRecord`

In order to use the abstract class for implementing your own monitoring record type, you need to:

1. Create a class that extends `AbstractMonitoringRecord`
2. and
   a) Override the methods *initFromArray*, *toArray*, *getValueTypes*
   b) For immutable record types: implement `IMonitoringRecord.Factory`, a constructor with a single `Object[]` argument, and a public static *TYPES* field. In this case, *initFromArray* (which is not called by the framework then) should throw an `UnsupportedOperationException`.

The class `MyResponseTimeRecord`, shown in the class diagram in Figure 3.3 and in Listing 3.2, is an example of a custom monitoring record type that can be used to monitor response times of method executions. Implementing `IMonitoringRecord.Factory`, `MyResponseTimeRecord` is an immutable type, i.e., it includes only final fields.

```java
public class MyResponseTimeRecord extends AbstractMonitoringRecord implements
    IMonitoringRecord.Factory, IMonitoringRecord.BinaryFactory {
  public static final int SIZE = (2 * TYPE_SIZE_STRING) + TYPE_SIZE_LONG;
  public static final Class<?>[] TYPES = { String.class, String.class, long.
      class, };

  private static final long serialVersionUID = 7837873751833770201L;

  // Attributes storing the actual monitoring data:
  private final String className;
  private final String methodName;
  private final long responseTimeNanos;

  public MyResponseTimeRecord(final String clazz, final String method, final
      long rtNano) {
    this.className = clazz;
    this.methodName = method;
    this.responseTimeNanos = rtNano;
  }

  public MyResponseTimeRecord(final Object[] values) {
    AbstractMonitoringRecord.checkArray(values, MyResponseTimeRecord.TYPES);

    this.className = (String) values[0];
    this.methodName = (String) values[1];
    this.responseTimeNanos = (Long) values[2];
  }

```

```java
public MyResponseTimeRecord(final ByteBuffer buffer, final IRegistry<String>
     stringRegistry) throws BufferUnderflowException {
  this.className = stringRegistry.get(buffer.getInt());
  this.methodName = stringRegistry.get(buffer.getInt());
  this.responseTimeNanos = buffer.getLong();
}

@Deprecated
// Will not be used because the record implements IMonitoringRecord.Factory
public final void initFromArray(final Object[] values) {
  throw new UnsupportedOperationException();
}

@Deprecated
// Will not be used because the record implements IMonitoringRecord.
    BinaryFactory
public final void initFromBytes(final ByteBuffer buffer, final IRegistry<
    String> stringRegistry) throws BufferUnderflowException {
  throw new UnsupportedOperationException();
}

public Object[] toArray() {
  return new Object[] { this.getClassName(), this.getMethodName(), this.
      getResponseTimeNanos(), };
}

public void writeBytes(final ByteBuffer buffer, final IRegistry<String>
    stringRegistry) throws BufferOverflowException {
  buffer.putInt(stringRegistry.get(this.getClassName()));
  buffer.putInt(stringRegistry.get(this.getMethodName()));
  buffer.putLong(this.getResponseTimeNanos());
}

public Class<?>[] getValueTypes() {
  return MyResponseTimeRecord.TYPES;
}

public int getSize() {
  return SIZE;
}

public final String getClassName() {
  return this.className;
}

public final String getMethodName() {
    return this.methodName;
  }

  public final long getResponseTimeNanos() {
    return this.responseTimeNanos;
  }
}
```

Listing 3.2: MyResponseTimeRecord.java

3.4 Monitoring Probes

The probes are responsible for collecting the monitoring data and passing it to the monitoring controller. In Chapter 2.3, we have already demonstrated how to manually instrument a Java application. Listing 3.3 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 = MONITORING_CONTROLLER.getTimeSource().getTime();
      this.catalog.getBook(false);
      final long tout = MONITORING_CONTROLLER.getTimeSource().getTime();
      // Create a new record and set values
      final MyResponseTimeRecord e = new MyResponseTimeRecord(
          "mySimpleKiekerExample.bookstoreTracing.Catalog", "getBook(..)", tout
               - tin);
      // Pass the record to the monitoring controller
      MONITORING_CONTROLLER.newMonitoringRecord(e);
```

Listing 3.3: Excerpt from Bookstore.java

In order to avoid multiple calls to the *getInstance* method of the `MonitoringController` class, singleton instances should be stored in a final static variable, as shown in Listing 3.4.

```
  private static final IMonitoringController MONITORING_CONTROLLER =
      MonitoringController.getInstance();
```

Listing 3.4: 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 [7], Spring [8], and Apache CXF [9] provide interception/AOP [2] interfaces which are well-suited to implement monitoring probes. AspectJ [10] 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 writers serialize monitoring records to the monitoring log/stream 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 3.4 shows the monitoring writers already implemented in Kieker.Monitoring. The available properties for the included writers are well-documented in the example configuration file (see Appendix E).

Different writers can be used to store monitoring records to filesystems and databases respectively (e.g., `AsyncFsWriter`, `SyncFsWriter`, `AsyncDbWriter`, and `SyncDbWriter`). 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 that has already been used in Section 2.3. Please note that the database writers are currently in a prototype stage and that they should be used with care. The `PrintStreamWriter` simply sends the String representation of incoming records to the standard output or standard error streams, which can be helpful for debugging purposes.

The `AsyncJMSWriter` and `JMXWriter` write records to a JMS (Java Messaging Service [5]) queue and JMX (Java Management Extensions [6]) queue respectively. The `PipeWriter` allows to pass records via in-memory record streams (named pipes). These writers allow 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 more detailed description of how to use the `AsyncJMSWriter` can be found in Appendix C.

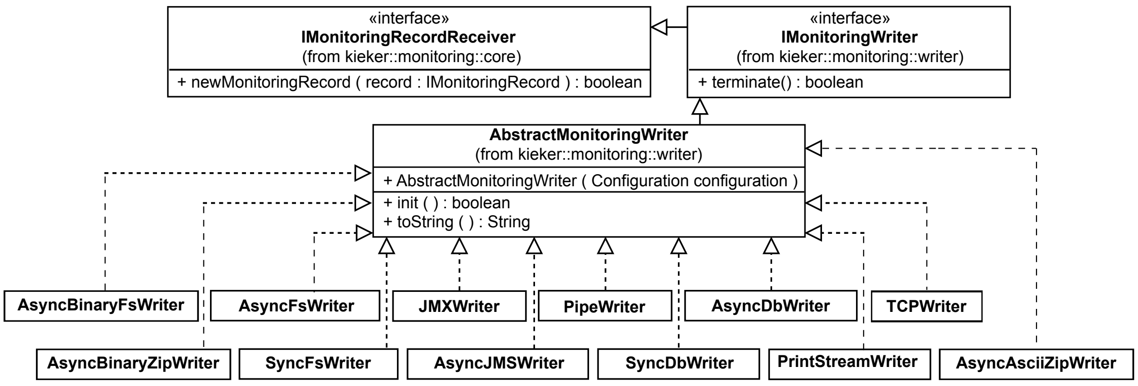


Figure 3.4: Interface `IMonitoringWriter` and the implementing classes

Listing 3.5 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 F.1.

```java
public class MyPipeWriter extends AbstractMonitoringWriter {

  public static final String CONFIG_PROPERTY_NAME_PIPE_NAME =
      MyPipeWriter.class.getName() + ".pipeName";

  private volatile MyPipe pipe;
  private final String pipeName;

  public MyPipeWriter(final Configuration configuration) {
    super(configuration);
    this.pipeName =
        configuration.getStringProperty(CONFIG_PROPERTY_NAME_PIPE_NAME);
  }

  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(), record.getClass()));
    } catch (final InterruptedException e) {
      return false; // signal error
    }
    return true;
  }

  @Override
  protected Configuration getDefaultConfiguration() {
    final Configuration configuration = new Configuration(super.
        getDefaultConfiguration());
    configuration.setProperty(CONFIG_PROPERTY_NAME_PIPE_NAME, "kieker-pipe");
    return configuration;
  }

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

  public void terminate() {
    // nothing to do
  }
}
```

Listing 3.5: MyPipeWriter.java

The monitoring writer to be used is selected by the Kieker.Monitoring configuration property (Section 3) *kieker.monitoring.writer*. Writer-specific configuration properties can be provided by properties prefixed by the fully-qualified writer classname. Listing 3.6 demonstrates how to use the custom writer `MyPipeWriter` defined above. In this example, the pipe name is passed as the property value *pipeName*.

```
kieker.monitoring.writer=kieker.examples.userguide.ch3and4bookstore.MyPipeWriter
kieker.examples.userguide.ch3and4bookstore.MyPipeWriter.pipeName=somePipe
```

Listing 3.6: Configuration of the custom writer `MyPipeWriter`

As the data structure of this kind of monitoring stream, we created a class `PipeData` in order to demonstrate the use of the *toArray* and *initFromArray* (in Section 4.2.3) methods. A `PipeData` object holds a logging timestamp and an `Object` array containing the serialized record data. Appendix F.1 includes a source code listing of this class. Alternatively, we could have used `IMonitoringRecord` as the data structure used by the pipe. This is the way, Kieker's `PipeWriter` works.

4 Kieker.Analysis Component

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

4.1 Pipe-and-Filter Framework and Included Plugins

Kieker.Analysis provides a framework to define and execute pipe-and-filter architectures of analysis plugins, i.e., monitoring readers and analysis filters, as well as repositories. This section describes how to use and develop readers, filters, and repositories. The description is based on the example pipe-and-filter architecture shown in Figure 4.1. The custom monitoring reader `MyPipeReader`, which corresponds to the writer developed in Section 3.5, sends records to the connected custom filter `MyResponseTimeFilter`. This filter accepts only events of the record type `MyResponseTimeRecord`, developed in Section 3.3. The `MyResponseTimeFilter` classifies incoming `MyResponseTimeRecords` based on whether they satisfy or exceed a configured threshold and passes them to the respective output ports, *validResponseTimes* or *invalidResponseTimes*. Two instances of a second custom filter, `MyResponseTimeOutputPrinter`, print the received records to the standard output stream.
Figure 4.2 shows the class diagram with the important Kieker.Analysis classes and their relationships. Note that only the most important methods are included. An analysis with Kieker.Analysis is set up and executed employing the class `AnalysisController`.

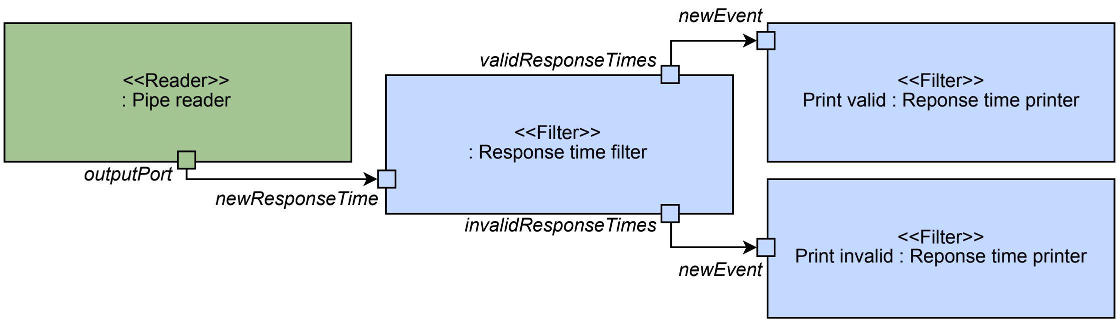


Figure 4.1: Example pipe-and-filter configuration

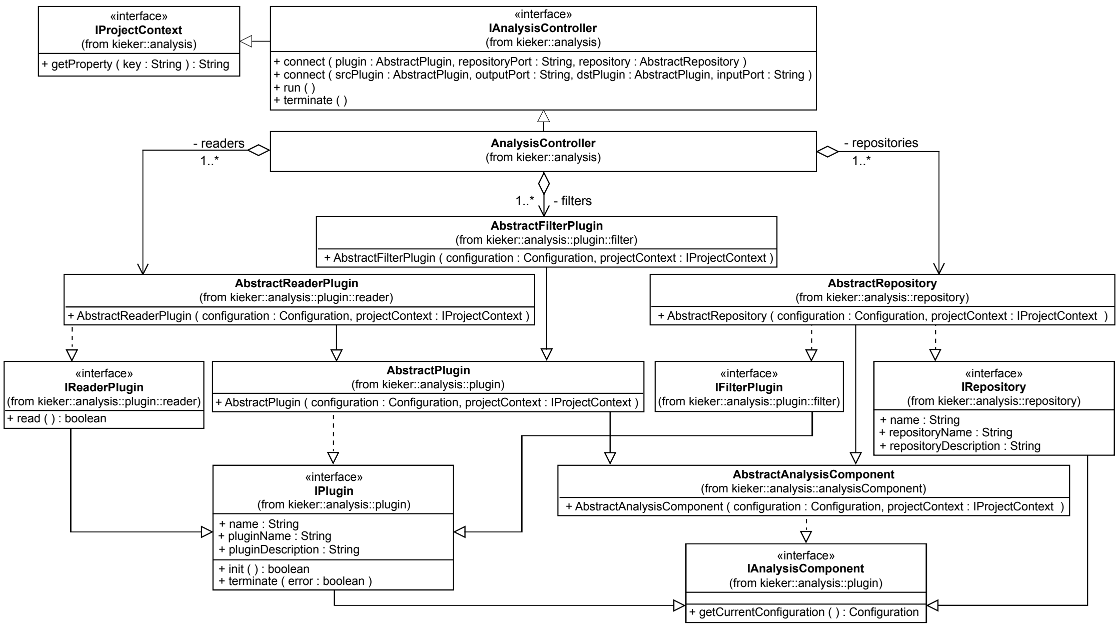

«interface»
IProjectContext
(from kieker::analysis)
+ getProperty (key : String) : String
«interface»
IAnalysisController
(from kieker::analysis)
+ connect (plugin : AbstractPlugin, repositoryPort : String, repository : AbstractRepository)
+ connect (srcPlugin : AbstractPlugin, outputPort : String, dstPlugin : AbstractPlugin, inputPort : String)
+ run ( )
+ terminate ( )
- readers
1..*
AnalysisController
(from kieker::analysis)
- repositories
1..*
1..*
- filters
AbstractFilterPlugin
(from kieker::analysis::plugin::filter)
+ AbstractFilterPlugin (configuration : Configuration, projectContext : IProjectContext)
AbstractReaderPlugin
(from kieker::analysis::plugin::reader)
+ AbstractReaderPlugin (configuration : Configuration, projectContext : IProjectContext)
AbstractRepository
(from kieker::analysis::repository)
+ AbstractRepository (configuration : Configuration, projectContext : IProjectContext)
«interface»
IReaderPlugin
(from kieker::analysis::plugin::reader)
+ read ( ) : boolean
AbstractPlugin
(from kieker::analysis::plugin)
+ AbstractPlugin (configuration : Configuration, projectContext : IProjectContext)
«interface»
IFilterPlugin
(from kieker::analysis::plugin::filter)
«interface»
IRepository
(from kieker::analysis::repository)
+ name : String
+ repositoryName : String
+ repositoryDescription : String
«interface»
IPlugin
(from kieker::analysis::plugin)
+ name : String
+ pluginName : String
+ pluginDescription : String
+ init ( ) : boolean
+ terminate (error : boolean)
AbstractAnalysisComponent
(from kieker::analysis::analysisComponent)
+ AbstractAnalysisComponent (configuration : Configuration, projectContext : IProjectContext)
«interface»
IAnalysisComponent
(from kieker::analysis::plugin)
+ getCurrentConfiguration ( ) : Configuration

Figure 4.2: Class diagram showing important Kieker.Analysis types and their relationship

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

1. Creating an instance of the `AnalysisController` class
2. Creating monitoring readers, filters, and repositories.
3. Connecting plugins to other plugins and to repositories (*connect*)
4. Starting the analysis instance (*run*).

On invocation of the *run* method, the `AnalysisController` calls the *init* method of all filter plugins allowing them to initialize. Then, it starts the configured monitoring readers by calling its *read* method. Plugins send data via their output ports to connected input ports of other plugins. Being the source in a pipe-and-filter architecture, readers don't have input ports. Plugins can be connected to repositories, which may provide shared services, such as managed access to a common architectural model of the analyzed system. As soon as all readers have returned from the execution of their *read* methods, the method *terminate* of each registered plugin is called by the `AnalysisController`. Kieker.Analysis configurations can be saved to a `.kax` file by calling the `AnalysisController`'s *saveToFile* method. The `AnalysisController` provides a constructor which accepts the file system location of a `.kax` file to load the configuration from. See Appendix A.3 and A.4 for included tools/scripts which execute and visualize `.kax` files. In order to support the asynchronous execution of the `AnalysisController` instance, we provide the `AnalysisControllerThread` class.

4.1.1 Programmatic Creation of Pipe-and-Filter Architectures

To give a first impression of the programmatic instantiation, configuration, and connection of plugins, Listing 4.1 demonstrates this procedure for the example, using `MyPipeReader` and `MyResponseTimeFilter`, according to Figure 4.1.

The configuration for the `MyPipeReader` is created in lines 50–51. Using this configuration, the reader is created in line 52. Similarly, lines 55–61 initialize the `MyResponseTimeFilter`. The reader's output is connected to the filter's input in line 62. The entire programmatic creation of the pipe-and-filter architecture shown in Figure 4.1, can be found in the example file `Starter.java`.

```
47    final IAnalysisController analysisController = new AnalysisController();
48
49    // Configure and register the reader
50    final Configuration readerConfig = new Configuration();
51    readerConfig.setProperty(MyPipeReader.CONFIG_PROPERTY_NAME_PIPE_NAME, "
          somePipe");
52    final MyPipeReader reader = new MyPipeReader(readerConfig,
          analysisController);
53
54    // Configure, register, and connect the response time filter
55    final Configuration filterConfig = new Configuration();
56    final long rtThresholdNanos =
57       TimeUnit.NANOSECONDS.convert(1900, TimeUnit.MICROSECONDS);
58    filterConfig.setProperty( // configure threshold of 1.9 milliseconds:
59       MyResponseTimeFilter.CONFIG_PROPERTY_NAME_TS_NANOS,
60       Long.toString(rtThresholdNanos));
61    final MyResponseTimeFilter filter = new MyResponseTimeFilter(filterConfig,
          analysisController);
62    analysisController.connect(reader, MyPipeReader.OUTPUT_PORT_NAME, filter,
          MyResponseTimeFilter.INPUT_PORT_NAME_RESPONSE_TIMES);
```

Listing 4.1: Initializing and connecting the example reader and filter (Starter.java)

4.1.2 Monitoring Reader Plugins

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 4.3. The `FSReader` has already been used in Section 2.4. A brief description of how to use the `JMSReader` can be found in Appendix C. Please note that the database reader (`DBReader`) is currently in a prototype stage and that it should be used with care. Like each plugin, readers are configured via properties, as used in Section 4.1.1 and detailed in Section 4.2.1.

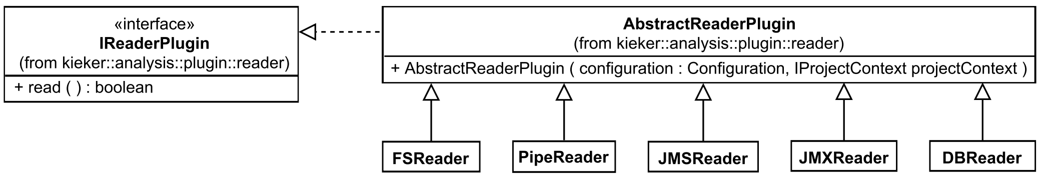

«interface»
IReaderPlugin
(from kieker::analysis::plugin::reader)
+ read ( ) : boolean
AbstractReaderPlugin
(from kieker::analysis::plugin::reader)
+ AbstractReaderPlugin (configuration : Configuration, IProjectContext projectContext)
FSReader
PipeReader
JMSReader
JMXReader
DBReader

Figure 4.3: Monitoring reader plugins included with Kieker

4.1.3 Filter Plugins

Filter plugins receive events (Java objects) via input ports from other plugins and implement analyses or visualizations based on these events. Kieker already includes some basic filter plugins. For example, the `CountingFilter` and `TeeFilter` forward incoming events to their output ports. The `CountingFilter` additionally provides the current number of received records via a second output port. The `TeeFilter` additionally prints incoming events to an output stream, which may be the standard output, standard error, a logger, or a file. A `TimestampFilter` and a `TypeFilter` filter incoming records by timestamp and by type, respectively. A `TraceIdFilter` filters incoming trace events (e.g., `OperationExecutionRecords`, see Section 2) by trace ID. Additional filters for trace analysis, architecture reconstruction and visualization are included as part of the Kieker.TraceAnalysis tool, presented in Chapter 5. Like each plugin, filters are configured via properties, as used in Section 4.1.1 and detailed in Section 4.2.1.

4.1.4 Repositories

Currently, Kieker includes a single repository, `SystemModelRepository`, which is used by the Kieker.TraceAnalysis filters to update and query a component-based system model representing architectural entities and structures discovered while processing the incoming monitoring data. The development and use of repositories is detailed in Section 4.2.5.

When using components of the Kieker.TraceAnalysis, make sure that write access to the `SystemModelRepository` is only triggered by readers. Some filters are terminated after the readers and expect the repository to be in a completed state.

4.2 Developing Analysis Plugins and Repositories

When implementing analysis plugins (i.e., readers or filters) and repositories, the classes `AbstractReaderPlugin`, `AbstractFilterPlugin`, or, respectively, `AbstractRepository` need to be extended (Figure 4.2). Section 4.2.1 describes how plugins and repositories can be configured via properties. Section 4.2.2 describes how to declare meta-information for plugins using dedicated annotations. Specific information on the development of custom filters, readers, and repositories are given in Sections 4.2.4–4.2.5.

4.2.1 Configuration

According to the configuration of the Kieker.Monitoring components (see Section 3.2), plugins and repositories are configured via `Configuration` objects. Classes must provide a public constructor, accepting a `Configuration` and an `IProjectContext` (normally the `IAnalysisController` instance) object as its only arguments. It is important to invoke the constructor of the super class. The configuration properties accepted by a plugin or repository should be provided via `public static` constants with prefix *CONFIG_PROPERTY_NAME_* in order to ease the programmatic initialization of plugins (Section 4.1.1). For the example filter `MyResponseTimeFilter`, Listing 4.2 shows the constructor, the configuration property, and the corresponding member value obtained from the configuration.

```java
public static final String CONFIG_PROPERTY_NAME_TS_NANOS = "thresholdNanos";

private final long rtThresholdNanos; // the configured threshold for this
    filter instance

public MyResponseTimeFilter(final Configuration configuration, final
    IProjectContext projectContext) {
  super(configuration, projectContext);

  this.rtThresholdNanos = configuration.getLongProperty(
      CONFIG_PROPERTY_NAME_TS_NANOS);
}
```

Listing 4.2: Plugin constructor accepting a `Configuration` and an `IProjectContext` object

Additionally, the current configuration must be provided via the method *getCurrentConfiguration*. Please note that the returned configuration should be sufficient to initialize the plugin or repository via the mentioned constructor. The `AnalysisController` uses the *getCurrentConfiguration* to save the pipe-and-filter configuration. Listing 4.3 shows how the methods are implemented for the example filter `MyResponseTimeFilter`.

```java
public Configuration getCurrentConfiguration() {
  final Configuration configuration = new Configuration();
  configuration.setProperty(CONFIG_PROPERTY_NAME_TS_NANOS,
      Long.toString(this.rtThresholdNanos));
  return configuration;
}
```

Listing 4.3: Plugin returning its current configuration

The declaration of the available properties and their default values within a plugin is shown in section 4.2.2, as this is done with annotations.

4.2.2 @Plugin Annotation and Output Ports

The `@Plugin` class annotation is used to define a plugin name, a description, and the lists of output ports and configuration properties with default values. Listing 4.4 shows the `@Plugin` annotation for the example filter.

If the `@Plugin` annotation is not present for a plugin, the *name* defaults to the plugin's (simple) classname, the *description* defaults to the empty string, and the list of output ports is empty. These default values are also used in case a respective attribute is omitted. Note that the name is not required to be a unique among filters; it is simply used for descriptive purposes, such as in Figure 4.1.

Output ports are specified using the nested `@OutputPort` annotation. In addition to a name and a description for the output port, a list of event types can be specified. Note that in this case, the name is mandatory and must be unique for a plugin, as it is used for connecting input and output ports. The list of event types defaults to a list including only `Object.class`. The output port names should be provided as a `public static` constant with prefix *OUTPUT_PORT_NAME_*, in order to ease the programmatic connection of readers and filters, as described in Section 4.1.1. Repositories required by filters are also specified as part of the `@Plugin` annotation. This is detailed in Section 4.2.5.

```java
@Plugin(
    name = "Response time filter",
    description = "Filters incoming response times based on a threshold",
    outputPorts = {
      @OutputPort(name = MyResponseTimeFilter.OUTPUT_PORT_NAME_RT_VALID,
          description = "Outputs response times satisfying the threshold",
          eventTypes = { MyResponseTimeRecord.class }),
      @OutputPort(name = MyResponseTimeFilter.OUTPUT_PORT_NAME_RT_EXCEED,
          description = "Outputs response times exceeding the threshold",
          eventTypes = { MyResponseTimeRecord.class }) },
    configuration = {
      @Property(name = MyResponseTimeFilter.CONFIG_PROPERTY_NAME_TS_NANOS,
          defaultValue = "1000000")
    })
public class MyResponseTimeFilter extends AbstractFilterPlugin {
```

Listing 4.4: @Plugin annotation for the example plugin `MyResponseTimeFilter`

Plugins can send events to their output ports by calling the *deliver* method provided by the super class. The method expects the output port name and the event to be sent as arguments. Listing 4.5 shows how the example filter plugin `MyResponseTimeFilter` delivers records to its two output ports declared in the `@Plugin` annotation.

```java
    if (rtRecord.getResponseTimeNanos() > this.rtThresholdNanos) {
      super.deliver(OUTPUT_PORT_NAME_RT_EXCEED, rtRecord);
    } else {
      super.deliver(OUTPUT_PORT_NAME_RT_VALID, rtRecord);
    }
```

Listing 4.5: Plugin sending events to output ports

Listing 4.4 shows also how the properties are declared. Using the `@Property` annotation, it is possible to declare the existing properties. Each property has a default value which should be sufficient to initialize the plugin.

The `@Plugin` annotation (as well as the later introduced `@Repository` annotation) contains furthermore the two fields *dependencies* and *programmaticOnly*. The first one offers the possibility to give a description of the needed dependencies for a plugin (other libraries e.g.). The latter marks whether the current plugin (or repository) is for programmatic purposes only, i.e., they are of little use in graphical analysis tools.

4.2.3 Developing Monitoring Reader Plugins

Custom readers must extend the class `AbstractReaderPlugin` (see Figure 4.2), and implement the methods *init*, *read*, and *terminate*, which are called by the `AnalysisController` to trigger the reader's initialization, reading, and termination. Like each plugin (Section 4.2.1), readers are configured via a constructor accepting a `Configuration` and an `IProjectContext` object as its only arguments; they must provide the current configuration via the implemented *getCurrentConfiguration* method. Readers start reading on invocation of the *read* method, providing the obtained records to connected filters via the output port(s) declared in the `@Plugin` annotation (Section 4.2.2). The *read* method should be implemented synchronously, i.e., it should return after reading is finished or has been aborted via an invocation of the *terminate* method.

```java
@Plugin(
    name = "Pipe reader",
    description = "Reads records from a configured pipe",
    outputPorts = { @OutputPort(
        name = MyPipeReader.OUTPUT_PORT_NAME,
        description = "Outputs any received record",
        eventTypes = { IMonitoringRecord.class })
    },
    configuration = { @Property(
        name = MyPipeReader.CONFIG_PROPERTY_NAME_PIPE_NAME,
        defaultValue = "kieker-pipe")
    })
public class MyPipeReader extends AbstractReaderPlugin {
```

Listing 4.6: @Plugin annotation for the example reader

Listing 4.6 shows the `@Plugin` annotation of the example reader `MyPipeReader`. Reading monitoring records from the monitoring pipe introduced in the previous Chapter 3.5, the reader provides received monitoring records via its output port.
Listing 4.7 shows an excerpt of the `MyPipeReader`'s constructor. In this case, the reader reads the pipe name from the configuration and connects to the named pipe. Optionally, the reader can override the *init* method.

```
    this.pipeName = configuration.getStringProperty(MyPipeReader.
        CONFIG_PROPERTY_NAME_PIPE_NAME);

    try {
      this.pipe = MyNamedPipeManager.getInstance().acquirePipe(this.pipeName);
    } catch (final Exception ex) {
      this.log.error("Failed to acquire pipe '" + this.pipeName + "'", ex);
    }
```

Listing 4.7: Example reader's initialization in the constructor (excerpt)

Listing 4.8 shows the `MyPipeReader`'s *read* method. In this case, the reader polls the pipe for new records and forwards these to its output port.

```
  public boolean read() {
    try {
      // Wait max. 4 seconds for the next data.
      PipeData data = this.pipe.poll(4);
      while (data != null) {
        // Create new record, init from received array ...
        final IMonitoringRecord record = // throws MonitoringRecordException:
        AbstractMonitoringRecord.createFromArray(data.getRecordType(),
            data.getRecordData());
        record.setLoggingTimestamp(data.getLoggingTimestamp());
        // ...and delegate the task of delivering to the super class.
        super.deliver(MyPipeReader.OUTPUT_PORT_NAME, record);
        // next turn
        data = this.pipe.poll(4);
      }
    } catch (final Exception e) {
      return false; // signal error
    }
    return true;
  }
```

Listing 4.8: Example reader's *read* method

4.2.4 Developing Filter Plugins

Custom filters must extend the class `AbstractFilterPlugin`. In addition to providing meta information, including output ports, via the `@Plugin` annotation (Section 4.2.2), as well as implementing a constructor and the getter for handling the `Configuration` (Section 4.2.1), filters may override the methods *init* and *terminate*, implementing initialization and cleanup tasks. The `@Plugin` annotation of the example filter `MyResponseTimeFilter` was shown in Listing 4.4 already.

Filters receive events via methods marked with the `@InputPort` annotation. These methods must accept a single argument, which has to be a super type of the set of accepted event types declared in the respective `@InputPort` annotation's *eventTypes*. In addition to an optional *description*, each `@InputPort` must have a *name*, which is unique for this filter. The input port names should be provided as a `public static` constants with prefix *INPUT_PORT_NAME_*, in order to ease the programmatic connection of readers and filters, as described in Section 4.1.1. Listing 4.9 shows the declaration of the input port provided by the example plugin `MyResponseTimeFilter`. The body of this method was shown in Listing 4.5 already.

```java
@InputPort(
    name = MyResponseTimeFilter.INPUT_PORT_NAME_RESPONSE_TIMES,
    description = "Filter the given record depending on the response time",
    eventTypes = { MyResponseTimeRecord.class })
public void newResponseTime(final MyResponseTimeRecord rtRecord) {
```

Listing 4.9: @InputPort annotation for the example plugin's input method

4.2.5 Developing and Accessing Required Repositories

Custom repositories must extend the class `AbstractRepository`. The `@Repository` annotation is used to provide a *name* and a *description* for a repository type. Listing 4.10 shows the `@Repository` annotation of the `SystemModelRepository`, which is included in Kieker as part of the Kieker.TraceAnalysis tool.

```java
@Repository(
  name = "System model repository",
  description = "Model manager for Kieker's component model ")
public class SystemModelRepository extends AbstractRepository {
```

Listing 4.10: `@Respository` annotation of Kieker's `SystemModelRepository`

Plugins specify the list of required repositories in their `@Plugin` annotation. Repositories are connected to filter-provided repository ports. A plugin's repository ports are specified using the nested `@RepositoryPort` annotation, as depicted for a Kieker.TraceAnalysis filter in Listing 4.11. Like for input and output port names, this name must be unique for the plugin and should be provided as a `public static` constant with prefix *REPOSITORY_PORT_NAME_*, in order to ease the programmatic connection of repositories to readers and filters.

```java
/** The name of the repository port for the system model repository. */
public static final String REPOSITORY_PORT_NAME_SYSTEM_MODEL = "
    systemModelRepository";
```

Listing 4.11: Declaration of required repositories in the @Respository annotation

Plugins can access their connected repositories via the *getRepository* method provided by the super class, as shown in Listing 4.12.

```java
    this.systemEntityFactory = (SystemModelRepository)
        this.getRepository(REPOSITORY_PORT_NAME_SYSTEM_MODEL);
```

Listing 4.12: Accessing a repository within a plugin

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 [10], Java EE Servlet [7], Spring [8], and Apache CXF [9] 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, please refer to our technical report [11].

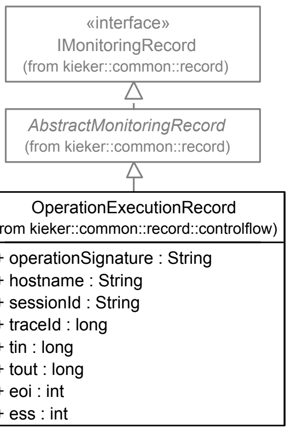


Figure 5.1: The class diagram of the operation execution record

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.

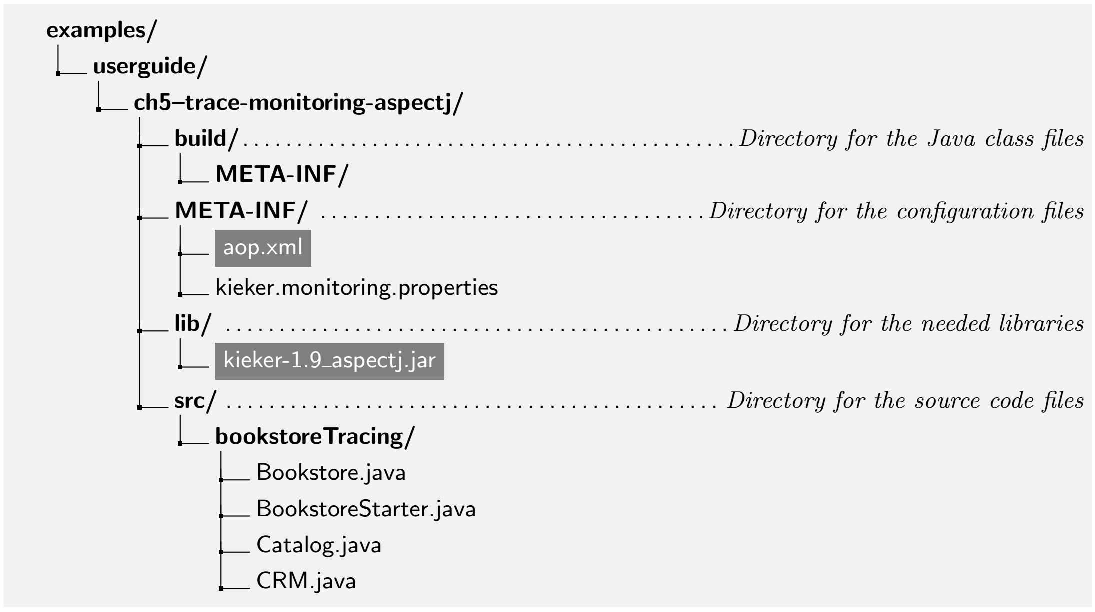
5.1 Monitoring Trace Information

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 [10] 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 example presented in this section, as well as a pre-compiled binary, 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
      │  └─ kieker.monitoring.properties
      ├─ lib/ ............................. Directory for the needed libraries
      │  └─ kieker-1.9_aspectj.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 `kieker-1.9_aspectj.jar` already includes 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`, for which a working sample file is provided in the example's `META-INF/` directory. Instead of registering the `kieker-1.9_aspectj.jar` as an agent to the JVM, the `aspectjweaver-1.7.4.jar` can be used. In this case, the `kieker-1.9.jar` needs to be added to the classpath.

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
public class Bookstore {

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

  @OperationExecutionMonitoringProbe
  public void searchBook() {
    this.catalog.getBook(false);
    this.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 instrument 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.

```xml
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="kieker.examples.userguide.ch5bookstore..*"/>
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 example package. AspectJ allows to use wild-cards for the definition of classes to include—e.g., `<include within="bookstoreTracing.Bookstore*"/>` to weave all classes with the prefix `Bookstore` located in a package `bookstoreTracing`.

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

Listings 5.3 and 5.4 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. Additionally, a `kieker.monitoring.properties` is copied to the `META-INF/` directory. This configuration file may be adjusted as desired (see Section 3.2).

```
▷ mkdir build
▷ mkdir build/META−INF
▷ javac src/kieker/examples/userguide/ch5bookstore/*.java \
        −d build/ −classpath lib/kieker-1.9_aspectj.jar

▷ cp META−INF/aop.xml build/META−INF/
▷ cp META−INF/kieker.monitoring.properties build/META−INF/

▷ java −javaagent:lib/kieker-1.9_aspectj.jar \
       −classpath build/  kieker.examples.userguide.ch5bookstore.BookstoreStarter
```

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

```
▷ mkdir build
▷ mkdir build\META−INF
▷ javac src\kieker\examples\userguide\ch5bookstore\*.java
        −d build −classpath lib\kieker-1.9_aspectj.jar

▷ copy META−INF\aop.xml build\META−INF\
▷ copy META−INF\kieker.monitoring.properties build\META−INF\

▷ java −javaagent:lib\kieker-1.9_aspectj.jar
       −classpath build\  kieker.examples.userguide.ch5bookstore.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 G.2.

Instrumentation without annotations 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="kieker.examples.userguide.ch5bookstore..*"/>
6      </weaver>
7
8      <aspects>
9        <aspect name="kieker.monitoring.probe.aspectj.operationExecution.
             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 5.3 and 5.4 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 classname in the include directives of the aop.xml.

5.1.2 Servlet Filters

The Java Servlet API [7] includes the `javax.servlet.Filter` interface. It can be used to implement interceptors for incoming HTTP requests. Kieker includes the probe `SessionAndTraceRegistrationFilter` which implements the `javax.servlet.Filter` interface. It initializes the session and trace information for incoming requests. If desired, it additionally creates an `OperationExecutionRecord` for each invocation of the filter and passes it to the `MonitoringController`.

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

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

```
<filter>
  <filter-name>sessionAndTraceRegistrationFilter</filter-name>
  <filter-class>kieker.monitoring.probe.servlet.SessionAndTraceRegistrationFilter</filter-class
      >
  <init-param>
    <param-name>logFilterExecution</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>sessionAndTraceRegistrationFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

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

5.1.3 Spring

The Spring framework [8] 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. If you are not using the `OperationExecutionWebRequestRegistrationInterceptor`, you should use one of the previously described Servlet filters to register session information for incoming requests (Section 5.1.2).

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 [9] allows to implement interceptors for web 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.

In order to use this tool, it is necessary to install two third-party 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).
3. **ps2pdf** The `ps2pdf` tool is used to convert ps files to pdf files.

Under Windows it is recommended to add the `bin/` directories of both tools to the “path” environment variable. It is also possible that the GNU PlotUtils are unable to process sequence diagrams. In this case it is recommended to use the Cygwin port of PlotUtils.

Once both programs 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.5.

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-Deployment-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-Deployment-Sequence-Diagrams
                     --plot-Call-Trees
```

Listing 5.8: Commands to produce the diagrams under Windows

The Windows `.bat` wrapper scripts (including `trace-analysis.bat`) must be executed from within the `bin/` directory.

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

```
out/
├── deploymentSequenceDiagram-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 deploymentSequenceDiagram−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.6 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/kieker-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 [11]. The output can be found in the directory `examples/userguide/ch5-trace-monitoring-aspectj/testdata/kieker-20100830-082225522-UTC-example-plots/`.

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 (Mon, 30 Aug 2010 08:21:38 +0000 (UTC));
    maxTout=1283156498820012272 (Mon, 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 'Entry' −−> 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 −−> 'Entry'>
```

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

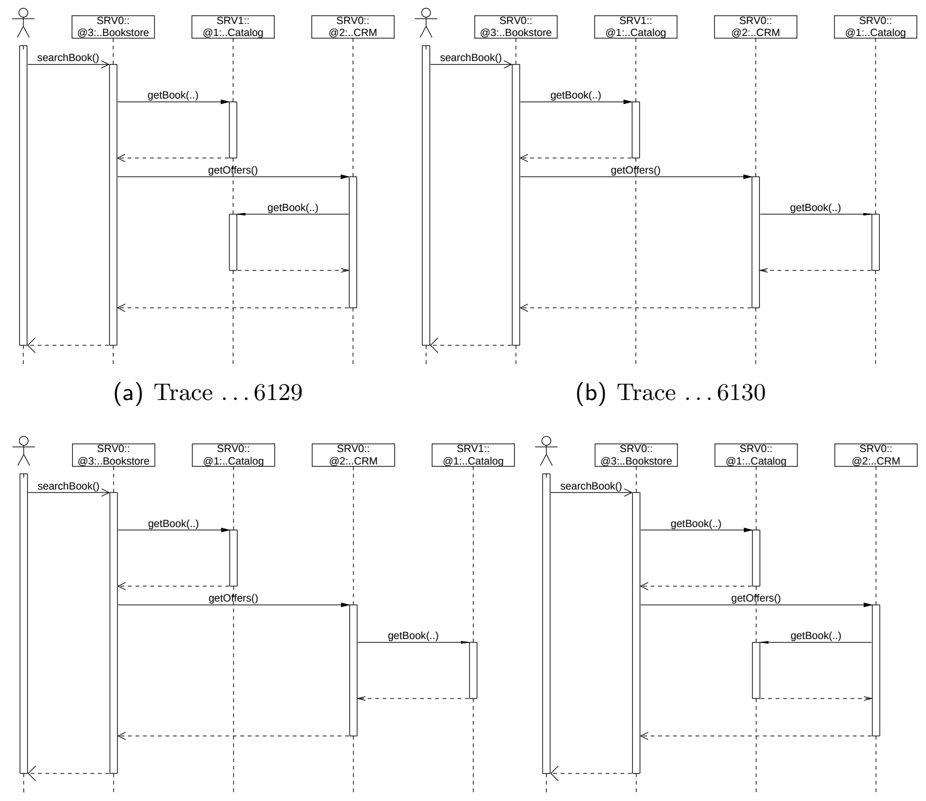
```
Class 0 ;  cardinality : 1635; # executions: 4;  representative : 6488138950668976129; max. stack depth: 2
```

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 shows the sequence diagram for the assembly-level trace equivalence representative (Section 5.3.1).

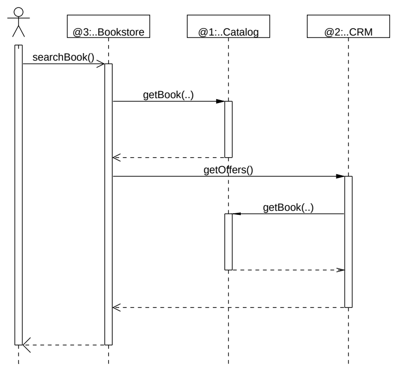
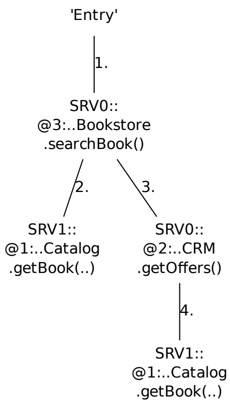


Figure 5.4: *Assembly-level* sequence diagram of trace . . . 6129

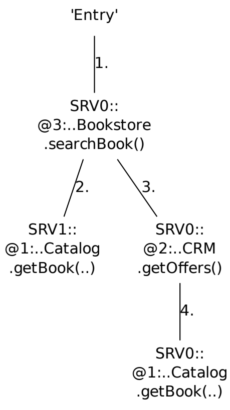
5.3.3 Call Trees

Trace Call Trees

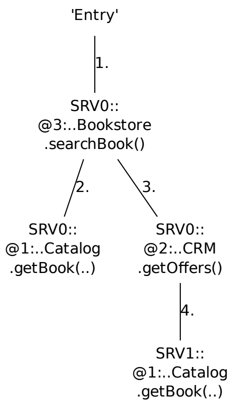
Trace call trees are generated using the command-line option `--plot-Call-Trees`. Figures 5.5(a)–5.5(d) show these call trees for each deployment-level trace equivalence representative (Section 5.3.1).



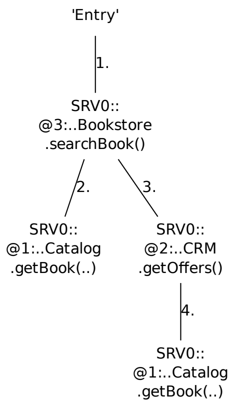
(a) Trace . . . 6129



(b) Trace . . . 6130



(c) Trace . . . 6131

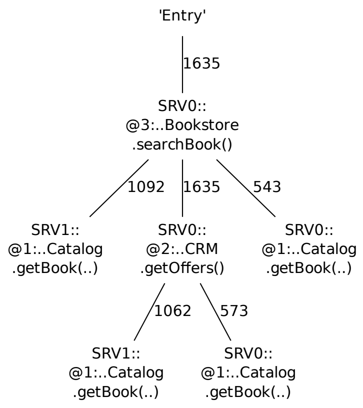


(d) Trace . . . 6141

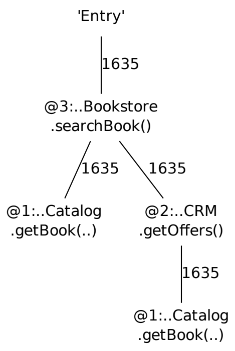
Figure 5.5: Calls trees of the trace equivalence class representatives (Listing 5.13)

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.

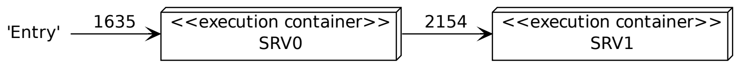
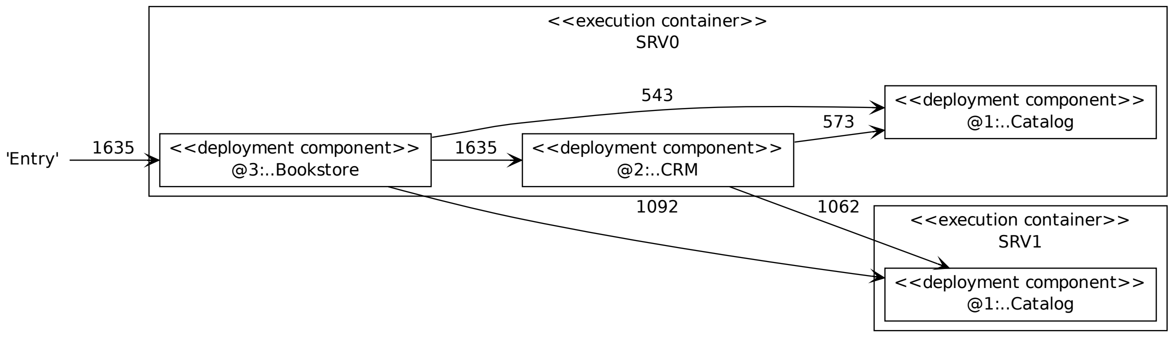


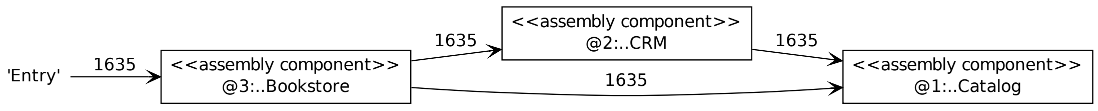
Figure 5.7: Container dependency graph

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

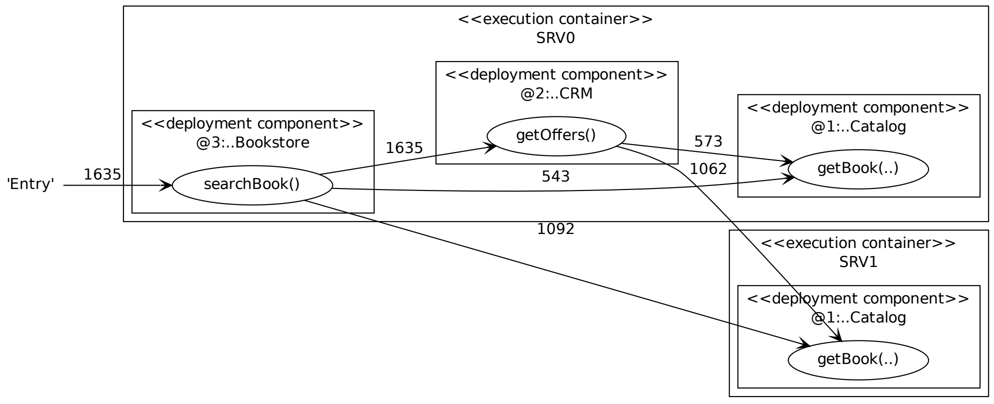


(b) assembly-level

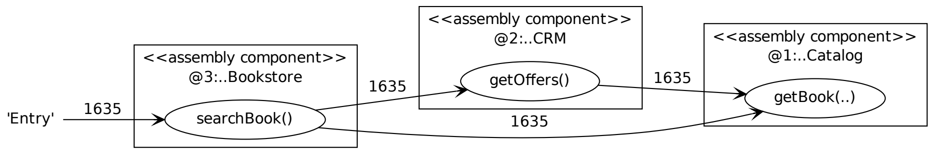
Figure 5.8: Component dependency graphs

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

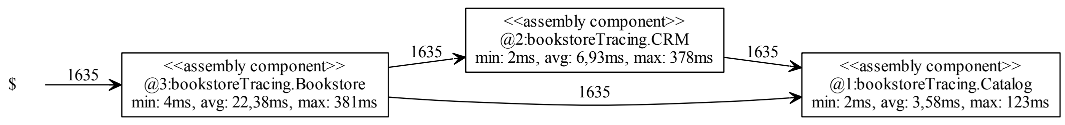


(b) assembly-level

Figure 5.9: Operation dependency graphs

5.3.5 Response Times in Dependency Graphs

The afore-mentioned dependency graphs can also be decorated by the response times, adding the minimum, the average, and the maximum response times of the components. The decoration will be generated with the additional command line parameter `responseTimes` behind the corresponding `plot-`command. An exemplaric graph with response times is shown in Figure 5.10.

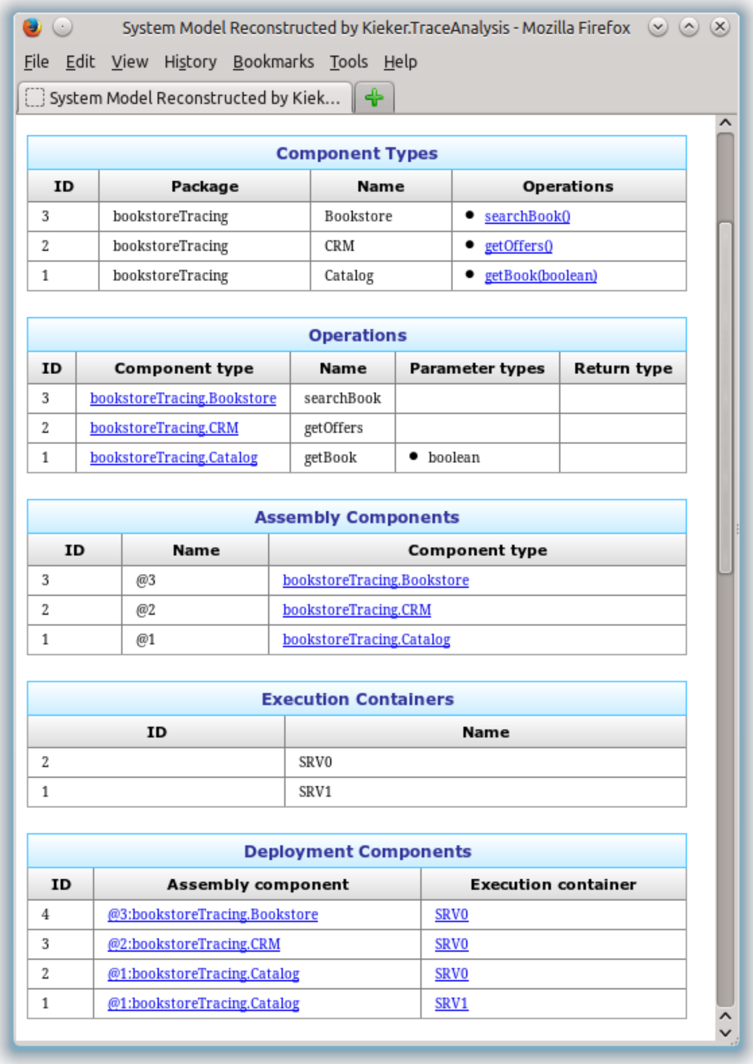


$
1635
<<assembly component>>
@3:bookstoreTracing.Bookstore
min: 4ms, avg: 22,38ms, max: 381ms
1635
<<assembly component>>
@2:bookstoreTracing.CRM
min: 2ms, avg: 6,93ms, max: 378ms
1635
1635
<<assembly component>>
@1:bookstoreTracing.Catalog
min: 2ms, avg: 3,58ms, max: 123ms

Figure 5.10: Assembly component dependency graph with response times

5.3.6 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 5.11 shows a screenshot of this file rendered by a web browser.


System Model Reconstructed by Kieker.TraceAnalysis - Mozilla Firefox
File Edit View History Bookmarks Tools Help
System Model Reconstructed by Kiek...

Component Types

ID	Package	Name	Operations
3	bookstoreTracing	Bookstore	● searchBook()
2	bookstoreTracing	CRM	● getOffers()
1	bookstoreTracing	Catalog	● getBook(boolean)

Operations

ID	Component type	Name	Parameter types	Return type
3	bookstoreTracing.Bookstore	searchBook		
2	bookstoreTracing.CRM	getOffers		
1	bookstoreTracing.Catalog	getBook	● boolean	

Assembly Components

ID	Name	Component type
3	@3	bookstoreTracing.Bookstore
2	@2	bookstoreTracing.CRM
1	@1	bookstoreTracing.Catalog

Execution Containers

ID	Name
2	SRV0
1	SRV1

Deployment Components

ID	Assembly component	Execution container
4	@3:bookstoreTracing.Bookstore	SRV0
3	@2:bookstoreTracing.CRM	SRV0
2	@1:bookstoreTracing.Catalog	SRV0
1	@1:bookstoreTracing.Catalog	SRV1

Figure 5.11: 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-1.9.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.

STOP The Windows `.bat` wrapper scripts must be executed from within the `bin/` directory.

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))
```

Listing A.1: Execution under UNIX-like systems

```
▷ convertLoggingTimestamp.bat --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))
```

Listing A.2: Execution under Windows

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 3.2).
- Replaying a recorded filesystem monitoring log in real-time (or faster/slower) in order to simulate incoming monitoring data from a running system, e.g., via JMS (see also Appendix C).

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

Usage

```
usage: kieker.tools.logReplayer.FilesystemLogReplayerStarter
 --a,--realtime-acceleration-factor <factor>
        Factor by which to accelerate (>1.0) or slow down (<1.0) the
        replay in realtime mode (defaults to 1.0, i.e., no
        acceleration/slow down).

 --c,--monitoring.configuration </path/to/monitoring.properties>
        Configuration to use for the Kieker monitoring instance

 --i,--inputdirs <dir1 ... dirN>
        Log directories to read data from

    --ignore-records-after-date <yyyyMMdd-HHmmss>
        Records logged after this date (UTC timezone) are ignored
        (disabled by default).

    --ignore-records-before-date <yyyyMMdd-HHmmss>
        Records logged before this date (UTC timezone) are ignored
        (disabled by default).

 --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/kieker-20100830-082225522-UTC
  --keep-logging-timestamps true
  --realtime false
```

Listing A.3: Execution under UNIX-like systems

```
▷ logReplay.bat
  --inputdirs ..\examples\userguide\ch5–trace-monitoring-aspectj\testdata\kieker-20100830-082225522-UTC
  --keep-logging-timestamps true
  --realtime false
```

Listing A.4: Execution under Windows

A.3 Script `kax-run.sh|bat`

Executes a Kieker.Analysis pipe-and-filter configuration file (`.kax` file), described in Section 4.1.
Main-class: `kieker.tools.KaxRun`

Usage

```
usage: kieker.tools.KaxRun -i <filename>
 --i,--input <filename>
        the analysis project file (.kax) loaded
```

A.4 Script `kax-viz.sh|bat`

Visualizes a Kieker.Analysis pipe-and-filter configuration file (`.kax` file), described in Section 4.1.
Main-class: `kieker.tools.KaxViz`

Usage

```
usage: kieker.tools.KaxViz -i <filename> [-svg <filename>]
 --i,--input <filename>
        the analysis project file (.kax) loaded

 --svg <filename>
        name of svg saved on close
```

A.5 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 <responseTimes>]
        [−−plot−Assembly−Component−Dependency−Graph <responseTimes>]
        [−−plot−Container−Dependency−Graph]
        [−−plot−Deployment−Operation−Dependency−Graph <responseTimes>]
        [−−plot−Assembly−Operation−Dependency−Graph <responseTimes>]
        [−−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]
        [−−include−self−loops] [−−traceColoring <color map file>]
        [−−addDescriptions <descriptions  file >]
 −−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)

    −−plot−Assembly−Sequence−Diagrams
         Generate and store  assembly−level  sequence diagrams (. pic)

    −−plot−Deployment−Component−Dependency−Graph <responseTimes>
         Generate and store  a deployment−level component dependency graph (.dot)

    −−plot−Assembly−Component−Dependency−Graph <responseTimes>
         Generate and store  an assembly−level  component dependency graph (.dot)

    −−plot−Container−Dependency−Graph
         Generate and store  a  container  dependency graph (.dot  file )

    −−plot−Deployment−Operation−Dependency−Graph <responseTimes>
         Generate and store  a deployment−level operation  dependency graph (.dot)

    −−plot−Assembly−Operation−Dependency−Graph <responseTimes>
         Generate and store  an assembly−level  operation  dependency graph (.dot)

    −−plot−Aggregated−Deployment−Call−Tree
         Generate and store  an aggregated deployment−level  call  tree  (. dot)

    −−plot−Aggregated−Assembly−Call−Tree
         Generate and store  an aggregated assembly−level  call  tree  (. dot)

    −−plot−Call−Trees
         Generate and store  call  trees  for  the  selected  traces  (. dot)
```

```
--print-Message-Traces
    Save message trace  representations  of  valid  traces  (.txt)

--print-Execution-Traces
    Save execution  trace  representations  of  valid  traces  (.txt)

--print-invalid-Execution-Traces
    Save a execution  trace  representations  of  invalid  trace  artifacts  (.txt)

--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  ms) after  which incomplete  traces  become invalid.  Defaults
    to  600,000 (i.e,  10 minutes).

--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  .

--include-self-loops
     If  selected,  self-loops are  included  in  the  visualizations  .

--traceColoring <color map file>
    Color  traces  according  to the  given  color  map given as a  properties  file
    (key:  trace  ID,  value:  color  in  hex format,  e.g.,  0xff0000  for  red;  use
    trace  ID  'default'  to  specify  the  default  color)

--addDescriptions <descriptions  file>
    Adds  descriptions  to  elements  according  to the  given  file  as  a
     properties  file  (key:  component ID, e.g.,  @1; value:  description)
```

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/kieker-20100830-082225522-UTC
  --outputdir .
  --plot-Deployment-Operation-Dependency-Graph
▷ dot -T pdf  deploymentOperationDependencyGraph.dot > deploymentOperationDependencyGraph.pdf
```

Listing A.5: Execution under UNIX-like systems

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

Listing A.6: Execution under Windows

Additional examples can be found in Chapter 5.

A.6 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 graphic output formats. This scripts simply calls the *Graphviz* and *PlotUtils* tools `dot` and `pic2plot`.

Usage

```
Example: dotPic-fileConverter.bat C:\Temp pdf png ps
```

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
```

Listing A.7: Execution under UNIX-like systems

```
▷ dotPic-fileConverter.bat out\ pdf png
```

Listing A.8: Execution under Windows

B Java EE Servlet Container Example

Using the sample Java web application MyBatis JPetStore,[1] this example demonstrates how to employ Kieker.Monitoring for monitoring a Java application running in a Java EE container—in this case Jetty.[2] Monitoring probes based on the Java EE Servlet API, Spring, and AspectJ are used to monitor execution, trace, and session data (see Section 5). The directory `examples/JavaEEServletContainerExample/` contains the prepared Jetty server with the MyBatis JPetStore application and the Kieker-based demo analysis application known from `http://demo.kieker-monitoring.net/`.

B.1 Setting

The subdirectory `jetty-hightide-jpetstore/` includes the Jetty server with the JPetStore application already deployed to the server's `webapps/` directory. The example is prepared to use two alternative types of Kieker probes: either the Kieker Spring interceptor (default) or the Kieker AspectJ aspects. Both alternatives additionally use Kieker's Servlet filter.

Required Libraries and Kieker.Monitoring Configuration Both settings require the files `aspectjweaver-1.7.4.jar` and `kieker-1.9.jar`, which are already included in the webapps's `WEB-INF/lib/` directory. Also, a Kieker configuration file is already included in the Jetty's root directory, where it is considered for configuration by Kieker.Monitoring in both modes.

Servlet Filter (Default) The file `web.xml` is located in the webapps's `WEB-INF/` directory. Kieker's Servlet filters are already enabled:

```
<filter>
  <filter-name>sessionAndTraceRegistrationFilter</filter-name>
  <filter-class>kieker.monitoring.probe.servlet.SessionAndTraceRegistrationFilter</filter-class
      >
  <init-param>
    <param-name>logFilterExecution</param-name>
    <param-value>true</param-value>
  </init-param>
```

[1] `http://www.mybatis.org/spring/sample.html`
[2] `http://www.eclipse.org/jetty/`

```
</ filter >
< filter -mapping>
  < filter -name>sessionAndTraceRegistrationFilter</filter-name>
  <url-pattern>/*</url-pattern>
</ filter -mapping>
```

Listing B.1: Enabling the Servlet filter in `web.xml`

This filter can be used with both the Spring-based and the AspectJ-based instrumentation mode.

Spring-based Instrumentation (Default) Kieker's Spring interceptor are already enabled in the file `applicationContext.xml`, located in the webapps's `WEB-INF/` directory:

```
  <bean id="opEMII"
        class ="kieker.monitoring.probe.spring.executions.
             OperationExecutionMethodInvocationInterceptor" />
  <aop:config>
      <aop:advisor advice-ref="opEMII"
             pointcut="execution(public * org.mybatis.jpetstore ..*.*(..) )"/>
  </aop:config>
```

Listing B.2: Enabling the Spring interceptor in `applicationContext.xml`

When using, for example, the `@Autowired` feature in your Spring beans, it can be necessary to force the usage of CGLIB proxy objects with `<aop:aspectj-autoproxy proxy-target-class="true"/>`.

AspectJ-based Instrumentation In order to use AspectJ-based instrumentation, the following changes need to be performed. The file `start.ini`, located in Jetty's root directory, allows to pass various JVM arguments, JVM system properties, and other options to the server on startup. When using AspectJ for instrumentation, the respective JVM argument needs to be activated in this file:

```
--exec
-javaagent:webapps/jpetstore/WEB-INF/lib/kieker-1.9_aspectj.jar
-Dkieker.monitoring.skipDefaultAOPConfiguration=true
-Daj.weaving.verbose=true
```

Listing B.3: Enabling the AspectJ weaver in Jetty's `start.ini`

The AspectJ configuration file `aop.xml` is already located in the webapps's `WEB-INF/classes/META-INF/` directory and configured to instrument the JPetStore classes with Kieker's `OperationExecutionAspectFull` aspect (Section 5).

When using the AspectJ-based instrumentation, make sure to disable the Spring interceptor in the file `applicationContext.xml`, mentioned above.

1. Start the Jetty server using the `start.jar` file. You should make sure that the server started properly by taking a look at the console output that appears during server startup.
2. Now, you can access the JPetStore application by opening the URL `http://localhost:8080/jpetstore/` (Figure B.1). Kieker initialization messages should appear in the console output.

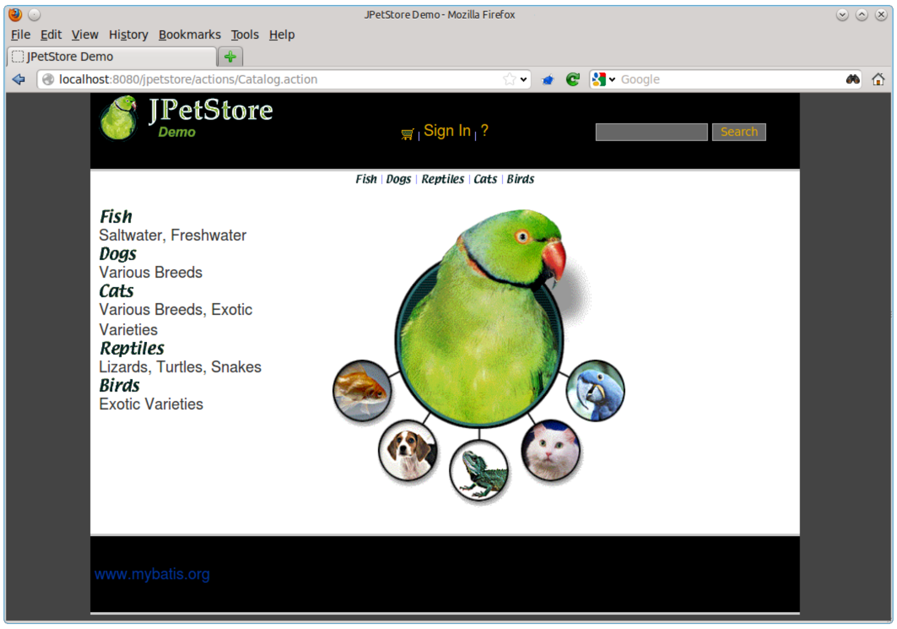

JPetStore Demo - Mozilla Firefox
File Edit View History Bookmarks Tools Help
JPetStore Demo
localhost:8080/jpetstore/actions/Catalog.action
Google
JPetStore
Demo
Sign In | ?
Search
Fish | Dogs | Reptiles | Cats | Birds
Fish
Saltwater, Freshwater
Dogs
Various Breeds
Cats
Various Breeds, Exotic Varieties
Reptiles
Lizards, Turtles, Snakes
Birds
Exotic Varieties
www.mybatis.org

Figure B.1: MyBatis JPetStore

3. Browse through the application to generate some monitoring data.
4. In this example, Kieker is configured to write the monitoring data to JMX in order to communicate with the Kieker-based demo analysis application, which is accessible via `localhost:8080/demo/`.
5. In order to write the monitoring data to the file system, the JMX writer needs to be disabled in the file `kieker.monitoring.properties`, which is located in the directory `webapps/jpetstore/WEB-INF/classes/META-INF/`. After a restart of the Jetty server, the Kieker startup output includes the information where the monitoring data is written to (should be a `kieker-<DATE-TIME>/` directory) located in the default temporary directory. This data can be analyzed and visualized using Kieker.TraceAnalysis, as described in Chapter 5.

C 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.

The following sections provide step-by-step instructions for the JMS server implementations ActiveMQ (Section C.1), HornetQ (C.2), and OpenJMS (C.3). The general procedure for each example is the following:

1. Download and prepare the respective JMS server implementation
2. Copy required libraries to the example directory
3. Start the JMS server
4. Start the analysis instance which receives records via JMS
5. Start the monitoring instance which sends records via JMS

Due to a bug in some JMS servers, avoid paths including white spaces.

C.1 ActiveMQ

C.1.1 Download and Prepare ActiveMQ

Download an ActiveMQ archive from `http://activemq.apache.org/download.html` and decompress it to the base directory of the example. Note, that there are two different distributions, one for Unix/Linux/Cygwin and another one for Windows.

Under UNIX-like systems, you'll need to set the executable-bit of the start script:

```
▷ chmod +x bin/activemq
```

Also under UNIX-like systems, make sure that the file `bin/activemq` includes UNIX line endings (e.g., using your favorite editor or the *dos2unix* tool).

C.1.2 Copy Kieker and ActiveMQ Libraries

Kieker Libraries Copy the following file from Kieker's binary distribution to the example's `lib/` directory.

1. `kieker-1.9_emf.jar` (from `dist/`)

ActiveMQ Libraries Copy the following files from the ActiveMQ release to the `lib/` directory of this example:

1. `activemq-all-<version>.jar` (from ActiveMQ's base directory)
2. `slf4j-log4j<version>.jar` (from ActiveMQ's `lib/optional` directory)
3. `log4j-<version>.jar` (from ActiveMQ's `lib/optional` directory)

C.1.3 Kieker Monitoring Configuration for ActiveMQ

The file `examples/userguide/appendix-JMS/META-INF/kieker.monitoring.properties-activeMQ` is already configured to use the `AsyncJMSWriter` via ActiveMQ. The important properties are the definition of the provider URL and the context factory:

```
kieker.monitoring.writer.jms.AsyncJMSWriter.ProviderUrl=tcp://127.0.0.1:61616/
```

Listing C.1: Excerpt from `kieker.monitoring.properties-activemq` configuring the provider URL of the JMS writer via ActiveMQ

```
kieker.monitoring.writer.jms.AsyncJMSWriter.ContextFactoryType=org.apache.activemq.jndi.
    ActiveMQInitialContextFactory
```

Listing C.2: Excerpt from `kieker.monitoring.properties-activemq` configuring the context factory of the JMS writer via ActiveMQ

C.1.4 Running the Example

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):

   ```
   ▷ bin/activemq start
   ```

   Listing C.3: Start of the JMS server under UNIX-like systems

   ```
   ▷ bin\activemq
   ```

   Listing C.4: Start of the JMS server under Windows

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

   ```
   ▷ ant run-analysis-activemq
   ```

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

   ```
   ▷ ant run-monitoring-activemq
   ```

C.2 HornetQ

C.2.1 Download and Prepare HornetQ

Download a HornetQ archive from `http://www.jboss.org/hornetq/downloads.html` and decompress it to the root directory of the example.
You need to create a queue in the HornetQ configuration file `config/stand-alone/non-clustered/hornetq-jms.xml`, as shown in Listing C.5.

```
<queue name="queue1">
   <entry name="/queue/queue1"/>
</queue>
```

Listing C.5: Queue definition to be added to the HornetQ configuration file

C.2.2 Copy Kieker and HornetQ Libraries

Kieker Libraries Copy the following file from Kieker's binary distribution to the example's `lib/` directory.

1. `kieker-1.9_emf.jar` (from `dist/`)

HornetQ Libraries Copy the following files from the HornetQ `lib/` folder to the `lib/` directory of this example:

1. `hornetq-jms-client.jar`
2. `hornetq-commons.jar` (if available)
3. `hornetq-core-client.jar`
4. `jboss-jms-api.jar`
5. `jnp-client.jar`
6. `netty.jar`

C.2.3 Kieker Monitoring Configuration for HornetQ

The file `examples/userguide/appendix-JMS/META-INF/kieker.monitoring.properties-hornetq` is already configured to use the `AsyncJMSWriter` via HornetQ. The important properties are the definition of the provider URL, the context factory, and the queue:

```
kieker.monitoring.writer.jms.AsyncJMSWriter.ProviderUrl=jnp://localhost:1099/
```

Listing C.6: Excerpt from `kieker.monitoring.properties-hornetq` configuring the provider URL of the JMS writer via HornetQ

```
kieker.monitoring.writer.jms.AsyncJMSWriter.ContextFactoryType=org.jnp.interfaces.
    NamingContextFactory
```

Listing C.7: Excerpt from `kieker.monitoring.properties-hornetq` configuring the context factory of the JMS writer via HornetQ

C.2.4 Running the Example

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

1. Start the JMS server:

   ```
   ▷ ./run.sh
   ```

   Listing C.8: Start of the JMS server under UNIX-like systems

   ```
   ▷ run.bat
   ```

   Listing C.9: Start of the JMS server under Windows

   Note that the script must be called from within HornetQ's `bin/` directory.

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

   ```
   ▷ ant run−analysis−hornetq
   ```

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

   ```
   ▷ ant run−monitoring−hornetq
   ```

C.3 OpenJMS

C.3.1 Download and Prepare OpenJMS

Download an OpenJMS install archive from `http://openjms.sourceforge.net` and decompress it to the root directory of the example. Under UNIX-like systems, make sure that the executable-bit of all scripts within the `bin/` directory are set.

C.3.2 Copy Kieker and OpenJMS Libraries

Kieker Libraries Copy the following files from Kieker's binary distribution to the example's `lib/` directory.

1. `kieker-1.9_emf.jar` (from `dist/`)
2. `commons-logging-1.1.2.jar` (from `lib/`)

OpenJMS Libraries Copy the following files from the OpenJMS `lib/` folder to the `lib/` directory of this example:

1. `openjms-<version>.jar`
2. `openjms-common-<version>.jar`
3. `openjms-net-<version>.jar`
4. `jms-<version>.jar`
5. `concurrent-<version>.jar`
6. `spice-jndikit-<version>.jar`

C.3.3 Kieker Monitoring Configuration for OpenJMS

The file `examples/userguide/appendix-JMS/META-INF/kieker.monitoring.properties-openjms` is already configured to use the `AsyncJMSWriter` via OpenJMS. The important properties are the definition of the provider URL and the context factory:

```
kieker.monitoring.writer.jms.AsyncJMSWriter.ProviderUrl=tcp://127.0.0.1:3035
```

Listing C.10: Excerpt from `kieker.monitoring.properties-openjms` configuring the provider URL of the JMS writer via OpenJMS

```
kieker.monitoring.writer.jms.AsyncJMSWriter.ContextFactoryType=org.exolab.jms.jndi.
    InitialContextFactory
```

Listing C.11: Excerpt from `kieker.monitoring.properties-openjms` configuring the context factory of the JMS writer via OpenJMS

C.3.4 Running the Example

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` and `OPENJMS_HOME` variables first):

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

   Listing C.12: Start of the JMS server under UNIX-like systems

   ```
   ▷ openjms-<version>\bin\startup.bat
   ```

   Listing C.13: Start of the JMS server under Windows

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

   ```
   ▷ ant run-analysis-openjms
   ```

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

   ```
   ▷ ant run-monitoring-openjms
   ```

D Sigar-Based Samplers for System-Level Monitoring

This chapter gives a brief description on how to use the included periodic samplers (Section 3.1.4) for monitoring CPU utilization and memory/swap usage. The directory `examples/userguide/appendix-Sigar/` contains the sources, ant scripts etc. used in this example. These samplers employ the Sigar API [1].

D.1 Preparation

1. Copy the files `kieker-1.9_emf.jar` and `sigar-1.6.4.jar` from the binary distribution to the example's `lib/` directory.
2. Additionally, depending on the underlying system platform, corresponding Sigar native libraries need to be placed in the example's `lib/` directory. Kieker's `lib/sigar-native-libs/` folder already includes the right libraries for 32 and 64 bit Linux/Windows platforms. Native libraries for other platforms can be downloaded from [1].

D.2 Using the Sigar-Based Samplers

Using a very short sampling period with Sigar ($<$ 500 ms) can result in monitoring log entries with NaN values.

The Sigar API [1] provides access to a number of system-level inventory and monitoring data, e.g., regarding memory, swap, cpu, file system, and network devices. Kieker includes Sigar-based samplers for monitoring CPU utilization (`CPUsDetailedPercSampler`, `CPUsCombinedPercSampler`) and memory/swap usage (`MemSwapUsageSampler`). When registered as a periodic sampler (Section 3.1.4), these samplers collect the data of interest employing the Sigar API, and write monitoring records of types `CPUUtilizationRecord`, `ResourceUtilizationRecord`, and `MemSwapUsageRecord` respectively to the configured monitoring log/stream.

Listing D.1 shows an excerpt from this example's `MonitoringStarter` which creates and registers two Sigar-based peridioc samplers. For reasons of performance and threadsafety, the `SigarSamplerFactory` should be used to create instances of the Sigar-based Samplers.

```java
   final ISigarSamplerFactory sigarFactory = SigarSamplerFactory.INSTANCE;

   final CPUsDetailedPercSampler cpuSampler =
      sigarFactory.createSensorCPUsDetailedPerc();
   final MemSwapUsageSampler memSwapSampler =
      sigarFactory.createSensorMemSwapUsage();

   final long offset = 2; // start after 2 seconds
   final long period = 5; // monitor every 5 seconds

   monitoringController.schedulePeriodicSampler(
      cpuSampler, offset, period, TimeUnit.SECONDS);
   monitoringController.schedulePeriodicSampler(
      memSwapSampler, offset, period, TimeUnit.SECONDS);
```

Listing D.1: Excerpt from MonitoringStarter.java

Based on the existing samplers, users can easily create custom Sigar-based samplers by extending the class `AbstractSigarSampler`. For example, Listing 3.1 in Section 3.1.4 shows the `MemSwapUsageSampler`'s *sample* method. Typically, it is also required to define a corresponding monitoring record type, as explained in Section 3.3. When implementing custom Sigar-based samplers, the `SigarSamplerFactory`'s *getSigar* method should be used to retrieve a `Sigar` instance.

This example uses a stand-alone Java application to set up a Sigar-based monitoring process. When using servlet containers, users may consider implementing this routine as a `ServletContextListener`, which are executed when the container is started and shutdown. As an example, Kieker includes a `CPUMemUsageServletContextListener`.

D.3 Executing the Example

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

1. Monitoring CPU utilization and memory usage for 30 seconds (class `MonitoringStarter`):

   ```
   ▷ ant run−monitoring
   ```

   Kieker's console output lists the location of the directory containing the file system monitoring log. The following listing shows an excerpt:

```
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-20110511-10095928-UTC-avanhoorn-thinkpad-KIEKER-SINGLETON'
```

A sample monitoring log can be found in the directory `examples/userguide/appendix-Sigar/testdata/kieker-20110511-10095928-UTC-avanhoorn-thinkpad-KIEKER-SINGLETON/`.

2. Analyzing the monitoring data (class `AnalysisStarter`):

```
▷ ant run-analysis -Danalysis.directory=</path/to/monitoring/log/>
```

You need to replace `</path/to/monitoring/log/>` by the location of the file system monitoring log. You can also use the above-mentioned monitoring log included in the example.

The `AnalysisStarter` produces a simple console output for each monitoring record, as shown in the following excerpt:

```
Wed, 11 May 2011 10:10:01 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 0 ;  utilization:  0.00 %
Wed, 11 May 2011 10:10:01 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 1 ;  utilization:  0.00 %
Wed, 11 May 2011 10:10:01 +0000 (UTC): [Mem/Swap] host: thinkpad ; mem usage: 722.0 MB ; swap
    usage: 0.0 MB
Wed, 11 May 2011 10:10:06 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 0 ;  utilization:  5.35 %
Wed, 11 May 2011 10:10:06 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 1 ;  utilization:  1.31 %
Wed, 11 May 2011 10:10:06 +0000 (UTC): [Mem/Swap] host: thinkpad ; mem usage: 721.0 MB ; swap
    usage: 0.0 MB
Wed, 11 May 2011 10:10:11 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 0 ;  utilization:  1.80 %
Wed, 11 May 2011 10:10:11 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 1 ;  utilization:  0.20 %
Wed, 11 May 2011 10:10:11 +0000 (UTC): [Mem/Swap] host: thinkpad ; mem usage: 721.0 MB ; swap
    usage: 0.0 MB
Wed, 11 May 2011 10:10:16 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 0 ;  utilization:  1.40 %
Wed, 11 May 2011 10:10:16 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 1 ;  utilization:  0.79 %
Wed, 11 May 2011 10:10:16 +0000 (UTC): [Mem/Swap] host: thinkpad ; mem usage: 721.0 MB ; swap
    usage: 0.0 MB
Wed, 11 May 2011 10:10:21 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 0 ;  utilization:  1.80 %
Wed, 11 May 2011 10:10:21 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 1 ;  utilization:  0.79 %
Wed, 11 May 2011 10:10:21 +0000 (UTC): [Mem/Swap] host: thinkpad ; mem usage: 721.0 MB ; swap
    usage: 0.0 MB
Wed, 11 May 2011 10:10:26 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 0 ;  utilization:  0.40 %
Wed, 11 May 2011 10:10:26 +0000 (UTC): [CPU] host: thinkpad ; cpu-id: 1 ;  utilization:  0.59 %
Wed, 11 May 2011 10:10:26 +0000 (UTC): [Mem/Swap] host: thinkpad ; mem usage: 721.0 MB ; swap
    usage: 0.0 MB
```

E Kieker.Monitoring Default Configuration

This is the file `kieker.monitoring.properties` from the binary release and constitutes Kieker.Monitoring's default configuration. Section 3.2 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

## Whether a debug mode is activated.
## This changes a few internal id generation mechanisms to enable
## easier debugging. Additionally, it is possible to enable debug
## logging in the settings of the used logger.
## This setting should usually not be set to true.
kieker.monitoring.debug=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 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. (currently not used)
kieker.monitoring.initialExperimentId=1
```

```
## Automatically add a metadata record to the monitoring log when writing
## the first monitoring record. The metadata record contains infromation
## on the configuration of the monitoring controller.
kieker.monitoring.metadata=true

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

## Whether a shutdown hook should be registered.
## This ensures that necessary cleanup steps are finished and no
## information is lost due to asynchronous writers.
## This should usually not be set to false.
kieker.monitoring.useShutdownHook=true

## Whether any JMX functionality is available
kieker.monitoring.jmx=false
kieker.monitoring.jmx.domain=kieker.monitoring

## Enable/Disable the MonitoringController MBean
kieker.monitoring.jmx.MonitoringController=true
kieker.monitoring.jmx.MonitoringController.name=MonitoringController

## Controls JMX remote access
kieker.monitoring.jmx.remote=false
kieker.monitoring.jmx.remote.port=59999
kieker.monitoring.jmx.remote.name=JMXServer
## If the SUN-JMX Bootstrap class is not available, a fallback to the
## default implementation can be used. The fallback solution prevents
## the VM from terminating.
## A graceful shutdown is done by connecting to the JMXServer, there to
## kieker.monitoring.JMXServer and using the operation stop()
kieker.monitoring.jmx.remote.fallback=true
## These properties will be forwarded to configure the JMX server
com.sun.management.jmxremote.local.only=false
com.sun.management.jmxremote.authenticate=false
com.sun.management.jmxremote.ssl=false

## The size of the thread pool used to execute registered periodic sensor jobs.
## The thread pool is also used to periodically read the config file for adaptive
## monitoring.
## Set to 0 to deactivate scheduling.
kieker.monitoring.periodicSensorsExecutorPoolSize=1

## Enable or disable adaptive monitoring.
kieker.monitoring.adaptiveMonitoring.enabled=false
#
## Default location of the adaptive monitoring configuration File
kieker.monitoring.adaptiveMonitoring.configFile=META-INF/kieker.monitoring.adaptiveMonitoring.conf
```

```
#
## Enable/disable the updating of the pattern file by activating or deactivating
## probes through the api.
kieker.monitoring.adaptiveMonitoring.updateConfigFile=false
#
## The delay in seconds in which the pattern file is checked for changes.
## Requires kieker.monitoring.periodicSensorsExecutorPoolSize > 0.
## Set to 0 to disable the observation.
kieker.monitoring.adaptiveMonitoring.readInterval=30


###########################
####### TIMER #######
###########################
## Selection of the timer used by Kieker (classname)
## The value must be a fully-qualified classname of a class implementing
## kieker.monitoring.timer.ITimeSource and providing a constructor that
## accepts a single Configuration.
kieker.monitoring.timer=kieker.monitoring.timer.SystemNanoTimer

####
#kieker.monitoring.timer=kieker.monitoring.timer.SystemMilliTimer
#
## A timer with millisecond precision.
#
## The offset of the timer. The time returned is since 1970-1-1
## minus this offset. If the offset is empty it is set to the current
## time.
## The offset must be specified in milliseconds.
kieker.monitoring.timer.SystemMilliTimer.offset=0
## The timeunit used to report the timestamp.
## Accepted values:
## 0 - nanoseconds
## 1 - microseconds
## 2 - milliseconds
## 3 - seconds
kieker.monitoring.timer.SystemMilliTimer.unit=0

####
#kieker.monitoring.timer=kieker.monitoring.timer.SystemNanoTimer
#
## A timer with nanosecond precision.
#
## The offset of the timer. The time returned is since 1970-1-1
## minus this offset. If the offset is empty it is set to the current
## time.
## The offset must be specified in milliseconds.
kieker.monitoring.timer.SystemNanoTimer.offset=0
## The timeunit used to report the timestamp.
## Accepted values:
```

```
## 0 - nanoseconds
## 1 - microseconds
## 2 - milliseconds
## 3 - seconds
kieker.monitoring.timer.SystemNanoTimer.unit=0


###########################
####### WRITER #######
###########################
## Selection of monitoring data writer (classname)
## The value must be a fully-qualified classname of a class implementing
## kieker.monitoring.writer.IMonitoringWriter and providing a constructor that
## accepts 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.AsyncDummyWriter
#
## Configuration Properties of the AsyncDummyWriter
kieker.monitoring.writer.AsyncDummyWriter.key=value
#
## 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.AsyncDummyWriter.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.AsyncDummyWriter.QueueFullBehavior=0
#
## Maximum time to wait for the writer threads to finish (in milliseconds).
## A MaxShutdownDelay of -1 means infinite waiting.
kieker.monitoring.writer.AsyncDummyWriter.MaxShutdownDelay=-1


#####
#kieker.monitoring.writer=kieker.monitoring.writer.PrintStreamWriter
#
```

```
## The PrintStream used to print the monitoring records.
## Either STDOUT or STDERR.
## Other values are used as a filenames for a target log file .
## You should use another writer instead of this writer for logging to files !
kieker.monitoring.writer.PrintStreamWriter.Stream=STDOUT


#####
#kieker.monitoring.writer=kieker.monitoring.writer.filesystem.SyncFsWriter
#
## In order to use a custom directory, set customStoragePath as desired. Examples:
## /var/kieker or C:\\KiekerData (ensure the folder exists).
## Otherwise the default temporary directory will be used
kieker.monitoring.writer.filesystem.SyncFsWriter.customStoragePath=
#
## The maximal number of entries (records) per created file.
## Must be greater than zero.
kieker.monitoring.writer.filesystem.SyncFsWriter.maxEntriesInFile=25000
#
## The maximal file size of the generated monitoring log. Older files will be
## deleted if this file size is exceeded. Given in MiB.
## At least one file will always remain, regardless of size!
## Use -1 to ignore this functionality.
kieker.monitoring.writer.filesystem.SyncFsWriter.maxLogSize=-1
#
## The maximal number of log files generated. Older files will be
## deleted if this number is exceeded.
## At least one file will always remain, regardless of size!
## Use -1 to ignore this functionality.
kieker.monitoring.writer.filesystem.SyncFsWriter.maxLogFiles=-1
#
## When flushing is disabled, it could require a lot of records before
## finally any writing is done.
kieker.monitoring.writer.filesystem.SyncFsWriter.flush=true
#
## When flushing is disabled, records are buffered in memory before written.
## This setting configures the size of the used buffer in bytes.
kieker.monitoring.writer.filesystem.SyncFsWriter.bufferSize=8192


#####
#kieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncFsWriter
#
## In order to use a custom directory, set customStoragePath as desired. Examples:
## /var/kieker or C:\\KiekerData (ensure the folder exists).
## Otherwise the default temporary directory will be used
kieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath=
#
## The maximal number of entries (records) per created file.
## Must be greater than zero.
```

```
kieker.monitoring.writer.filesystem.AsyncFsWriter.maxEntriesInFile=25000
#
## The maximal file size of the generated monitoring log. Older files will be
## deleted if this file size is exceeded. Given in MiB.
## At least one file will always remain, regardless of size!
## Use -1 to ignore this functionality.
kieker.monitoring.writer.filesystem.AsyncFsWriter.maxLogSize=-1
#
## The maximal number of log files generated. Older files will be
## deleted if this number is exceeded.
## At least one file will always remain, regardless of size!
## Use -1 to ignore this functionality.
kieker.monitoring.writer.filesystem.AsyncFsWriter.maxLogFiles=-1
#
## When flushing is disabled, it could require a lot of records before
## finally any writing is done.
kieker.monitoring.writer.filesystem.AsyncFsWriter.flush=true
#
## When flushing is disabled, records are buffered in memory before written.
## This setting configures the size of the used buffer in bytes.
kieker.monitoring.writer.filesystem.AsyncFsWriter.bufferSize=8192
#
## 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
#
## Maximum time to wait for the writer threads to finish (in milliseconds).
## A MaxShutdownDelay of -1 means infinite waiting.
kieker.monitoring.writer.filesystem.AsyncFsWriter.MaxShutdownDelay=-1


#####
#kieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncBinaryFsWriter
#
## In order to use a custom directory, set customStoragePath as desired. Examples:
## /var/kieker or C:\\KiekerData (ensure the folder exists).
## Otherwise the default temporary directory will be used
kieker.monitoring.writer.filesystem.AsyncBinaryFsWriter.customStoragePath=
#
## The maximal number of entries (records) per created file.
## Must be greater than zero.
kieker.monitoring.writer.filesystem.AsyncBinaryFsWriter.maxEntriesInFile=25000
```

```
#
## The maximal file size of the generated monitoring log. Older files will be
## deleted if this file size is exceeded. Given in MiB.
## At least one file will always remain, regardless of size!
## Use -1 to ignore this functionality.
kieker.monitoring.writer.filesystem.AsyncBinaryFsWriter.maxLogSize=-1
#
## The maximal number of log files generated. Older files will be
## deleted if this number is exceeded.
## At least one file will always remain, regardless of size!
## Use -1 to ignore this functionality.
kieker.monitoring.writer.filesystem.AsyncBinaryFsWriter.maxLogFiles=-1
#
## Whether the generated log files are compressed before writing to disk.
## Supported values are: NONE, DEFLATE, GZIP, ZIP
kieker.monitoring.writer.filesystem.AsyncBinaryFsWriter.compress=NONE
#
## Records are buffered in memory before written to disk.
## This setting configures the size of the used buffer in bytes.
kieker.monitoring.writer.filesystem.AsyncBinaryFsWriter.bufferSize=8192
#
## 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.AsyncBinaryFsWriter.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.AsyncBinaryFsWriter.QueueFullBehavior=0
#
## Maximum time to wait for the writer threads to finish (in milliseconds).
## A MaxShutdownDelay of -1 means infinite waiting.
kieker.monitoring.writer.filesystem.AsyncBinaryFsWriter.MaxShutdownDelay=-1


#####
#kieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncBinaryNFsWriter
#
## In order to use a custom directory, set customStoragePath as desired. Examples:
## /var/kieker or C:\\KiekerData (ensure the folder exists).
## Otherwise the default temporary directory will be used
kieker.monitoring.writer.filesystem.AsyncBinaryNFsWriter.customStoragePath=
#
## The maximal number of entries (records) per created file.
## Must be greater than zero.
kieker.monitoring.writer.filesystem.AsyncBinaryNFsWriter.maxEntriesInFile=25000
#
```

```
## The maximal file size of the generated monitoring log. Older files will be
## deleted if this file size is exceeded. Given in MiB.
## At least one file will always remain, regardless of size!
## Use -1 to ignore this functionality.
kieker.monitoring.writer.filesystem.AsyncBinaryNFsWriter.maxLogSize=-1
#
## The maximal number of log files generated. Older files will be
## deleted if this number is exceeded.
## At least one file will always remain, regardless of size!
## Use -1 to ignore this functionality.
kieker.monitoring.writer.filesystem.AsyncBinaryNFsWriter.maxLogFiles=-1
#
## Records are buffered in memory before written to disk.
## This setting configures the size of the used buffer in bytes.
kieker.monitoring.writer.filesystem.AsyncBinaryNFsWriter.bufferSize=65535
#
## 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.AsyncBinaryNFsWriter.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.AsyncBinaryNFsWriter.QueueFullBehavior=0
#
## Maximum time to wait for the writer threads to finish (in milliseconds).
## A MaxShutdownDelay of -1 means infinite waiting.
kieker.monitoring.writer.filesystem.AsyncBinaryNFsWriter.MaxShutdownDelay=-1


#####
#kieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncAsciiZipWriter
#
## In order to use a custom directory, set customStoragePath as desired. Examples:
## /var/kieker or C:\\KiekerData (ensure the folder exists).
## Otherwise the default temporary directory will be used
kieker.monitoring.writer.filesystem.AsyncAsciiZipWriter.customStoragePath=
#
## The maximal number of entries (records) per created file.
## Must be greater than zero.
kieker.monitoring.writer.filesystem.AsyncAsciiZipWriter.maxEntriesInFile=25000
#
## Records are buffered in memory before written to disk.
## This setting configures the size of the used buffer in bytes.
kieker.monitoring.writer.filesystem.AsyncAsciiZipWriter.bufferSize=8192
#
## 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.AsyncAsciiZipWriter.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.AsyncAsciiZipWriter.QueueFullBehavior=0
#
## Maximum time to wait for the writer threads to finish (in milliseconds).
## A MaxShutdownDelay of -1 means infinite waiting.
kieker.monitoring.writer.filesystem.AsyncAsciiZipWriter.MaxShutdownDelay=-1
#
## Sets the compression level. The only valid values are:
## -1: default compression
## 0: no compression
## 1-9: from best speed to best compression
kieker.monitoring.writer.filesystem.AsyncAsciiZipWriter.compressionLevel=-1


#####
#kieker.monitoring.writer=kieker.monitoring.writer.filesystem.AsyncBinaryZipWriter
#
## In order to use a custom directory, set customStoragePath as desired. Examples:
## /var/kieker or C:\\KiekerData (ensure the folder exists).
## Otherwise the default temporary directory will be used
kieker.monitoring.writer.filesystem.AsyncBinaryZipWriter.customStoragePath=
#
## The maximal number of entries (records) per created file.
## Must be greater than zero.
kieker.monitoring.writer.filesystem.AsyncBinaryZipWriter.maxEntriesInFile=25000
#
## Records are buffered in memory before written to disk.
## This setting configures the size of the used buffer in bytes.
kieker.monitoring.writer.filesystem.AsyncBinaryZipWriter.bufferSize=8192
#
## 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.AsyncBinaryZipWriter.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.AsyncBinaryZipWriter.QueueFullBehavior=0
#
```

```
## Maximum time to wait for the writer threads to finish (in milliseconds).
## A MaxShutdownDelay of -1 means infinite waiting.
kieker.monitoring.writer.filesystem.AsyncBinaryZipWriter.MaxShutdownDelay=-1
#
## Sets the compression level. The only valid values are:
## -1: default compression
## 0: no compression
## 1-9: from best speed to best compression
kieker.monitoring.writer.filesystem.AsyncBinaryZipWriter.compressionLevel=-1


#####
#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
## - ActiveMQ: tcp://127.0.0.1:61616/
## - HornetQ: jnp://localhost:1099/
## - OpenJMS: tcp://127.0.0.1:3035/
kieker.monitoring.writer.jms.AsyncJMSWriter.ProviderUrl=tcp://127.0.0.1:61616/
#
## 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.
## - ActiveMQ: org.apache.activemq.jndi.ActiveMQInitialContextFactory
## - HornetQ: org.jnp.interfaces.NamingContextFactory
## - OpenJMS: org.exolab.jms.jndi.InitialContextFactory
kieker.monitoring.writer.jms.AsyncJMSWriter.ContextFactoryType=org.apache.activemq.jndi.
    ActiveMQInitialContextFactory
#
## 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
#
## Maximum time to wait for the writer threads to finish (in milliseconds).
## A MaxShutdownDelay of -1 means infinite waiting.
kieker.monitoring.writer.jms.AsyncJMSWriter.MaxShutdownDelay=-1


#####
#kieker.monitoring.writer=kieker.monitoring.writer.database.SyncDbWriter
#
## Database driver classname
## Examples:
##  MySQL: com.mysql.jdbc.Driver
##  DerbyDB: org.apache.derby.jdbc.EmbeddedDriver
kieker.monitoring.writer.database.SyncDbWriter.DriverClassname=org.apache.derby.jdbc.
    EmbeddedDriver
#
## 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:derby:tmp/KIEKER;user=
    DBUSER;password=DBPASS;create=true
#
## Prefix for the names of the database tables
kieker.monitoring.writer.database.SyncDbWriter.TablePrefix=kieker
#
## Drop already existing tables or terminate monitoring with an error.
kieker.monitoring.writer.database.SyncDbWriter.DropTables=false


#####
#kieker.monitoring.writer=kieker.monitoring.writer.database.AsyncDbWriter
#
## Database driver classname
##  MySQL: com.mysql.jdbc.Driver
##  DerbyDB: org.apache.derby.jdbc.EmbeddedDriver
kieker.monitoring.writer.database.AsyncDbWriter.DriverClassname=org.apache.derby.jdbc.
    EmbeddedDriver
#
## 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:derby:tmp/KIEKER;user=
    DBUSER;password=DBPASS;create=true
```

```
#
## Prefix for the names of the database tables
kieker.monitoring.writer.database.AsyncDbWriter.TablePrefix=kieker
#
## Drop already existing tables or terminate monitoring with an error.
kieker.monitoring.writer.database.AsyncDbWriter.DropTables=false
#
## The number of concurrent Database connections.
kieker.monitoring.writer.database.AsyncDbWriter.numberOfConnections=4
#
## 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.database.AsyncDbWriter.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.database.AsyncDbWriter.QueueFullBehavior=0
#
## Maximum time to wait for the writer threads to finish (in milliseconds).
## A MaxShutdownDelay of -1 means infinite waiting.
kieker.monitoring.writer.database.AsyncDbWriter.MaxShutdownDelay=-1


#####
#kieker.monitoring.writer=kieker.monitoring.writer.jmx.JMXWriter
#
## The domain used to register the MonitoringLog. If empty, the value
## of "kieker.monitoring.jmx.domain" will be used.
kieker.monitoring.writer.jmx.JMXWriter.domain=
#
## The name of the MonitoringLog in the domain.
kieker.monitoring.writer.jmx.JMXWriter.logname=MonitoringLog


#####
#kieker.monitoring.writer=kieker.monitoring.writer.tcp.TCPWriter
#
## The hostname the TCPWriter connects to.
kieker.monitoring.writer.tcp.TCPWriter.hostname=localhost
#
## The ports the TCPWriter connects to.
kieker.monitoring.writer.tcp.TCPWriter.port1=10133
kieker.monitoring.writer.tcp.TCPWriter.port2=10134
#
## The size of the buffer used by the TCPWriter in bytes.
## Should be large enough to fit at least single string records (> 1KiB).
```

```
kieker.monitoring.writer.tcp.TCPWriter.bufferSize=65535
#
## Should each record be immediately sent?
kieker.monitoring.writer.tcp.TCPWriter.flush=false
#
## 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.tcp.TCPWriter.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.tcp.TCPWriter.QueueFullBehavior=0
#
## Maximum time to wait for the writer threads to finish (in milliseconds).
## A MaxShutdownDelay of -1 means infinite waiting.
kieker.monitoring.writer.tcp.TCPWriter.MaxShutdownDelay=-1


#####
#kieker.monitoring.writer=kieker.monitoring.writer.explorviz.ExplorVizExportWriter
#
## The hostname the ExplorVizExportWriter connects to.
kieker.monitoring.writer.explorviz.ExplorVizExportWriter.hostname=localhost
#
## The ports the TCPWriter connects to.
kieker.monitoring.writer.explorviz.ExplorVizExportWriter.port=10133
#
## The size of the buffer used by the TCPWriter in bytes.
## Should be large enough to fit at least single string records (> 1KiB).
kieker.monitoring.writer.explorviz.ExplorVizExportWriter.bufferSize=65535
#
## Should each record be immediately sent?
kieker.monitoring.writer.explorviz.ExplorVizExportWriter.flush=false
#
## 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.explorviz.ExplorVizExportWriter.QueueSize=1000000
#
## Behavior of the asynchronous writer when the internal queue is full:
## 0: terminate Monitoring with an error
## 1: writer blocks until queue capacity is available (default)
## 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.explorviz.ExplorVizExportWriter.QueueFullBehavior=1
#
```

```
## Maximum time to wait for the writer threads to finish (in milliseconds).
## A MaxShutdownDelay of -1 means infinite waiting.
kieker.monitoring.writer.explorviz.ExplorVizExportWriter.MaxShutdownDelay=-1
```

Listing E.1: kieker.monitoring.properties

F Additional Source Code Listings

F.1 MyNamedPipeManager and MyPipe

```java
public class MyNamedPipeManager {

  private static final MyNamedPipeManager PIPE_MGR_INSTANCE = new
      MyNamedPipeManager();

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

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

  public MyPipe acquirePipe(final String pipeName) throws
      IllegalArgumentException {
    if ((pipeName == null) || (pipeName.length() == 0)) {
      throw new IllegalArgumentException("Invalid connection name: '" + pipeName
           + "'");
    }
    MyPipe conn;
    synchronized (this) {
      conn = this.pipeMap.get(pipeName);
      if (conn == null) {
        conn = new MyPipe(pipeName);
        this.pipeMap.put(pipeName, conn);
      }
    }
    return conn;
  }
}
```

Listing F.1: MyNamedPipeManager.java

```java
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 F.2: MyPipe.java

```java
public class PipeData {

  private final long loggingTimestamp;
  private final Object[] recordData;
  private final Class<? extends IMonitoringRecord> recordType;

  public PipeData(final long loggingTimestamp, final Object[] recordData, final
      Class<? extends IMonitoringRecord> recordType) {
    this.loggingTimestamp = loggingTimestamp;
    this.recordData = recordData; // in real settings we would clone
    this.recordType = recordType;
  }

  public final long getLoggingTimestamp() {
    return this.loggingTimestamp;
  }

  public final Object[] getRecordData() {
    return this.recordData; // in real settings we would clone
  }

  public Class<? extends IMonitoringRecord> getRecordType() {
    return this.recordType;
  }
}
```

Listing F.3: PipeData.java

G Example Console Outputs

G.1 Quick Start Example (Chapter 2)

```
Apr 14, 2014 10:29:09 PM kieker.monitoring.core.configuration.ConfigurationFactory
        createSingletonConfiguration
INFO: Loading properties from properties file in classpath: 'META-INF/kieker.monitoring.properties'
Apr 14, 2014 10:29:09 PM kieker.monitoring.core.configuration.ConfigurationFactory
        loadConfigurationFromResource
WARNING: File 'META-INF/kieker.monitoring.properties' not found in classpath
Apr 14, 2014 10:29:09 PM kieker.monitoring.core.controller.MonitoringController createInstance
INFO: Current State of kieker.monitoring (1.9) Status: 'enabled'
        Name: 'KIEKER-SINGLETON'; Hostname: 'myHost'; experimentID: '1'
JMXController: JMX disabled
RegistryController: 0 strings registered.
TimeSource: 'kieker.monitoring.timer.SystemNanoTimer'
        Time in nanoseconds (with nanoseconds precision) since Thu Jan 01 01:00:00 CET 1970'
ProbeController: disabled
WriterController:
        Number of Inserts: '0'
        Automatic assignment of logging timestamps: 'true'
Writer: 'kieker.monitoring.writer.filesystem.AsyncFsWriter'
        Configuration:
                kieker.monitoring.writer.filesystem.AsyncFsWriter.flush='true'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.maxLogSize='-1'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueFullBehavior='0'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.MaxShutdownDelay='-1'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.maxEntriesInFile='25000'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.bufferSize='8192'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.maxLogFiles='-1'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath=''
                kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueSize='10000'
        Records lost: 0
        Writer Threads (1):
                Finished: 'false'; Writing to Directory: '/tmp/kieker-20140414-202909312-UTC-myHost
                        -KIEKER-SINGLETON'
Sampling Controller: Periodic Sensor available: Poolsize: '0'; Scheduled Tasks: '0'
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
Apr 14, 2014 10:29:09 PM kieker.monitoring.core.controller.MonitoringController run
INFO: ShutdownHook notifies controller to initiate shutdown
```

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

```
Apr 14, 2014 10:33:08 PM kieker. analysis . plugin . reader . filesystem .FSDirectoryReader run
INFO: < Loading /tmp/kieker−20140414−202909312−UTC−myHost−KIEKER−SINGLETON/kieker
        −20140414−202909334−UTC−000−Thread−1.dat
Apr 14, 2014 10:33:08 PM kieker. analysis . AnalysisController  handleKiekerMetadataRecord
INFO: Kieker metadata: version ='1.9', controllerName='KIEKER−SINGLETON', hostname='myHost',
        experimentId='1', debugMode='false', timeOffset='0', timeUnit='NANOSECONDS', numberOfRecords='1'
 TeeFilter −2(OperationExecutionRecord) 1397507349333612418;public void kieker.examples.userguide.
        ch2bookstore.manual.Catalog.getBook(boolean);<no−session−id
        >;−1;1397507349331135863;1397507349333237208;myHost;−1;−1
 TeeFilter −2(OperationExecutionRecord) 1397507349336643445;public void kieker.examples.userguide.
        ch2bookstore.manual.Catalog.getBook(boolean);<no−session−id
        >;−1;1397507349334501352;1397507349336575189;myHost;−1;−1
 TeeFilter −2(OperationExecutionRecord) 1397507349338830403;public void kieker.examples.userguide.
        ch2bookstore.manual.Catalog.getBook(boolean);<no−session−id
        >;−1;1397507349336724513;1397507349338814261;myHost;−1;−1
 TeeFilter −2(OperationExecutionRecord) 1397507349340925965;public void kieker.examples.userguide.
        ch2bookstore.manual.Catalog.getBook(boolean);<no−session−id
        >;−1;1397507349338839217;1397507349340906276;myHost;−1;−1
 TeeFilter −2(OperationExecutionRecord) 1397507349343092031;public void kieker.examples.userguide.
        ch2bookstore.manual.Catalog.getBook(boolean);<no−session−id
        >;−1;1397507349340999396;1397507349343071939;myHost;−1;−1
 TeeFilter −2(OperationExecutionRecord) 1397507349345216288;public void kieker.examples.userguide.
        ch2bookstore.manual.Catalog.getBook(boolean);<no−session−id
        >;−1;1397507349343104120;1397507349345187386;myHost;−1;−1
 TeeFilter −2(OperationExecutionRecord) 1397507349347544442;public void kieker.examples.userguide.
        ch2bookstore.manual.Catalog.getBook(boolean);<no−session−id
        >;−1;1397507349345363211;1397507349347493714;myHost;−1;−1
 TeeFilter −2(OperationExecutionRecord) 1397507349349789220;public void kieker.examples.userguide.
        ch2bookstore.manual.Catalog.getBook(boolean);<no−session−id
        >;−1;1397507349347593226;1397507349349720521;myHost;−1;−1
 TeeFilter −2(OperationExecutionRecord) 1397507349352170480;public void kieker.examples.userguide.
        ch2bookstore.manual.Catalog.getBook(boolean);<no−session−id
        >;−1;1397507349350004459;1397507349352120991;myHost;−1;−1
 TeeFilter −2(OperationExecutionRecord) 1397507349354413163;public void kieker.examples.userguide.
        ch2bookstore.manual.Catalog.getBook(boolean);<no−session−id
        >;−1;1397507349352218753;1397507349354363380;myHost;−1;−1
Apr 14, 2014 10:33:08 PM kieker. analysis . AnalysisController  terminate
INFO: Terminating analysis .
Apr 14, 2014 10:33:08 PM kieker. analysis .analysisComponent.AbstractAnalysisComponent.FSReader−1
        terminate
INFO: Shutting down reader.
```

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

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

```
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
Apr 14, 2014 10:35:04 PM kieker.monitoring.core.configuration.ConfigurationFactory
    createSingletonConfiguration
INFO: Loading properties from properties file in classpath: 'META-INF/kieker.monitoring.properties'
Apr 14, 2014 10:35:04 PM kieker.monitoring.core.controller.MonitoringController createInstance
INFO: Current State of kieker.monitoring (1.9) Status: 'enabled'
        Name: 'KIEKER'; Hostname: 'myHost'; experimentID: '1'
JMXController: JMX disabled
 RegistryController: 0 strings registered.
TimeSource: 'kieker.monitoring.timer.SystemNanoTimer'
        Time in nanoseconds (with nanoseconds precision) since Thu Jan 01 01:00:00 CET 1970'
ProbeController: disabled
 WriterController:
        Number of Inserts: '0'
        Automatic assignment of logging timestamps: 'true'
Writer: 'kieker.monitoring.writer.filesystem.AsyncFsWriter'
        Configuration:
                kieker.monitoring.writer.filesystem.AsyncFsWriter.flush='true'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.maxLogSize='-1'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueFullBehavior='0'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.MaxShutdownDelay='-1'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.maxEntriesInFile='25000'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.bufferSize='8192'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.maxLogFiles='-1'
                kieker.monitoring.writer.filesystem.AsyncFsWriter.customStoragePath=''
                kieker.monitoring.writer.filesystem.AsyncFsWriter.QueueSize='10000'
        Records lost: 0
        Writer Threads (1):
                Finished: 'false'; Writing to Directory: '/tmp/kieker-20140414-203504785-UTC-myHost
                    -KIEKER'
Sampling Controller: Periodic Sensor available: Poolsize: '0'; Scheduled Tasks: '0'
Apr 14, 2014 10:35:04 PM kieker.monitoring.core.registry.ControlFlowRegistry <clinit>
INFO: First threadId will be 1063271724524503040
```

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

Bibliography

[1] Hyperic, Inc (2011). Hyperic SIGAR API. `http://www.hyperic.com/products/sigar`.

[2] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin (1997). Aspect-oriented programming. In *Proceedings of the 2007 European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer.

[3] Kieker Project (Oct. 2013). *Kieker 1.8 User Guide*. Software Engineering Group, Kiel University, Kiel, Germany. `http://kieker-monitoring.net/documentation/`.

[4] Kieker Project (2013). Kieker web site. `http://kieker-monitoring.net/`.

[5] Oracle (2011). Java Messaging Service (JMS). `http://www.oracle.com/technetwork/java/jms/`.

[6] OracleJava management extensions (jmx) technology. `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`.

[7] Oracle (2011). Java Servlet Technology. `http://www.oracle.com/technetwork/java/index-jsp-135475.html`.

[8] SpringSource (2011). Spring. `http://www.springsource.org/`.

[9] The Apache Foundation (2011). Apache CXF. `http://cxf.apache.org/`.

[10] The Eclipse Foundation (2011). The AspectJ Project. `http://www.eclipse.org/aspectj/`.

[11] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst (Nov. 2009). Continuous monitoring of software services: Design and application of the Kieker framework. Technical Report TR-0921, Department of Computer Science, University of Kiel, Germany. `http://www.informatik.uni-kiel.de/uploads/tx_publication/vanhoorn_tr0921.pdf`.

[12] A. van Hoorn, J. Waller, and W. Hasselbring (Apr. 2012). Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM.