Performance Monitoring of Database Operations

Master's Thesis

Christian Zirkelbach

July 26, 2015

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Florian Fittkau

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Kiel,

Abstract

The complexity of software projects and related software systems advances and likewise the quantity of data increases. Therefore within the area of software development, performance engineering is a popular topic. This is usually connected to performance tuning of a software system. Performance problems are often related to database operations, as complex database queries challenge the software engineer to achieve efficiency.

In order to aid the process of analyzing performance problems, in the context of database communication, it is useful to conduct a performance analysis focused on database statements, which are executed by the software system. Most tools, which support a performance analysis, cover just business operations, e.g., operation calls in Java. So seeking a suitable tool for this purpose is challenging, as we focus on database communication and software, which is freely-available.

In this thesis, we present an aspect-oriented approach for performance monitoring of database operations, in order to support software engineers in conducting a performance analysis. As we developed our monitoring approach in a generic manner, we allow several monitoring tools to use our generated monitoring records as input data for posterior analysis and visualization purposes. We verified the versatility of our monitoring approach through the integration into Kieker Trace Diagnosis, a graphical user interface for displaying program traces, followed by another tool, ExplorViz. As proof of concept, we investigated the first usage of our developed software by evaluating the usability through a first experiment based on a paper-based questionnaire in combination with our developed tool. In our experiment, we observed positive feedback from our participants, which correlated with the high average correctness rates of the results. Therefore, the experiment reveals that our software has a bearing on conducting a performance analysis of database operations. As we have validated the usability of our developed tool through our conducted experiment, our software system is worthwhile to be extended in the future.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

The complexity of software projects and related software systems advances and likewise the quantity of data increases. Therefore, within the area of software development, performance engineering is a popular topic. This is often connected to performance tuning of a software system. Ideally this aspect is handled during the development process in order to fix occurring problems or bottlenecks in an early phase. It is also possible to perform this work within the final phases or even afterwards of the development process, but it is not recommended based on the experience of previous projects like Dynamod [van Hoorn et al. 2011]. Therefore, handling performance problems is an important task.

Performance problems are often related to database operations, regardless of whether an *object-relational mapping* (ORM) framework, e.g., Hibernate,[1] is used or not. When an ORM framework is employed to handle database operations, it offers the developer an easy-to-use mapping between program entities, e.g., objects in Java, and the database layer. Although an object-relational mapping tool handles almost everything automatically, there are sometimes still certain situations left, at which the developer has to make important (design) decisions. And precisely this comprises a potential for typical errors, like generating an unique primary key, even if it is unnecessary [Fowler 2002]. Another example is the difference between several databases, which are addressed by the ORM. Despite the fact, that a standard for employing the *Structured Query Language* [Oracle 2011] (SQL) exists, database vendors like Oracle and IBM handle the implementation of the standard quite different. This complicates the mapping process for the ORM and can lead to performance issues.

In order to aid the process of analyzing performance problems, especially in the context of database communication, it is useful to conduct a performance analysis focused on database statements, which are executed by the software system. Most tools, which support a performance analysis, cover just business operations, e.g., operation calls in Java. So seeking a suitable tool for this purpose is challenging, because we focus on database communication and software, which is freely-available. Therefore, we will develop an

[1] `http://hibernate.org`

appropriate approach and subsequently a tool, which is on the one hand specifically developed for our purpose, and on the other hand as much generic and versatile as possible. The latter two properties are important, hence other tools or frameworks may be able to employ our approach. The following section describes our enriched goals, which are derived from our intention to develop a performance analysis software regarding database communication.

1.2 Goals

The main goal of our thesis is to design and implement a performance monitoring tool for database operations. In this master's thesis, the observed database communication is focused on those connections, which use the Java Database Connectivity[2] (JDBC) API. Figure 1.1 is based on the common workflow of performance problem detection. More precisely, the figure illustrates our enriched workflow towards detecting performance problems, which are related to database operations. We describe the workflow in detail in the following paragraph.

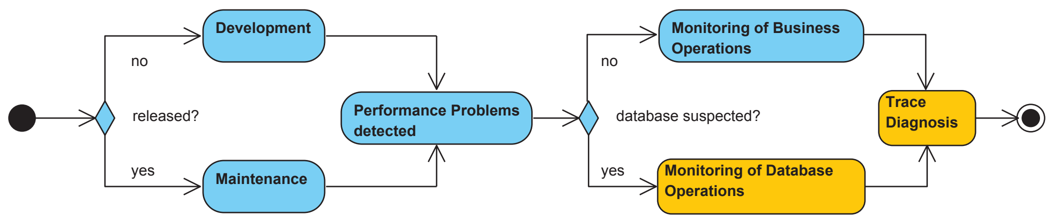

Development
no
released?
yes
Maintenance
Performance Problems detected
no
database suspected?
yes
Monitoring of Business Operations
Monitoring of Database Operations
Trace Diagnosis

Figure 1.1. Enriched performance issue detection workflow

We address both common cases within the life-cycle of a software system. Either the software is still in *development*, or it has been already released and is being *maintained*. In both cases performance problems can occur. Although the detection of performance problems within the development is possible, these are often recognized or spotted after the release of the software. Common application performance monitoring (APM) tools allow in many cases just to investigate business operations. As mentioned previously, database operations are also a frequent cause for performance issues. Therefore, we will provide a software system, which allows a developer to check the performance of database operations within his software, in order to detect potential performance issues.

Our solution is composed of a *monitoring* component, which logs data of an instrumented software system during its execution, an *analysis* component, which processes and filters

[2]http://www.oracle.com/technetwork/java/javase/jdbc/index.html

the recorded data, and a *visualization* component. The latter component allows the user to handle the processed data, so he can analyze the database communication regarding his software for performance issues. The *analysis* and *visualization* components will be integrated into Kieker Trace Diagnosis, a graphical user interface for displaying program traces. Within the entire illustrated workflow, our concrete contribution is to develop an approach for *Monitoring of Database Operations* and to integrate it into *Trace Diagnosis*, which are tagged yellow. We split our main goal into four consecutive subgoals. These subgoals are described in the following.

G1: Identification of Performance Analysis Methods and Tools

The first goal is the identification and discussion of related monitoring approaches within the research body or industrial context. Furthermore, necessary technologies need to be identified and compared. Finally, we will choose an existing monitoring technique or design our own approach based on a sufficient technology or technique. As mentioned previously, we will focus on database communication, which uses the JDBC API.

G2: Implementation of a Tool for Database Performance Analysis

The second goal covers the implementation of a database performance analysis tool. The software should allow the developer to instrument database operations within a monitored application, which are handled through a JDBC driver. Furthermore it is necessary to capture executed SQL and prepared SQL statements and additionally their call parameters, as well as their respective execution times. Since the instrumentation is a cross-cutting concern, we do not want to mix the instrumentation code with the program code. Therefore, the monitoring component should be isolated from the observed application and designed as a Java agent.

Once the needed data is collected, the developer is able to analyze the data based on a graphical user interface. The interface should offer at least abilities to show SQL traces and their corresponding response times. Prepared SQL statements, which are precompiled statements with parameters that are substituted during an execution, are important as well, as they are often used for performance improvements. So we are interested in concrete bound values for each execution. The first task covers the implementation of a prototype based on the given requirements. If its creation is successful and applicable for the context, it will be further developed to a fully functional implementation. This part also includes the integration of the analysis and visualization components into Kieker Trace Diagnosis for displaying purposes.

G3: Generic Monitoring Approach

The third goal makes sure, that the monitoring approach is generally applicable in the manner of using the generated monitoring records as input data for several analysis and visualization tools. We intend to employ two different monitoring approaches, namely Kieker and ExplorViz, to verify our goal. We want to enable the usage of our generated monitoring records as input data for both tools, in order to allow further analysis and visualization of our monitoring data.

G4: Evaluation of the Developed Tool

The fourth and last goal of the thesis is to evaluate the developed software solution. In any case the software system will be validated by an example application like JPetStore[3] to proof its functionality. Additionally, we plan to perform a survey to verify the usability of our approach, particularly the graphical user interface. Another capability would be an evaluation within an industrial context, in order to verify the applicability and especially the provided user interface. A commercial information system would be a great application for such tests.

1.3 Document Structure

The remainder of the thesis is structured as follows. In Chapter 2 an overview of foundations of our research subject and the used technologies is given. In chapters 3 and 6, our approach to accomplish our defined goals is presented. Subsequently, chapters 7 and 10 cover the implementation of our approach. In order to validate our implementation, we conduct an evaluation and present the results in Chapter 11. We provide a comparison to related work in Chapter 12. Finally, in Chapter 13, we outline and conclude our work.

[3] `http://blog.mybatis.org/p/products.html`

Chapter 2

Foundations and Technologies

The following chapter provides an overview of foundations and used technologies within this thesis and is divided into two same-named sections. We begin with basic definitions, continue to describe relevant technologies and standards, and close with tools and frameworks, which are necessary for working on the research subject of our thesis.

2.1 Foundations

This section gives an overview of foundations, which are used in this thesis.

2.1.1 Dynamic Analysis

Dynamic analysis comprises the process of analyzing a running program with focus to its deriving properties, which are valid at least for one or more executions of a program [Ball 1999]. This analysis step is often based on an instrumentation of a software or its compiled or interpreted program code. Compared to static analysis, it provides advantages like detecting unused parts of the software, often referred to as *dead code*, and allows to make better assumptions about the behavior of the program [Ball 1999]. The gathered information allows us to understand the software in a detailed way, especially with the purpose to modify it when performance problems occur.

2.1.2 Reverse Engineering

Reverse engineering contains a wide range of methods and tools with reference to understand existing software and the ability to modify it [Canfora Harman and Di Penta 2007]. Regarding software engineering, the term was defined by Chikofsky and Cross [1990] as the process of "*analysing a subject system to (i) identify its current component and their dependencies and (ii) to extract and create system abstractions and design information*". Based on this definition the main target of our reverse engineering process is to analyze the system in order to aid the task of program comprehension. Many tools were developed in the past years to support this task with capabilities to explore, manipulate, analyze, summarize and visualize software artifacts [Müller et al. 2000]. Figure 2.1 illustrates the differences between forward engineering and reverse engineering. The common process of forward engineering, within the software engineering domain, describes the process from high-level abstractions and designs towards a concrete physical implementation of a software system. In comparison, reverse engineering is quite the opposite, it characterizes the analysis of a previously engineered software system in order to identify its underlying architecture and individual software components, which the software is composed of. In this thesis, we employ reverse engineering to discover related database operations of an observed software system within our dynamic analysis.

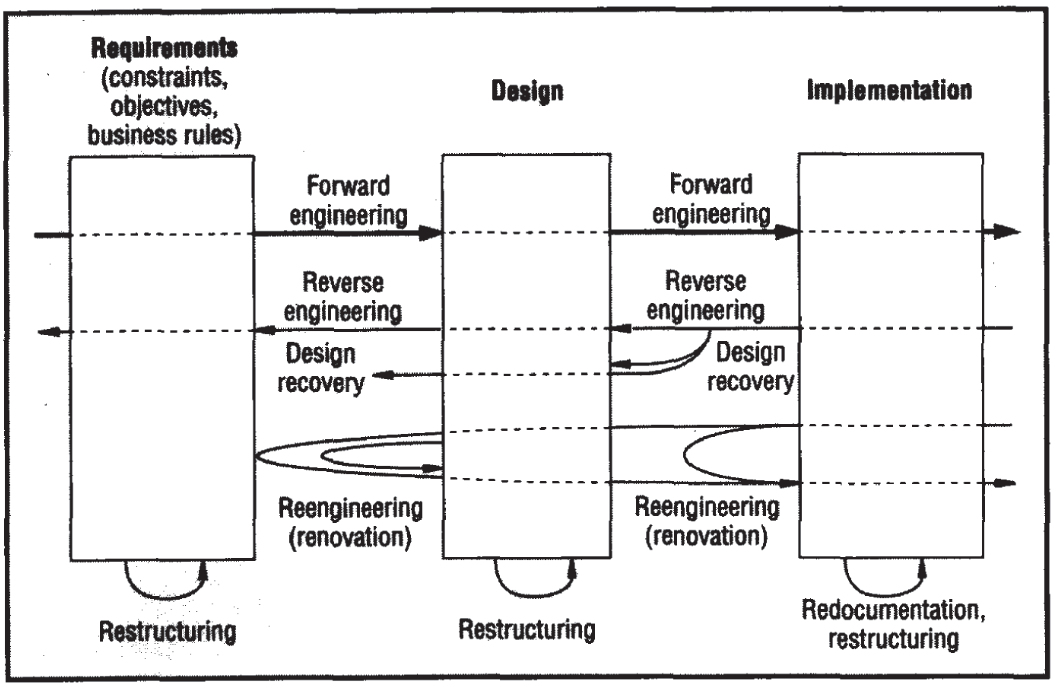


Figure 2.1. Reverse engineering in comparison to forward engineering [Müller et al. 2000]

Furthermore, we want to provide information about reverse engineering using dynamic analysis in detail. At first sight the process of reverse engineering utilizes the concept of dynamic analysis in order to get the required information to make abstractions of the software. At second glance it provides much more. It supplies useful information, especially the capability to perform architecture reconstructions, for the developer to understand the program. This allows him to make major changes to the software [Ducasse and Pollet 2009]. If it is possible to create and update this knowledge continuously, upcoming changes would not need costly reverse engineering efforts. Hence, it is appropriate to integrate this method into the development process. Additionally, this allows us to identify and instrument executed database operations within the development phase, in order to optimize the performance. Gathering such useful information is the basis for the upcoming section.

2.1.3 Performance Analysis

If we employ static or dynamic analysis, we can use this information for a performance analysis. Within our work, we define the overall process of estimating performance as a "*study of the performance of computer systems and networks attempts to understand and predict their time dependent behavior*" [Pooley 2000]. In order to aid the detection of occurring performance problems we need to use appropriate tools. Such a tool may expose a performance problem or even a bottleneck, which has a negative impact on the performance of a software system [Neilson et al. 1995]. Especially in the area of analyzing database communication within a software there is a lack of good, available tools.

In the context of software engineering, a bottleneck is a single point or part within a software, that limits the total system performance [Neilson et al. 1995]. Bottlenecks can occur on the one hand as device bottlenecks, like storage, memory, or CPU, and on the other hand as bottlenecks within the software that may consist of locking of data or blocking of processes when working in parallel. Database communication can also be a potential bottleneck, caused by poor design or increasing workload. Dynamic analysis is a good technique to find such performance bottlenecks as it is examining a well-defined use-case combined with a specific request towards the software system. Already observed suspicious parts of the software or possible problematic components, which may cause performance problems, can be monitored and analyzed with regard to bottleneck detection [Xu et al. 2010]. An general introduction in the terminology of performance metrics can be found in [Sabetta and Koziolek 2008].

2.1.4 Aspect-Oriented Programming

There are many kinds of instrumentation techniques existing. As there are various approaches, they can mainly be divided into two groups. The first group contains methods for a manual instrumentation. During a performance analysis, there are monitoring probes placed within the source code or around interesting parts of the software. As this is a very extensive and time-consuming strategy, it is not advisable to use this method within larger projects. The second group includes methods for setting up monitoring probes, without the need to interfere with the source code. One well-established method is aspect-oriented programming (AOP). Kiczales et al. [1997] came up with the initial concept of AOP. The basic idea of AOP is to improve the separation of concerns within software. Therefore, regarding an instrumentation of a software system, it is recommended to separate the instrumentation code or probes from the source code. This process is handled through weaving the monitoring code into the source code.

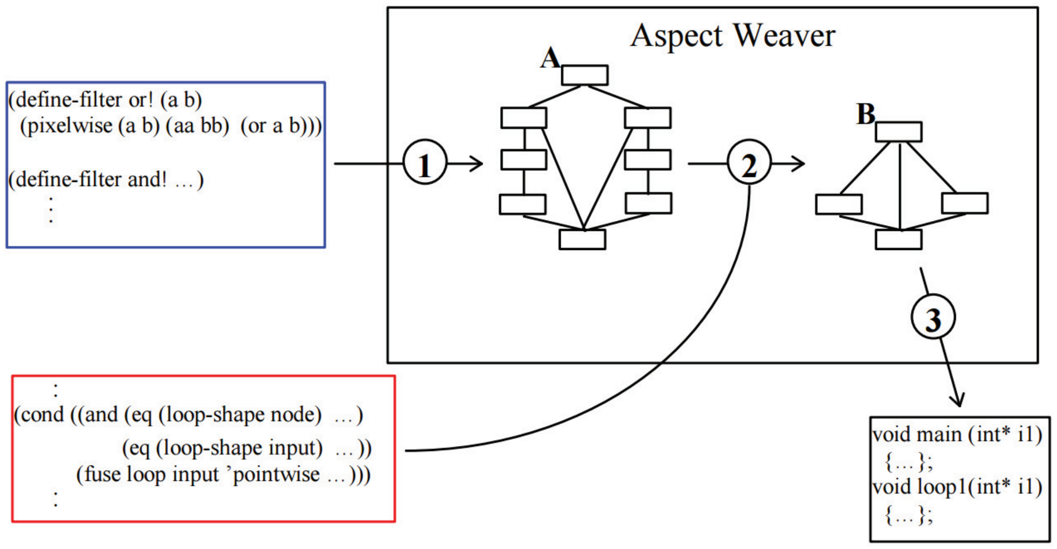

Aspect Weaver
A
B
1
2
3
(define-filter or! (a b)
(pixelwise (a b) (aa bb) (or a b)))
(define-filter and! …)
(cond ((and (eq (loop-shape node) …)
(eq (loop-shape input) …))
(fuse loop input 'pointwise …)))
void main (int* i1)
{…};
void loop1(int* i1)
{…};

Figure 2.2. Example of an aspect weaver for image processing [Kiczales et al. 1997]

Figure 2.2 illustrates an example of weaving code into an existing software. In this case the aspect weaver takes the component (blue) and aspect programs (red) as input, and emits another program as output. Tonella and Ceccato [2004] also used aspect-oriented programming for an aspect identification in existing code as a supporting technique within dynamic code analysis.

2.1.5 Usability

The International Organization for Standardization (ISO) defines usability as "The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use" (EN ISO 9241). In our thesis, usability describes the ease of use and learnability of our developed software application.

2.1.6 Usability Testing

Usability testing describes a technique, which is used within user-centered design, to evaluate a software by testing it on users [Karat 1997]. In contrast to other evaluation methods, which do not involve real users, it allows to get direct feedback from the users, as they use the software. Although we determined the requirements of our software, we can not guarantee, that these are under-specified. A real user of a software system might have different or additional needs. Since we are interested in developing a software solution focused on the requirements of the user, we need to verify, that we meet those needs [Coble et al. 1997]. This circumstance can be verified by usability testing.

2.1.7 Usability and User Experience Questionnaires

In order to verify the usability of a software, often an evaluation using an evaluation method, e.g., performing a questionnaire, is used. One of the advantages is being inexpensive, as no specific testing equipment is needed, and we reveal, what real users think of our software. One of the most popular web questionnaires is the System Usability Scale (SUS), as it is a quick and validated test. It is the most widely adopted test for perceived usability evaluation of software [Tullis and Albert 2008; Sauro and Lewis 2012]. Furthermore it seems to yield reliable results, across various sample sizes [Tullis and Stetson 2004].

2.2 Technologies and Tools

This section provides an overview of technologies and tools, which are used in this thesis.

2.2.1 JDBC

JDBC stands for Java Database Connectivity, which embodies a Java API for database-independent communication between Java and several databases. Since its introduction in 1997, the JDBC API has become widely accepted and implemented. Today it constitutes an industry standard [Oracle 2011] for the aforementioned purpose. The JDBC library offers a wide range of different possibilities for several tasks, e.g., handling database connections. This includes defining and executing SQL statements or delivering and displaying result sets. The ordinary process for using JDBC as a tool of database communication involves a JDBC driver. The driver enables a Java application to interact with a manufacturer-specific database. A driver is assigned to one of four categories. These are the *JDBC-ODBC Bridge Driver*, the *Partial JDBC Driver*, the *Network-Protocol Driver*, which is also known as the MiddleWare Driver, and the *Database-Protocol Driver* (Pure Java Driver).

Figure 2.3 illustrates the two most commonly used driver types. On the one hand there is the *MiddleWare Driver*, which converts the JDBC calls into the middleware protocol. Subsequently, the middleware translates it to a vendor specific DBMS protocol. A huge advantage is the connectivity to several different databases. On the other hand there is the *Partial JDBC Driver*. It transforms the JDBC calls into vendor specific calls on the client API (DB ClientLib) like Oracle or DB2. Afterwards the converted calls are used to communicate with the DBMS.

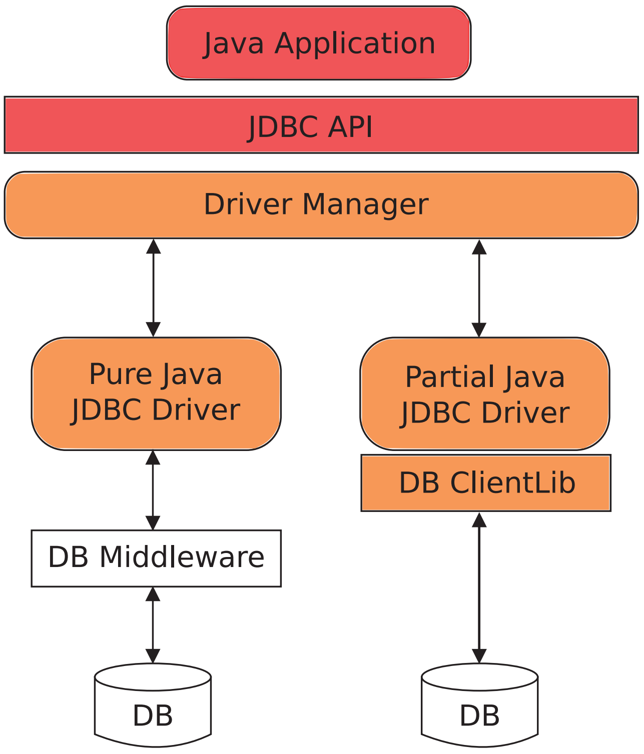


Figure 2.3. Common JDBC driver types

2.2.2 AspectJ

Based on Aspect-Oriented-Programming, which was mentioned in the previous section, Kiczales et al. [2001] developed AspectJ[1] as an aspect-oriented extension for Java. AspectJ can be employed to define cross-cutting concerns like security or monitoring requirements, without interfering with the source code. The necessary code or modification is woven into the program code. The weaving can be done in three different ways. The first option is to use Compile-Time-Weaving, which generates already instrumented class files from the source code. The second option is Binary-Weaving, which is used, if the source files are not available. The third and last option is Load-Time-Weaving, which modifies the Binary-Weaving, so that the instrumentation takes not place, until the Java Virtual Machine (JVM) is applied. The latter option can be applied by Java agents or wrapper scripts.

[1] `http://eclipse.org/aspectj`

2.2.3 Kieker Monitoring Framework

Kieker is a monitoring framework, which provides dynamic analysis capabilities for monitoring and profiling the runtime behavior of a software system in order to enable Application Performance Monitoring (APM) and Architecture Discovery. It collects and analyzes monitoring data on different abstraction levels. Furthermore, it measures operation response times and traces using dependency graphs [Hasselbring 2011] of a software run [van Hoorn et al. 2009; 2012].

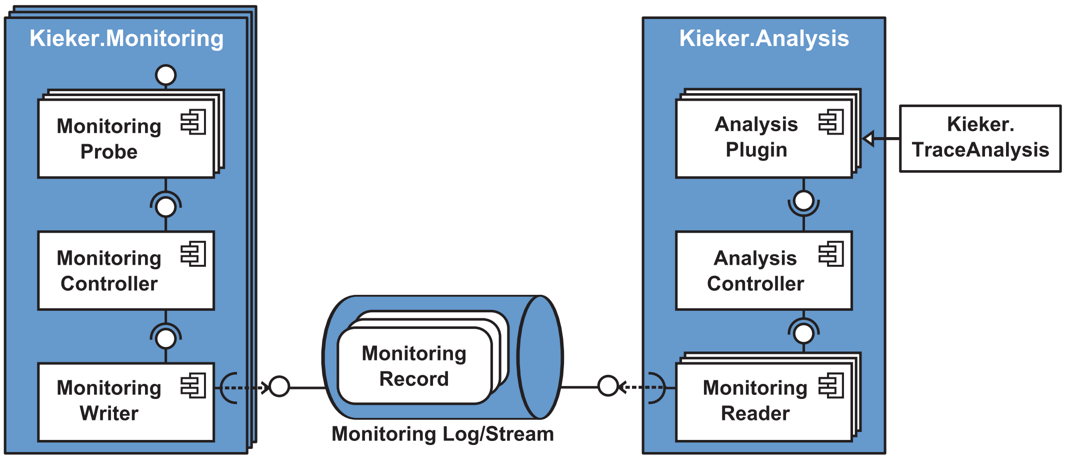


Figure 2.4. Overview of the framework components [Project 2013]

The Kieker Monitoring Framework is divided into a monitoring part, referred to as *Kieker.Monitoring*, and an analysis part, referred to as *Kieker.Analysis*. Figure 2.4 shows the components of the framework and their relations. While *Kieker.Monitoring* provides capabilities for obtaining and logging measurements from software systems, *Kieker.Analysis* supplies an appropriate infrastructure for analyzing this measurements. The exchange of data between the monitoring and analysis components is handled through monitoring records, which are generated by the *Monitoring Writer* and then passed to the *Monitoring Reader* via a file system or stream. The framework is extensible for individual purposes. It also provides for previously analyzed monitoring data several visualizations, such as Unified Modeling Language (UML) sequence diagrams and dependency graphs.

2.2.4 Instrumentation Record Language

The Instrumentation Record Language (IRL),[2] an extension to the Kieker Monitoring Framework, has been developed since 2013 as an approach for a model-driven instrumentation [Jung et al. 2013]. The language can be used to generate records based on a well-defined

[2] http://www.oiloftrop.de/code-research/instrumentation-record-language-release-1-0

declaration for specific purposes, particularly when none of the existing records is sufficient for the use case. Furthermore, it provides an abstract data model notation and integrates well as an plug-in into the Eclipse editor and generates data structures and serialization code for C, Perl, and Java. Additional generators to support other languages can be added. It is also possible to use generators without the Eclipse editor, as there is also a Command-Line-Interface (CLI) in form of a standalone compiler. In our thesis, we use the IRL to define compatible Kieker records for the integration into Kieker Trace Diagnosis.

2.2.5 TeeTime

TeeTime[3] is a Pipe-and-Filter Framework for Java. It allows to define abstract Pipe-and-Filter architectures, in order to use them for an analysis based upon information in form of data sources. It also provides ready-to-use primitive and composite stages, which can be extended by the developer. Although it is compatible to Kieker, it is not limited to it [Wulf et al. 2014]. We use TeeTime for the analysis of our generated monitoring logs.

2.2.6 JPetStore

JPetStore[4] is a web application, based on MyBatis 3, Spring 3, and Stripes. The Kieker project uses JPetStore as an example application for testing purposes. It uses a straight-forward architecture for accessing the database, which includes a functional SQL mapping framework. The default configuration uses the database HSQLDB (HyperSQL DataBase),[5] which is also written in Java. In our thesis, we use the JPetStore, as it uses an integrated database in combination with JDBC, as an example application to verify our approach.

2.2.7 Kieker Trace Diagnosis

Kieker Trace Diagnosis is an additional tool for further analysis and visualization purposes within the Kieker Monitoring Framework. As it is still under development, it has not been released yet. Therefore, a development snapshot is the only available version at the moment. The tool allows the user to interact with analyzed data and offers functions like filtering and sorting for monitored data. Furthermore, the tool provides a visualization of recorded program traces in form of tree views. Kieker Trace Diagnosis offers a kind of visualization, which is sufficient for our purpose of illustrating database calls. As there are currently only business operations, e.g., called operations within a Java program, supported, we integrate our analysis and visualization components into Kieker Trace Diagnosis, in order to reach our defined goals. Furthermore, we enhance the existing tool and extend its functionality.

[3]http://teetime.sourceforge.net

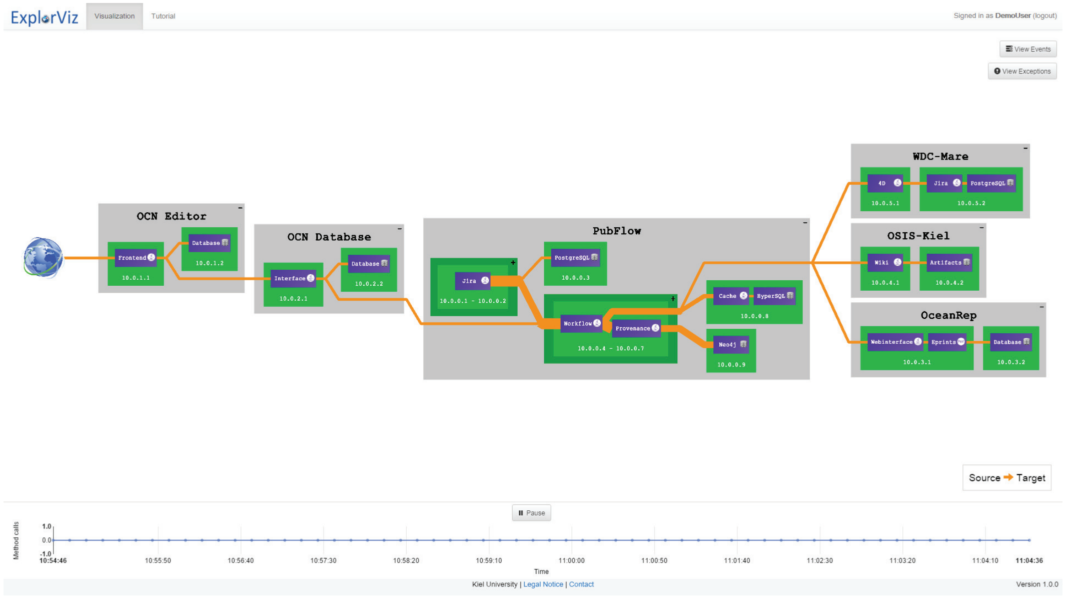
[4]http://blog.mybatis.org/p/products.html

[5]http://hsqldb.org

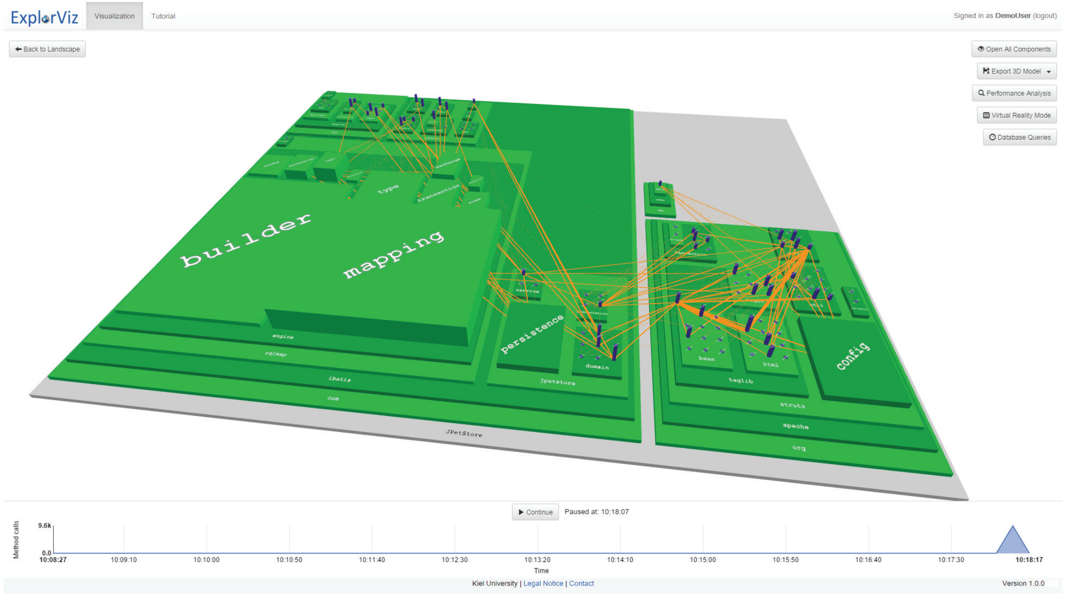
2.2.8 ExplorViz

ExplorViz[6] is a live trace visualization tool for large software landscapes [Fittkau et al. 2013b]. The tool focuses on system and program comprehension. It uses dynamic analysis techniques to monitor traces for large software landscapes and offers two different visualization perspectives, namely a landscape and an application level perspective. The landscape perspective, illustrated in Figure 2.5a, employs a notation similar to the UML and provides an overview of the software landscape. Furthermore, it offers additional abstraction levels to highlight communication within observed software systems. The application level perspective, displayed in Figure 2.5b, utilizes the 3D software city metaphor, which was introduced by Knight and Munro [2000], and allows the user to interact with a generated 3D software model for enhanced program comprehension [Fittkau et al. 2013a; 2015, b].

[6] `http://www.explorviz.net/`



(a) Landscape perspective



(b) Application level perspective

Figure 2.5. ExplorViz: Landscape and application level perspective [Florian Fittkau 2015]

Chapter 3

Approach for Database Monitoring

The following chapter covers our approach, based on our previously defined goals, and is divided into four, consecutive sections. At first we provide an overview of our enriched approach, followed by presenting the developed architecture of our software. Afterwards, we describe our designed software system and present the monitoring component, continue with the analysis component, and close with the visualization component.

We present an overview about the components within our related software system. Figure 3.1 shows the main components of our developed approach. The software system is composed of four components, which accomplish different, consecutive tasks. The components are the *Database*, the *Monitored application*, the *Monitoring component*, and *Kieker Trace Diagnosis*, which includes the integrated analysis and visualization components. Our approach is designed straight-forward, as our communication goes from the *Database*, over the *Monitored Application*, through our *Monitoring Component*, and finally towards *Kieker Trace Diagnosis*. The software system is divided into two separate parts, the *Existing Software*, which consists of the *Database* and the *Monitored Application* on the one hand, and our developed *Approach*, including the *Monitoring Component* and integrating our components into *Trace Diagnosis*, on the other hand. More precisely, our tool instruments the existing software system through AOP and processes the resulting data.

As we are interested in database communication related to JDBC, we first have to address the JDBC interface or API, which is provided by Java. The JDBC API offers a wide range of different Java classes, which are used to communicate with a vendor-specific database, in order to perform queries. Based on a vendor-specific JDBC driver, we are able to monitor such database operations. However, we still need a way to set-up our instrumentation code around these Java classes.

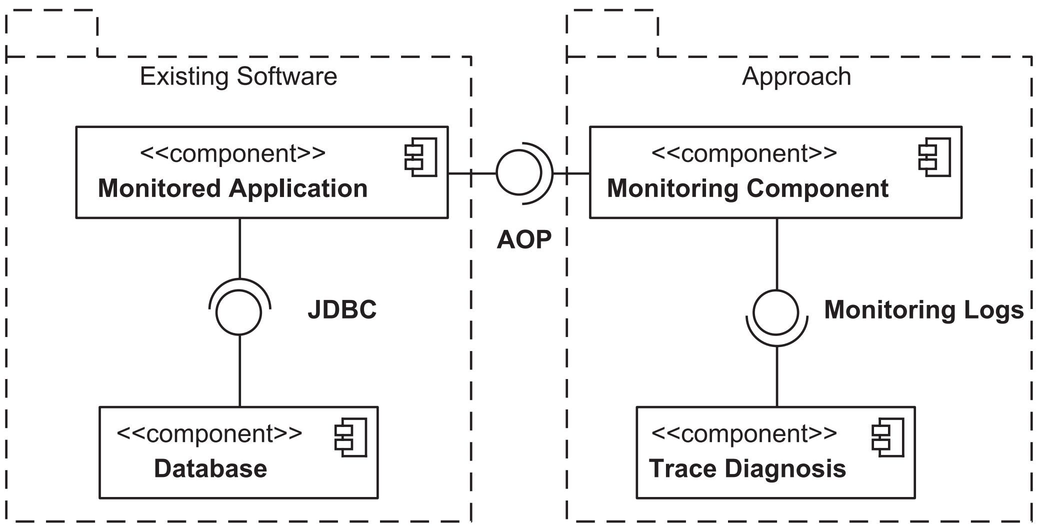


Figure 3.1. Architecture of our software system as component diagram

We employ AspectJ[1] for this task. We weave monitoring probes into the monitored Java software without interfering with the program code. As we develop our monitoring component as a Java agent, the monitoring data is recorded during each execution of the software system. Once the database operations within the observed software are recorded, the monitoring component receives the data. Afterwards it generates monitoring records, in form of a log file, which is then written down to the file system. In the future we also plan to offer streams as an option for extracting monitoring logs. The monitoring log can then be loaded into Kieker Trace Diagnosis to perform an analysis and to visualize the recorded data. The following chapter describes the monitoring component in detail.

[1] `http://eclipse.org/aspectj`

Chapter 4

Monitoring of Database Operations

The instrumentation gathers data for further tasks, e.g., analysis and visualization, which are necessary to conduct an efficient performance analysis on a software system. Therefore, we present our instrumented database operations in the following sections, which are based on SQL relationships, and show afterwards our enriched monitoring approach.

4.1 Related SQL Relationships

As we observe database communication, related to SQL database operations calls handled through JDBC, we provide an overview of relevant classes, interfaces, and relationships within the *java.sql* Java package, which is presented in Figure 4.1. The figure, which is extracted from the JDBC standard, is composed of three different types – classes, interfaces, and relationships between them [Oracle 2011].

Communication protocols share the requirement for a connection, which is being established before data can be exchanged and closed, when the communication is finished. A *Connection* or session within our context also needs to be established first, before database operations can be executed on a specific database. Once the connection is created, SQL statements can be executed and return results within the context of this connection. There are basically three options, to define and execute a statement, respectively a query, within JDBC, which are slightly different. These are *Statement*, *PreparedStatement*, and *CallableStatement*. They are created by the *Connection* using the operations *createStatement*, *prepareStatement*, and *prepareCall*.

We start with the *Statement*, which is the simplest of the three options. The only possibility to employ a *Statement* is to create an object, using the constructor. Afterwards one of the following operations *execute*, *executeQuery*, or *executeUpdate* is performed, which takes a parameter of the type string. The string contains the related SQL statement, e.g., "*SELECT * FROM users;*", which returns all records of the users relation. This SQL statement cannot be modified, as it is being directly executed afterwards. The *PreparedStatement* extends the *Statement*, with the purpose to set-up an abstract SQL statement with parameters in order to precompile it. Afterwards concrete values, which are applied to the *PreparedStatement* through several setters, are used as input for those parameters (Data Types), in order

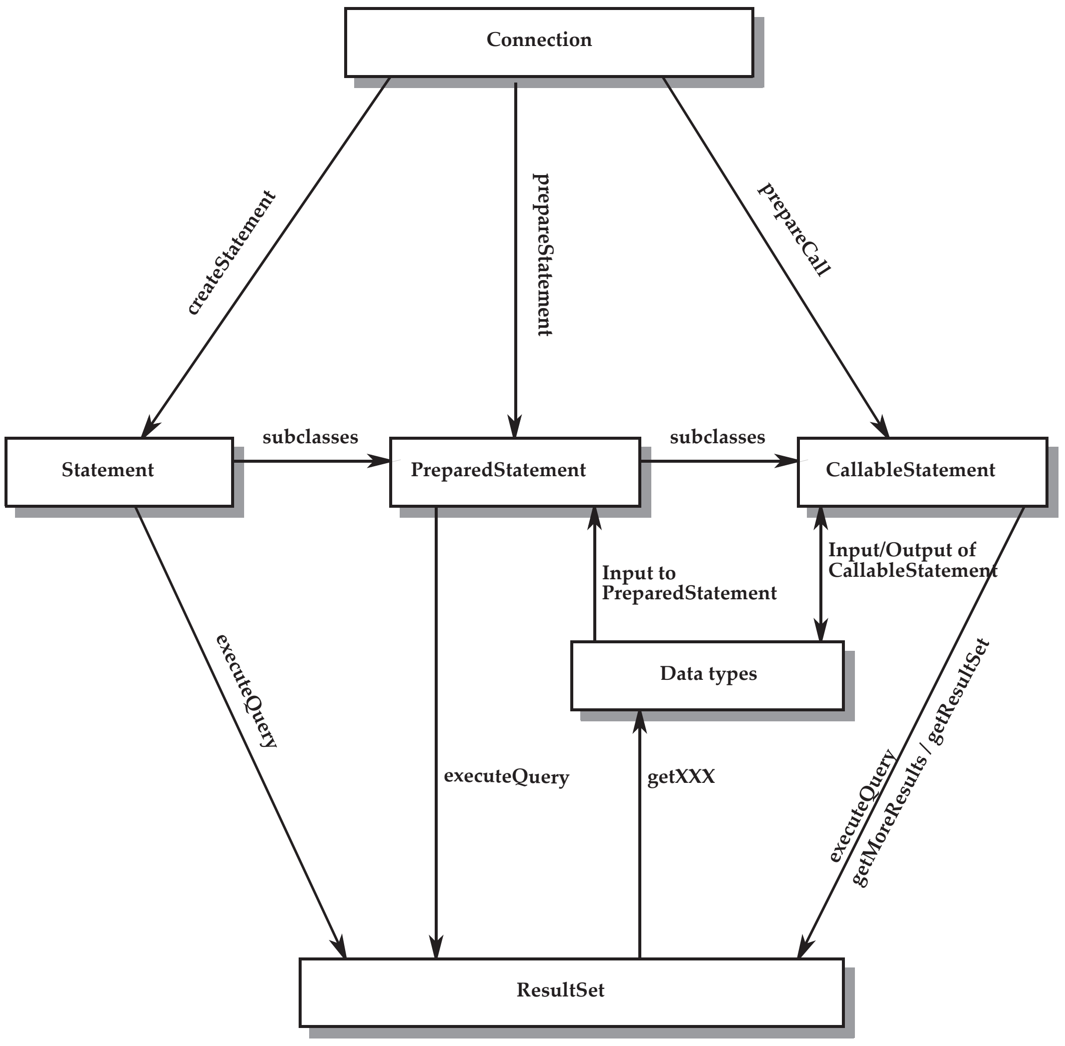


Figure 4.1. Relationships between major classes and interfaces in the *java.sql* package [Oracle 2011]

to execute a concrete prepared statement. The advantage over simple statements is an improvement of the execution time, when handling huge amounts of similar database queries. The third and last option is the *CallableStatement*. Again, it extends its predecessor, in this case the *PreparedStatement*, to enrich the prior functionality. A *CallableStatement* allows to execute *Stored Procedures*, which are developed functions that are stored within the database. These are capable to handle multiple results.

All three previously named options have the *ResultSet* in common, when they are executed through the operation *executeQuery*. If the different statements are being executed via *executeQuery*, they all return a *ResultSet*, if possible. This data type can be used to extract information, e.g., the number of affected database records. As mentioned before, there exists two other options to execute statements, namely the operation *execute* and *executeUpdate*. The first operation returns just a boolean, which indicates, whether the first result is a *ResultSet* object or not. The second operation, which is provided to update specific database entries, just returns the number of affected records due the given SQL statement. In our thesis, we focus on the two interfaces *Statement* and *PreparedStatement*, as they are the most commonly used database operations.

4.2 Generic Monitoring Approach

One of our subgoals is to develop our monitoring approach in a generic manner. We want to allow several monitoring tools to use our generated monitoring records as input data for posterior analysis and visualization purposes. Furthermore, we describe related monitoring tools and compare them. Finally, we will choose an existing monitoring tool or design our own approach based on a sufficient technology or technique. As mentioned previously, we will focus on database communication, which uses the JDBC API.

4.2.1 Choosing an Instrumentation Technique

First, we performed a literature search within the research body and industrial context. We focused on existing techniques and tools related to our context of instrumenting database operations. Furthermore, we filtered our results for approaches, which employ AOP or similar intercepting techniques. Our findings included a set of tools, which matched our purpose. Besides AspectJ, which we described in Chapter 2, there exist similar approaches. Three comparable approaches are presented in the following.

Java Proxy

Since version 1.3, Java provides an implementation of the dynamic proxy concept for interfaces (*java.lang.reflect.Proxy*).[1] If a method is invoked on the proxy, a registered

[1] `http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html`

invocation handler is being notified, which method has been executed, including its calling parameters, and on which object [Froihofer et al. 2007]. A disadvantage of this technique is, that it is limited towards classes, that implement interfaces.

cglib

Cglib[2] is a byte code instrumentation library, that provides a high level API to generate and transform Java byte code. Furthermore, it is used by AOP, testing, and data access frameworks to generate dynamic proxy objects and intercept field access. It is employed by the well-known Java framework *Spring*[3] and former by ORM framework *Hibernate*[4]. However, as there is no research literature available and the overall documentation is sparse and the future of *cglib* was quite uncertain, many projects moved away from the library towards other solutions.

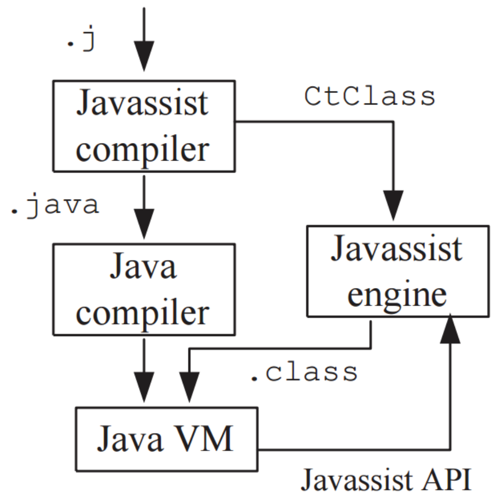


Figure 4.2. The Javassist architecture [Chiba 1998]

Javassist

Javassist,[5] which is the short version for JAVA programming ASSISTant, is another byte code manipulating library. It is still been maintained, although it was introduced in 1998 [Chiba 1998]. It allows to define new classes at runtime and to modify class files, when loaded by the JVM. Figure 4.2 presents the architecture of Javassist at the time of its initial

[2]https://github.com/cglib/cglib
[3]http://projects.spring.io/spring-frameworl
[4]http://http://hibernate.org
[5]http://jboss-javassist.github.io/javassist

release, which consists of two main components, namely the *Javassist compiler* and *Javassist engine*. The *Javassist compiler* processes annotations and the *Javassist engine* compiles these, in order to access them through the *Javassist API*. Summarized, the tool can be compared with the Java proxy, with reference to the inception mechanism.

4.2.2 Concrete Instrumentation Approach

Although we presented three different tools, which are similar to AspectJ, we choose the latter for our monitoring approach. As we mentioned previously, AspectJ is suitable for our purpose. Compared to the other three tools, it is easy-to-use and offers a straight-forward handling for instrumenting Java software, without interfering with the program code. Developed definitions can be extended or modified for other purposes. Interesting parts of the software, which are not monitored at the moment, can be covered by additional monitoring probes. Due its good documentation and publications, combined with our prior experience using AspectJ in previous projects, we decided to choose it as a basis for our monitoring approach. Therefore, we present our versatile monitoring approach, in form of a component diagram, in Figure 4.3.

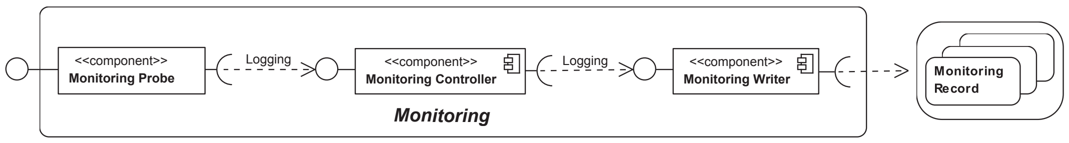


Figure 4.3. Monitoring components

Our monitoring component is composed of three components, namely *Monitoring Probe*, *Monitoring Controller*, and *Monitoring Writer*. The combination of them characterizes a workflow. The first step within the workflow describes monitoring database operations within an observed software system via *Monitoring Probes*. They are integrated into the program code via AspectJ, which was mentioned before. These probes are injected to instrument the classes and interfaces, which were shown in Figure 4.1. Relevant database operations are executed through these junctures, so it is an appropriate position within the monitored software for interception. This includes executed SQL statements, returned values, e.g., ResultSets, and related response times.

As soon as the instrumentation is injected into the program code and the software system is executed, the probes send their recored data to the *Monitoring Controller*. Afterwards, the *Monitoring Controller* creates records based on this data. More precisely, it generates one monitoring record for entering an above-mentioned juncture (before event), e.g., *Statement*, and another one for leaving it (after event). The duration between the first and the second record is later used to calculate the response time. Afterwards, the generated monitoring records are forwarded to the *Monitoring Writer*. This component allows to structure and export our previously recorded monitoring data, in form of a monitoring log file, to the file system. The monitoring log file, which is composed of many monitoring records, is defined as a self-created, well-structured, and human-readable representation. The structure of our monitoring records is presented in the following:

- record type: differs between before or after event
- timestamp: represents date and time of the record
- operation name: full Java class name
- return type: e.g., ResultSet, boolean or int
- return value: e.g., number of affected database records
- operation arguments: e.g., SQL statement

This monitoring record structure forms the basis for our following tasks, namely analysis and visualization. Furthermore, we offer the ability to export our records into at least one other monitoring representation, namely the Kieker record structure. The latter decision is necessary for employing our recorded data as input data for Kieker and ExplorViz, which allows to import Kieker monitoring log files. We choose these two monitoring tools in order to verify the versatility of our monitoring approach.

Chapter 5

Analysis of Database Records

In the following chapter, we describe how we analyze the recorded monitoring data, process it, and prepare it for the later described visualization.

As mentioned before, our monitoring records are well-structured and contain various information, regarding monitoring a concrete database operation. As we want to analyze and later visualize this data, we need to process the recorded data through a defined workflow. This includes tasks like merging, filtering, and aggregating. Similar monitoring tools, e.g., ExplorViz, perform different analysis steps in the same manner [Fittkau et al. 2013b; 2015, b]. Although in this paper the analysis is optimized for high-throughput, it defines an appropriate template, which we can use for our purpose. Additionally, we employ the Pipe-And-Filter pattern for our analysis. We utilize this pattern, in order to create a flexible, maintainable, and expendable analysis architecture for processing our previous recorded data [Gamma et al. 1994]. Therefore, we employ an implementation of this pattern, namely the Pipe-And-Filter framework *TeeTime*, and construct our analysis in a suitable manner [Wulf et al. 2014]. This implies to develop an analysis workflow, which can be separated into several processing units. These will be connected afterwards, so they can be executed consecutively. In the following, we present our developed approach and describe the involved, different processing tasks and their relationships.

5.1 Analysis Approach

Our analysis approach can be divided into two, consecutive abstract workflow processes. The first process describes the generic processing of our monitoring records. Once the first process is completed, the second process is executed. The second process handles the specific database records, which can be extracted from the recorded monitoring data. Both processes are explained in the upcoming sections.

5.1.1 Generic Monitoring Record Processing

The previously recorded monitoring data needs to be initially processed, such that the gathered data can be further analyzed. Therefore, we developed a process, which loads this data and prepares it for subsequent tasks. In Figure 5.1, the architecture is illustrated, in form of a component diagram. The diagram is composed of four components, namely the *Log Reader*, the *Record Filter*, the *Record Transformator*, and the *Record Merger*. As mentioned before, we focus on flexibility and extendibilit. Therefore, an implementation of one of these four components can be maintained and replaced easily.

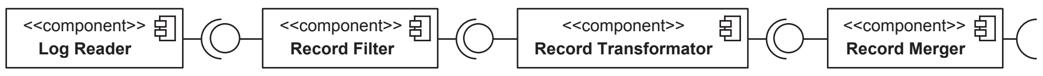


Figure 5.1. Analysis: Generic Monitoring Record Processing

The input data of our generic monitoring processing is the recorded monitoring data. As our concrete monitoring approach creates a monitoring log file within the file system, this log file needs to be loaded into the analysis process for further tasks. This task is done by the *Log Reader* component. It loads the monitoring log and provides this data for the next component, the *Record Filter*. The *Record Filter* analyzes and matches a monitoring record type, e.g., a before or after event record (entering or leaving a database operation), and creates a related database record object. This created record object is then transformed into a database operation call object through the *Record Transformator*.

Once the monitoring records have been converted into record objects, they can be used for the ongoing analysis. As we have basically two record types, before and after event, for a single executed database operation, we need to merge these two record objects into one record, in particular a database operation call object. This task is performed by the *Record Merger*. The latter task completes the generic monitoring record processing and allows to continue our analysis with handling specific database operations in the following subsection.

5.1.2 Specific Database Call Handling

Once we have completed the previously mentioned process chain, we can further process these database operation call objects. This allows us to extract the contained information to identify specific database operations, e.g., *Statements* and *Prepared Statements*, which we described in Section 4.1. The involved components are shown in Figure 5.2.

Since we intend to employ a Pipe-And-Filter-Architecture, each task or filter has exactly one input port and one output port. This characteristic is inconvenient, because at some places within our enriched process, we need the option to define more than one output port for a filter. A common solution for this problem is to employ a *Distributor*, which allows to define multiple output ports for a filter [Wulf et al. 2014]. We utilize this kind of filter to extract the results of a filter and to further process the data, in order to pass it through a pipe to another filter. This technique is similar to the command *tee* within

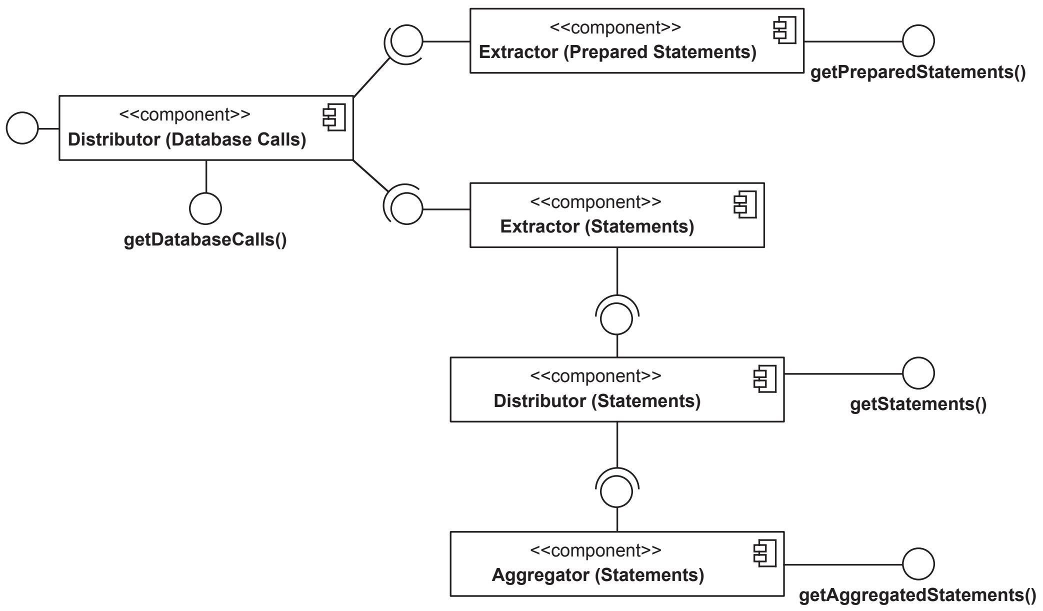


Figure 5.2. Analysis: Specific Database Call Handling

UNIX systems, [1] which can be used to forward data to another process on the one hand and redirect the data towards a file on the other hand.

With reference to Figure 5.2, the processed monitoring records from the previous section have been converted to database operations calls. These calls are the foundation for our ongoing filtering tasks. Based on the information within these calls, e.g., operation name or call argument, we can identify the type of database operation and the duration. Since we limited our approach to *Statements* and *Prepared Statements*, these are the subordinate database operations, we are interested in.

The first component within our process system is the *Distributor*, which handles the previously mentioned database calls. He distributes the database operation calls to the following filters, an *Extractor for Prepared Statements* on the one hand, and another *Extractor for Statements* on the other hand. Additionally, we can extract the database calls, if we need them for analysis purposes through the interface *getDataBaseCalls()*. The *Extractor for Prepared Statements* examines related database operation calls and extracts relevant information. First we create an abstract prepared statement with variables. As we are interested in concrete executions of an abstract *Prepared Statement*, we need to perceive

[1] `http://www.unix.com/man-page/posix/1p/tee`

related executed setters, in order to reconstruct concrete executions and attach the corresponding concrete prepared statements (with bound variables). This is important, as we later visualize the hierarchy of related calls. The processed calls are accessible through the interface *getPreparedStatements()*.

A similar order of events is employed for *Statements*. The *Extractor for Statements* merges statements, namely *createStatement()* and corresponding executors, into one call. Also the overall response time for a specific execution is calculated. The processed database calls are then forwarded towards the *Distributor for Statements*, which distributes the database call objects on the one hand towards an interface, to obtain these through *getStatements()*, and on the other hand to the next filter, the *Aggregator for Statements*. As its name implies, this filter handles the aggregation of statements. It takes the previously processed statements and performs a matching based on the executed SQL statement. For each SQL statement a parent call is created. Does a statement call already exist for a parent call, then the statement call is added as a child to the parent call. This procedure allows a further analysis, e.g., calculating the average response time. Once again, the results are available through an interface (*getAggregatedStatements()*).

5.2 Integration into Kieker Trace Diagnosis

We intent to integrate our analysis component into Kieker Trace Diagnosis. More precisely, we decide to separate the analysis from our monitoring component, in order to enable a light-weight monitoring approach, which is generic and versatile. Additionally, Kieker Trace Diagnosis offers us a read-to-use architecture, on which we can build upon our visualization. Therefore, we integrate the analysis into Kieker Trace Diagnosis.

Chapter 6

Visualization of Database Records

We now present our approach towards the visualization of our monitoring results. In the following sections, we provide an introduction to visualizing execution traces, present our visualization approach, and close with our integration of the latter one into Kieker Trace Diagnosis.

6.1 Different View Options

In the software engineering domain, visualization is a popular topic, because finding an appropriate form of representation for a specific purpose is difficult. Therefore, we performed a search within the literature and present our findings in the following.

6.1.1 Dependency Graphs

There exist several representations of visualizing execution traces. One option to visualize traces are dependency graphs [Hasselbring 2011]. Based on the hierarchy of a call, these graphs can be constructed and enriched with other metrics, e.g., response times, to support tasks like conducting a performance analysis. This representation is used in the Kieker monitoring framework to visualize the underlying architecture of a software. An example is presented in Figure 6.1. The shown dependency graph represents the aggregated call hierarchy of an execution of a Java program and shows the dependencies among them. Furthermore, the edges are annotated with the number of calls for a specific dependency. For our purpose, this form of representation is inappropriate because our monitored database operations have a flat hierarchy, so we would have a bulk of dependency graphs. This would be confusing and makes it impossible to analyze and compare the recorded data.

6.1.2 3D Software City Metaphor

Another approach is the 3D software city metaphor, which was introduced by Knight and Munro [2000]. The metaphor provides an overview of a software system and aids in the process of program comprehension [Wettel and Lanza 2008]. ExplorViz employs this approach and allows the user to interact with a generated 3D software model for

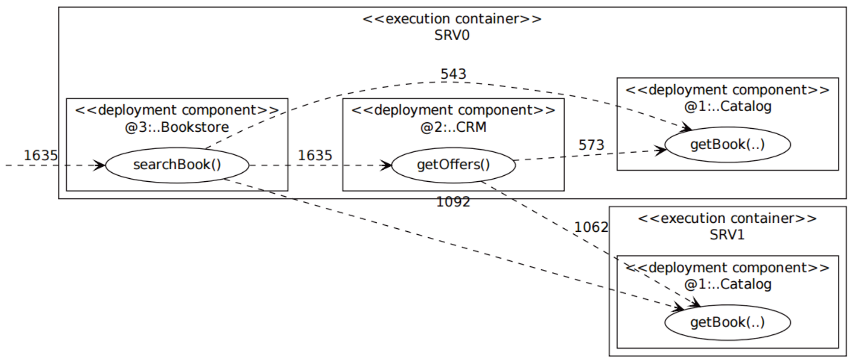


Figure 6.1. Generated operation dependency graph [Hasselbring 2011]

enhanced program comprehension based on execution traces [Fittkau et al. 2013a]. An example is shown in Figure 6.2. The figure sketches the four main components of the application JPetStore and their communication in-between. This kind of visualization is appropriate for the task of program comprehension [Fittkau et al. 2015a] and may be also suitable for visualizing database operations. However, we focus on performance analysis and as we intend to integrate our approach into Kieker Trace Diagnosis, the tool is not customizable for such a kind of 3D visualization. Therefore, in this thesis, we need a different visualization approach. However, this might be an alternative representation option in the future.

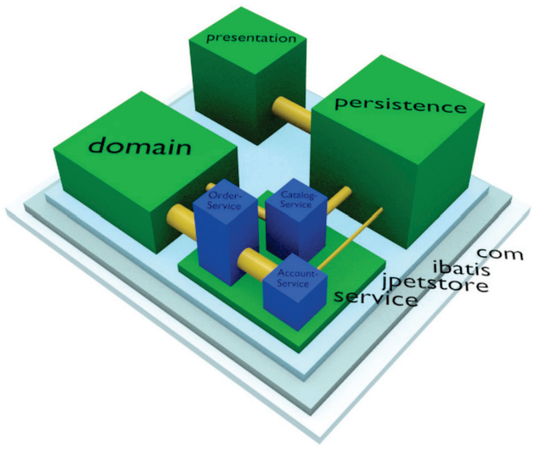
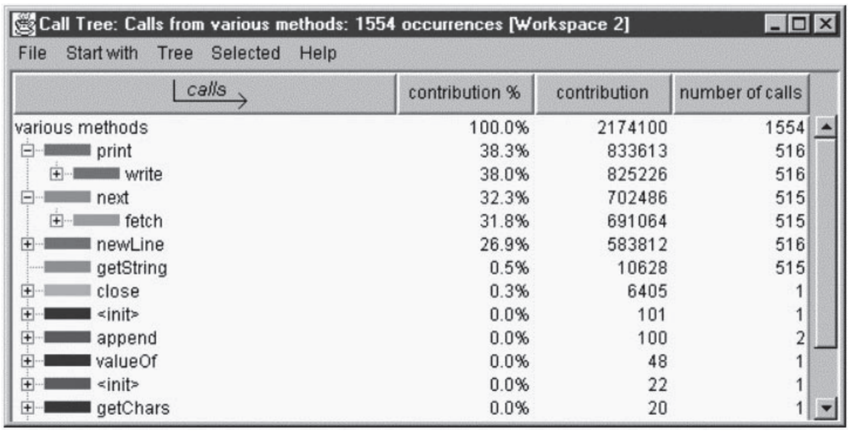


Figure 6.2. Software city metaphor based 3D visualization of JPetStore [Fittkau et al. 2013a]

6.1.3 Call Tree Views

A different, more familiar visualization form for analyzing and sorting data is offered by *Call Tree Views*. Jinsight is a tool for exploring a program's run-time behavior visually [De Pauw et al. 2002]. A user can explore program execution traces through one or more views. One of these views is a *Call Tree View*. Figure 6.3 illustrates this kind of visualization. The view shows an aggregated call hierarchy and enables to filter for items of interest, e.g., the number of calls of a specific operation or the name of an operation. Additionally, the spent time for an operation, regarding the total response time of the program execution, is presented. This allows to support the process of a performance analysis. The monitoring tool *InspectIT*, which we describe later in Chapter 12, also utilizes this way of representation for visualizing database calls. Furthermore, Kieker Trace Diagnosis already employs a similar way of representation for visualizing monitored business operations. For this reason, we choose this visualization form and adapt it for our approach, visualizing database operations.


Call Tree: Calls from various methods: 1554 occurrences [Workspace 2]
File
Start with
Tree
Selected
Help

calls	contribution %	contribution	number of calls
various methods	100.0%	2174100	1554
⊟ print	38.3%	833613	516
⊞ write	38.0%	825226	516
⊟ next	32.3%	702486	515
⊞ fetch	31.8%	691064	515
⊞ newLine	26.9%	583812	516
getString	0.5%	10628	515
⊞ close	0.3%	6405	1
⊞ <init>	0.0%	101	1
⊞ append	0.0%	100	2
⊞ valueOf	0.0%	48	1
⊞ <init>	0.0%	22	1
⊞ getChars	0.0%	20	1

Figure 6.3. Call Tree View featured in Jinsight [De Pauw et al. 2002]

6.2 Our Visualization Approach

In the previous section, we described different visualization concepts and decided to employ the latter representation, namely the *Call Tree Views* [De Pauw et al. 2002], for our approach. This form of visualization allows to represent the hierarchy of a database operation call and offers an architecture, which is suitable for functions like filtering and sorting recorded monitoring data. Our visualization should offer, at least, the following three different representation views.

▻ *Statements* – including the executed SQL statement and the response time.

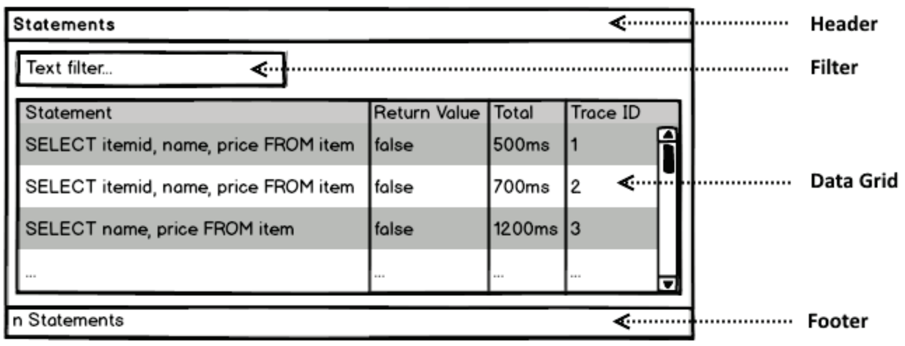
▻ *Aggregated Statements* – same as Statements, but aggregated based on their executed SQL statement.

▻ *Prepared Statements* – providing the abstract prepared statement and the concrete execution with bound parameters.

Essential visualization features for manipulating these options, e.g., sorting and filtering the monitoring data, also need to be covered. In order to give an impression of our concrete approach, we present in the upcoming three subsections different mock-ups for the already mentioned views. These mock-ups have been created with the web demo from balsamiq.[1]

6.2.1 Statements

Statements are a central database operation within our software solution. Therefore, our first mock-up, which is presented in Figure 6.4, covers this operation. Our mock-up represents a view, which is based on the concept of the *Call Tree Views*.



Statements

Text filter...

Statement	Return Value	Total	Trace ID
SELECT itemid, name, price FROM item	false	500ms	1
SELECT itemid, name, price FROM item	false	700ms	2
SELECT name, price FROM item	false	1200ms	3
...	...	...	...

n Statements

Figure 6.4. Mock-up View: Statements

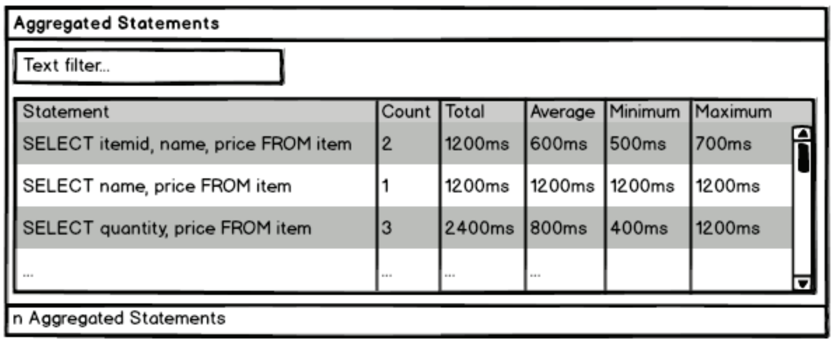
Our view is divided into four different components, namely the *Header*, the *Filter*, the *Data Grid*, and the *Footer*. The *Header* indicates the kind of database operation, which is *Statements* in our case. Below, we see an input box for filtering the presented database calls, the *Filter*. A user can type in an expression and the visualization will be filtered for matching database calls. The next component is the *Data Grid*. This visualization component is the most important one within our view. It shows our related statement calls, including the executed SQL statement string, the returned value, the response time, displayed in a changeable time unit, and the trace id. The latter one serves as an unique

[1] http://webdemo.balsamiq.com

identification, similar to a primary key in relational databases, and can be used for further analysis tasks. Since the size of the *Data Grid* is limited, a scrollbar allows the user to navigate through the statement items within the list. This keeps the layout structured and neat, even if the amount of related database calls increases. The fourth and last component is the *Footer*, which shows the number of Statements within the view, as an assistance for the user.

6.2.2 Aggregated Statements

In the previous section, we described our view for *Statements*. In this section, we present a mock-up, which shows statements, which have been aggregated based on their executed SQL statement. The mock-up is shown in Figure 6.5.



Aggregated Statements

Text filter...

Statement	Count	Total	Average	Minimum	Maximum
SELECT itemid, name, price FROM item	2	1200ms	600ms	500ms	700ms
SELECT name, price FROM item	1	1200ms	1200ms	1200ms	1200ms
SELECT quantity, price FROM item	3	2400ms	800ms	400ms	1200ms
...	...	...	...		

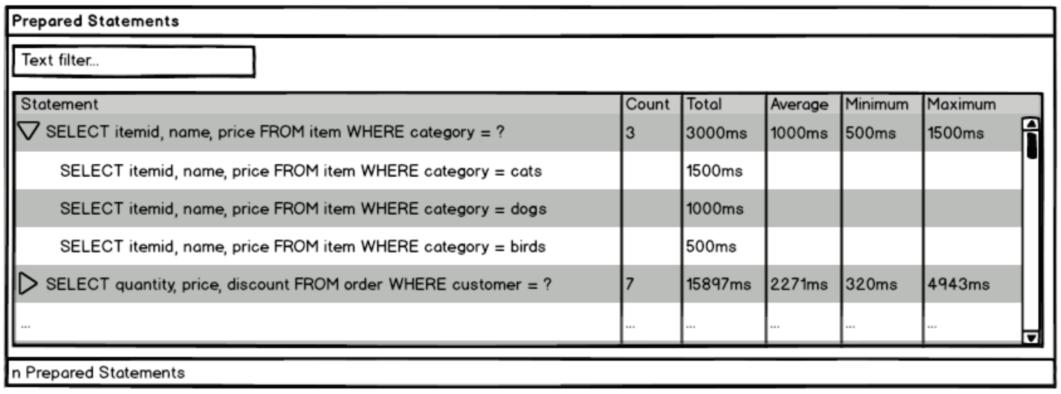
n Aggregated Statements

Figure 6.5. Mock-up View: Aggregated Statements

The layout is basically the same as in the previous mock-up. The *Header* and the *Footer* are customized for this view, but offer the same functionality. In comparison to the first view, some columns have been added. For each aggregated statement the SQL statement, the *count*, which represents the number of executions, and several response time columns are displayed. The time-related columns are the following – the total, the average, the minimum, and the maximum response time. Strong distinctions, between minimum and maximum response times, may indicate a performance problem, caused by the related aggregated statement. For this reason, these timings are important, regarding conducting a performance analysis. We removed the *Trace ID* as we are interested in the aggregated statement and not in a single statement within in this view.

6.2.3 Prepared Statements

The third and last view covers our monitored *Prepared Statements*. The related mock-up is illustrated in Figure 6.6. This view shows precompiled statements with variables, which are bound to later set call parameters, before their execution. This technique is often employed to increase the performance for frequently executed database operations, which are based on the same SQL statement, but differ in their concrete parameter values. This brings us to a slightly different layout, as we have an abstract statement, a SQL statement with parameters, and a list of concrete statements, which represent executions of the abstract one with bond values. The layout is not flat anymore, like in the previous two views. Each abstract-concrete statement hierarchy defines an own call tree. The abstract statement embodies the parent element with the tree and the concrete statements are children nodes. As we offer the functionality to analyze a concrete prepared statement with its executed parameters and the related response time, we present the real executed SQL statement, more precisely we integrate the bound values (SQL parameters) into the SQL statement string. The time-related columns are the same, as for the *Aggregated Statement* view.


Prepared Statements

Text filter...

Statement	Count	Total	Average	Minimum	Maximum
▽ SELECT itemid, name, price FROM item WHERE category = ?	3	3000ms	1000ms	500ms	1500ms
SELECT itemid, name, price FROM item WHERE category = cats		1500ms			
SELECT itemid, name, price FROM item WHERE category = dogs		1000ms			
SELECT itemid, name, price FROM item WHERE category = birds		500ms			
▷ SELECT quantity, price, discount FROM order WHERE customer = ?	7	15897ms	2271ms	320ms	4943ms
...	...	...	...	...	...

n Prepared Statements

Figure 6.6. Mock-up View: Prepared Statements

6.3 Integration into Kieker Trace Diagnosis

As mentioned before, we want to build our visualization component upon Kieker Trace Diagnosis because it already offers an appropriate ready-to-use architecture, which is extendable for our purpose. Furthermore, the tool already provides a visualization of recorded program traces in form of tree views, which are similar to the previously named *Call Tree Views*. On this basis, we can follow our visualization approach and integrate our developed views, which we presented through three different mock-ups, into Kieker Trace Diagnosis.

Chapter 7

Overview of our Implementation

This chapter covers the implementation of our enriched approach, which was explained in the previous chapter. Our implementation has the main goal to support a developer in conducting a performance analysis, regarding database operations. Our monitoring component is designed to be as light-weight and generic as possible, in order to allow multiple analysis and visualization tools to use our recorded monitoring data as input for their processing. Our analysis utilizes a Pipe-and-Filter Framework for an efficient processing of our monitoring records. Furthermore, we focus on displaying only the monitored information, relevant for the developer within our visualization.

This chapter is organized as follows. First we provide an overview of our implementation by presenting the developed architecture of our software. Afterwards, we reveal our designed software system in detail, and present the monitoring component, continuing with the analysis component, and close with the visualization component.

7.1 Architecture

Our implementation is based on our enriched approach, in order to met our previously defined requirements of our software solution. As mentioned before, we divide our software into into three parts. A monitoring part, an analysis part, and a visualization part. We enrich these parts with concrete technologies and explain the resulting architecture for our implementation. The architecture of our software solution is shown in Figure 7.1.

7.1.1 Monitoring Component

The figure is based on components, described in Chapter 3. The first difference consists of a distinction, between our *Monitoring* and *Analysis & Visualization* fragmentation. The system boundary of these fragments is illustrated as dashed lines. These two partitions can operate independently on different nodes or machines, as they are two separate applications. The *Monitoring* software contains our *Monitoring component* and performs the instrumentation of a *Monitored Application* through AspectJ. As the observed application employs the JDBC API for database communication, we intercept these operations to monitor related database calls. For each monitored database operation, there are two monitoring records created, one

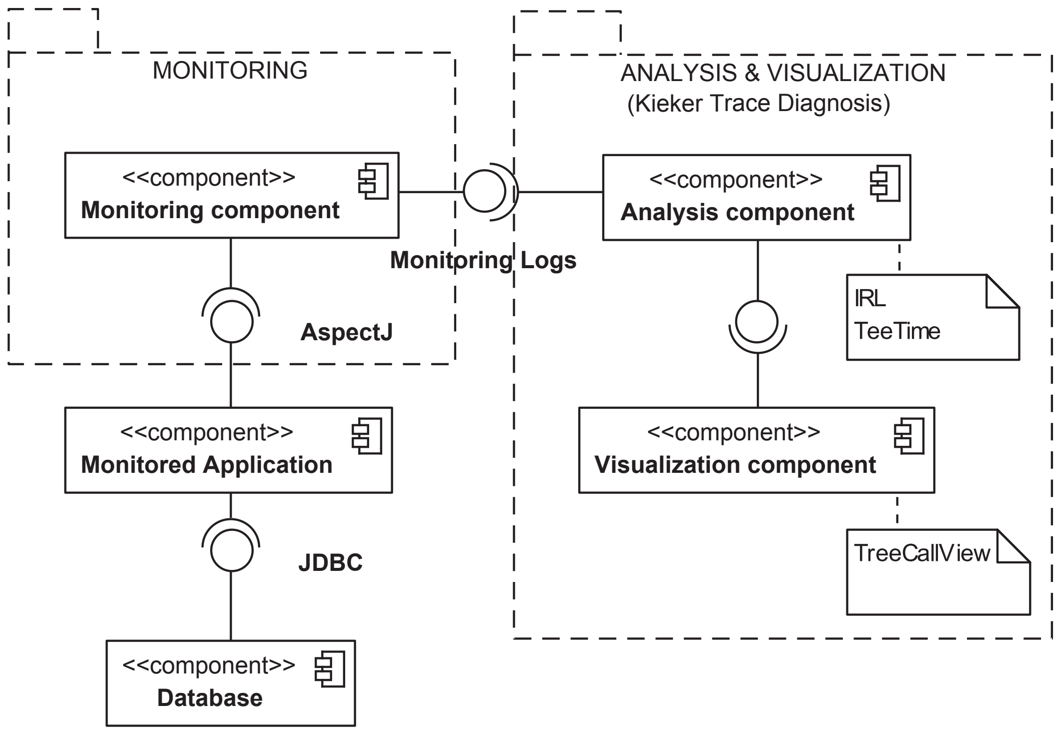


Figure 7.1. Architecture of our implementation

when the operation is started, and one, when the operation is completed. Afterwards, the previously created monitoring records are written to the file system in form of a monitoring log. The *Monitoring* component needs to be located within the execution container or environment of the application, as our instrumentation is woven into the program code by a Java agent.

7.1.2 Analysis Component

Once the *Monitoring* component has created a monitoring log file, it can be further processed within the *Analysis & Visualization* software, our second software part within our whole software system. As these parts have been integrated into Kieker Trace Diagnosis, we further refer to Kieker Trace Diagnosis, when we describe this second software fragment. The relevant components within Kieker Trace Diagnosis are the *Analysis component* and the *Visualization component*. The first one follows our enriched approach for analyzing database operations. Our *Analysis component* loads the previously created monitoring records from a monitoring log file and processes them for our *Visualization component*. As Kieker Trace

Diagnosis uses Kieker records for the analysis process of business operations, we want to integrate our database operation records into this existing structure. Therefore, we employ the IRL and generate Kieker database monitoring records, based on our requirements, and extended an existing record type. The analysis utilizes the TeeTime framework, an implementation of the Pipes & Filter Pattern. Therefore, we define different consecutive analysis tasks within the framework.

7.1.3 Visualization Component

The second component within our software fragment is the *Visualization component*. We apply the already existing Model-View-Controller (MVC) architecture within Kieker Trace Diagnosis for our visualization purposes. Generally speaking, we offer three different views, based on the MVC architecture, for our three observed database operations – statements, aggregated statements, and prepared statements. We employ *Tree Call Views* as a visualization form and visualize our monitored and processed information for the user. Furthermore, we provide capabilities to filter and sort the recorded data, in order to support the user in conducting a performance analysis on database operations.

Chapter 8

Monitoring Implementation

The general workflow of our developed monitoring component was explained in Section 4.2.2. In this section, we present our related implementation, which is based on our previous developed monitoring approach.

Figure 8.1 shows a class diagram of our implementation. It is structured into three packages, namely *monitoring*, *record*, and *utils*. Additionally, we have two configuration files: *aop.xml* and *monitoring-configuration.xml*, which are used to configure the instrumentation process dynamically, without the need to change the source code. The latter configuration file is loaded by the class *XMLConfigurationloader*. The *monitoring* package includes all necessary classes for the instrumentation through AspectJ. These are the classes *AspectConfiguration* and *MonitoringController*, which handle the instrumentation and related settings. In order to generate monitoring records of the instrumented software, we utilize our defined *MonitoringAbstractEventRecord*, more precisely its concrete derivatives *MonitoringBeforeEventRecord* and *MonitoringBeforeAfterRecord*. As we want to write our monitoring records to the file system in form of a monitoring log file, we use the class *FileLogWriter*, within in the *utils* package, for this task.

Our implementation of the monitoring component is structured into three parts, i.e., the instrumentation of an observed software system through AspectJ, the creating and processing of monitoring records, and writing these monitoring records into a monitoring log to the file system. These three tasks are described in the following.

8.1 Instrumentation Utilizing AspectJ

Our instrumentation through AspectJ and general configuration of our monitoring component is presented in Figure 8.2, which is an excerpt of the previously illustrated class diagram. We employ our two previously mentioned configuration files within our monitoring component to configure our instrumentation. The configuration file *monitoring-configuration.xml* allows to set general monitoring parameters, e.g., enabling or disabling the monitoring. The *MonitoringController* is a central class within our component, as it controls our monitoring. There are several options, which can be configured due the *monitoring-configuration.xml* document, which is shown in Listing 8.1. This file is loaded

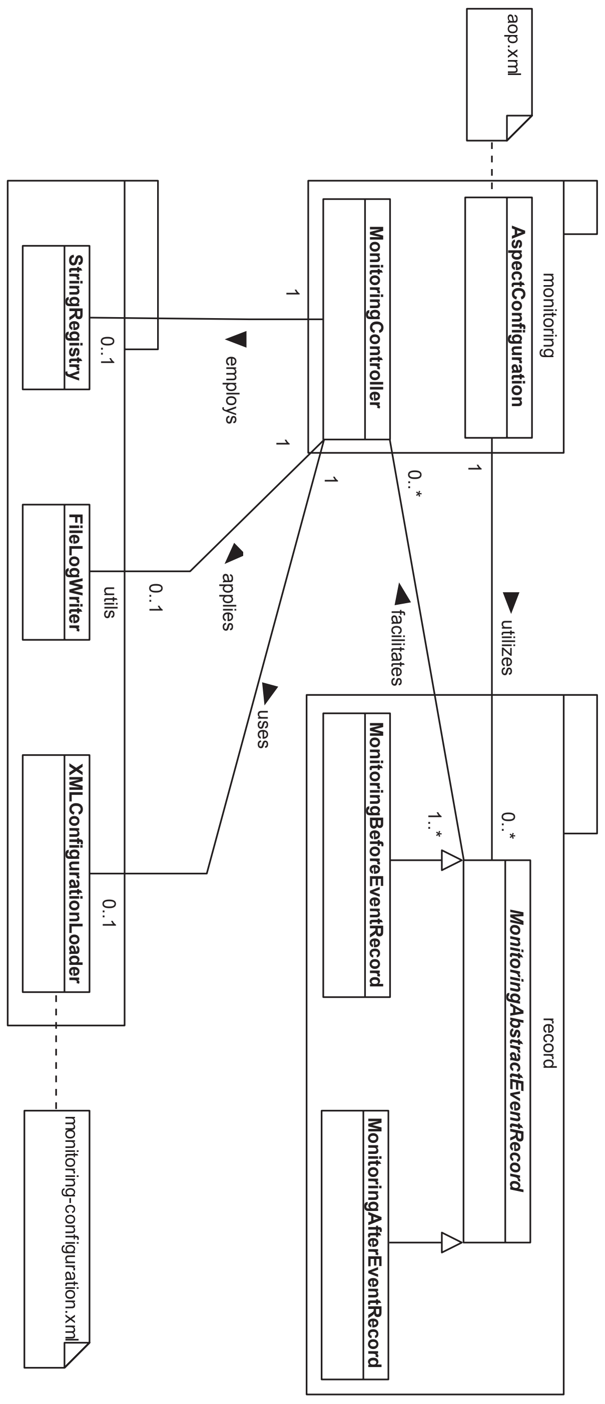


Figure 8.1. Main classes of our monitoring component

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- configuration for the monitoring controller -->
<monitoring-configuration>
  <!-- permitted values: true, false -->
  <monitoring-enabled value="true" />
  <debugging-enabled value="true" />
  <filewriting-enabled value="false" />
  <format-kieker-enabled value="true"/>
  <!-- permitted values: .dat, .log -->
  <logfile-extension value=".dat" />
</monitoring-configuration>
```

Listing 8.1. monitoring-configuration.xml: instrumentation settings configuration

and parsed through the *XMLConfigurationLoader*, which sets the values of the controller afterwards.

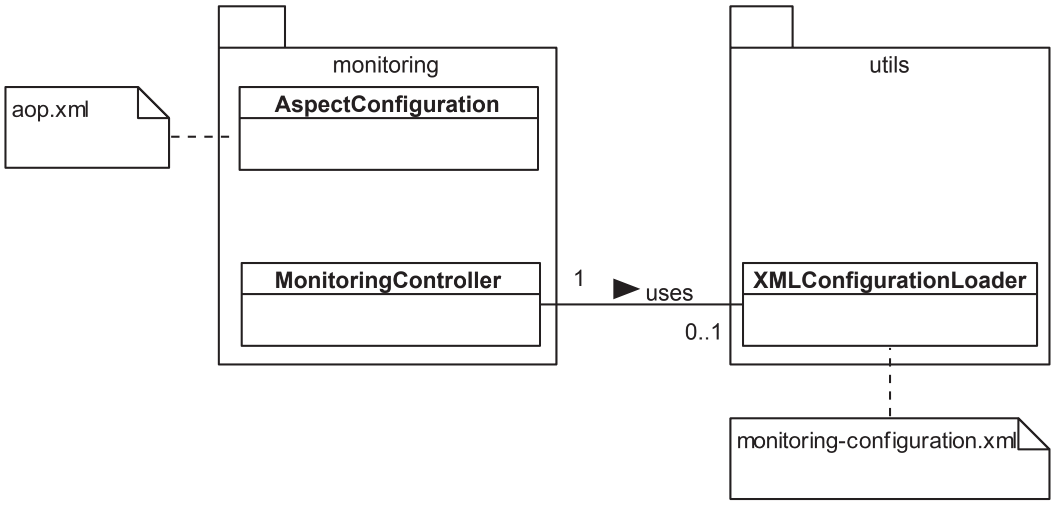


Figure 8.2. Excerpt of related classes for the instrumentation of our monitoring component

The monitoring configuration allows to toggle the monitoring on or off (true or false), also the debug mode. Furthermore, it enables or disables writing the monitoring log file to the file system, activates the conversion to the Kieker monitoring record format, and defines the extension of the monitoring log file. If one or more configuration settings are invalid, then the *MonitoringController* uses default values.

The class *AspectConfiguration* includes all definitions, which are necessary for an instrumentation through AspectJ. As we want to observe JDBC operations, we define them as *pointcuts*, related to our remarks in Section 4.1. Each time one of these operations is passed by, during an execution of the observed software system, we employ an *around advice* and create a corresponding monitoring record. Additionally, we have to distinguish between database operation calls, which are created when an operation has started on the one side, and when it is completed on the other side. This is a necessary step, as we are interested in the returned value. Furthermore, we enrich this record with information, e.g., the executed class name, the concrete operation, its arguments, and the returned value.

```xml
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "http://www.aspectj.org/dtd/aspectj_1_5_0.dtd
    ">
<aspectj>
    <weaver options="-Xset:weaveJavaPackages=true,weaveJavaxPackages=true">
      <include within="*" />
    </weaver>
    <aspects>
      <aspect name="dabamo.monitoring.AspectConfiguration" />
    </aspects>
</aspectj>
```

Listing 8.2. aop.xml: Configurating the monitoring scope

The instrumentation of our monitoring is presented in Listing 8.2. The file defines the monitoring scope for our instrumentation and is divided into two parts, *weaver* and *aspects*. The first one allows to set options for the weaving process. As we are interested in JDBC calls, we need to include the option *Java Packages*. Furthermore, we define the monitoring focus with the include tag. At the moment, we still monitor all classes, as a limitation, to classes of a concrete observed Java software system, did not work properly. In the future, we plan to fix this circumstance to increase the monitoring performance. Our setting under the tag *Aspects* references our *AspectConfiguration*, which is employed, when AspectJ is loaded by the JVM.

8.2 Monitoring Record Processing

Once the monitoring records are created, they need to be processed further. Related classes are presented in Figure 8.3. In order to process the created monitoring records, we utilize our defined monitoring record class *MonitoringAbstractEventRecord*, more precisely its concrete derivatives *MonitoringBeforeEventRecord* and *MonitoringAfterEventRecord*. Furthermore, the *MonitoringController* employs a *StringRegistry*, which operates like a mapper for strings.

Our monitoring record class *MonitoringAbstractEventRecord* offers all necessary attributes to enable a further processing of our created records. An overview of the class is given in

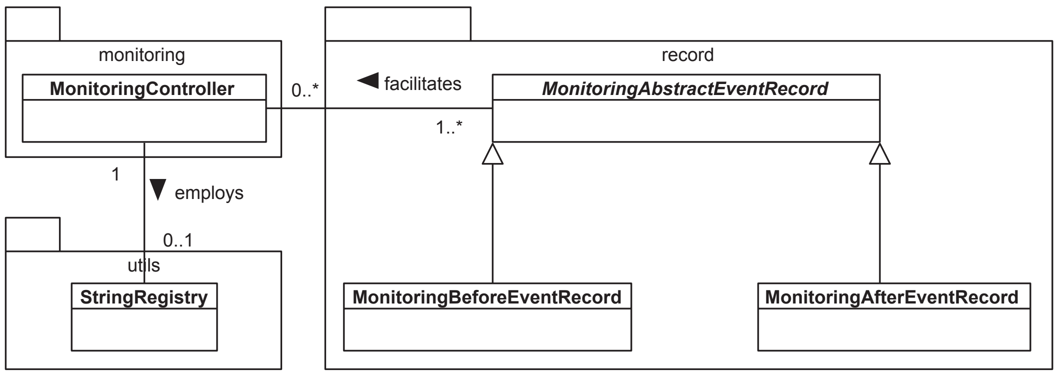


Figure 8.3. Excerpt of related classes for record processing in our monitoring component

Figure 8.4. Our previously mentioned monitoring record attributes, which we described in Section 4.2.2, are listed in the following:

▷ cid: an unique identifier for mapping before and after event records

▷ classNameId : mapped id, which represents the full Java class operation name

▷ classArgsId: mapped id, which represents the operation argument, e.g., a SQL statement

▷ timestamp: represents date and time of the record

▷ formattedReturnValue: returned value of an executed operation, formatted based on the returned data type of the operation

The type of the record, more precisely the distinction between a before or after event record, is handled through its derivate classes *MonitoringBeforeEventRecord* and *MonitoringAfterEventRecord*. Additionally, the type of the return value, e.g., ResultSet, boolean or int, is parsed from the full class name and used for processing the *formattedReturnValue*. The attribute *StringRegistry* references the self-named instance within the *MonitoringController*. Basically, it applies a mapping between strings and an id. We employ this technique to improve our monitoring speed, as multiple class operation names or arguments are just added once to the registry. Hence, the second and all further accesses only require a reading operation.

8.3 File Writing

The last task within our monitoring component is to write the monitoring records into a monitoring log file for the upcoming analysis. This task is handled by the class *FileLogWriter*. If the writing of a monitoring log file is enabled within the configuration of the

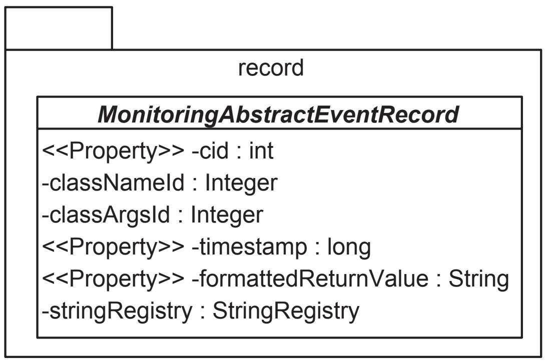

record
MonitoringAbstractEventRecord
<<Property>> -cid : int
-classNameId : Integer
-classArgsId : Integer
<<Property>> -timestamp : long
<<Property>> -formattedReturnValue : String
-stringRegistry : StringRegistry

Figure 8.4. MonitoringAbstractEventRecord attributes

```
$0;592199450609250;Statement java.sql.Connection.createStatement();java.sql.Connection.
    createStatement();null;null;0;0;592199450609250;null;null
$1;592199458627705;Statement java.sql.Connection.createStatement();java.sql.Connection.
    createStatement();null;null;0;0;592199458627705;null;null
$0;592199490311527;boolean java.sql.Statement.execute(String);java.sql.Statement.execute(
    String);null;null;0;0;592199490311527;null;create table signon ( username varchar(25)
    not null, password varchar(25) not null, constraint pk_signon primary key (username) )
$1;592199497972053;boolean java.sql.Statement.execute(String);java.sql.Statement.execute(
    String);null;null;0;0;592199497972053;false;create table signon ( username varchar(25)
    not null, password varchar(25) not null, constraint pk_signon primary key (username) )
$0;592228000197825;boolean java.sql.Statement.execute(String);java.sql.Statement.execute(
    String);null;null;0;0;592228000197825;null;SELECT itemid, productid, listprice FROM
    item
$1;592228000297821;boolean java.sql.Statement.execute(String);java.sql.Statement.execute(
    String);null;null;0;0;592228000297821;false;SELECT itemid, productid, listprice FROM
    item
```

Listing 8.3. Excerpt of a monitoring log file using the Kieker format

MonitoringController, the monitoring records are sequentially written to the file system. Depending on whether a conversion to the Kieker monitoring log format is requested or not, a corresponding monitoring log file is created. Also the extension is configurable through our monitoring configuration. As we need to employ the Kieker monitoring log format for our analysis and visualization tasks within Kieker Trace Diagnosis, an excerpt from an example log file is shown in Listing 8.3. Each line represents a monitoring record and the inherited logging attributes are separated by a semicolon. The leading value indicates the type of monitoring record ($0: *MonitoringBeforeEventRecord* and $1: *MonitoringAfterEventRecord*).

As Kieker needs to know, which exact record type is represented in the log file for the analysis, a mapping file, namely *kieker.map*, needs to be specified. Previously, we applied the IRL to generate compatible Kieker records. These record classes are defined in this file for a correct mapping, which is presented in Listing 8.4. Our generated monitoring record classes are located within the *kieker.common.record.io.database* package.

```
$0=kieker.common.record.io.database.DatabaseBeforeEventRecord
$1=kieker.common.record.io.database.DatabaseAfterEventRecord
```

Listing 8.4. Kieker record mapping definition

Chapter 9

Analysis Implementation

The general workflow of our developed analysis component was explained in Section 4.2.2. In this chapter, we describe our related implementation and name the relevant components, which are presented in Figure 9.1.

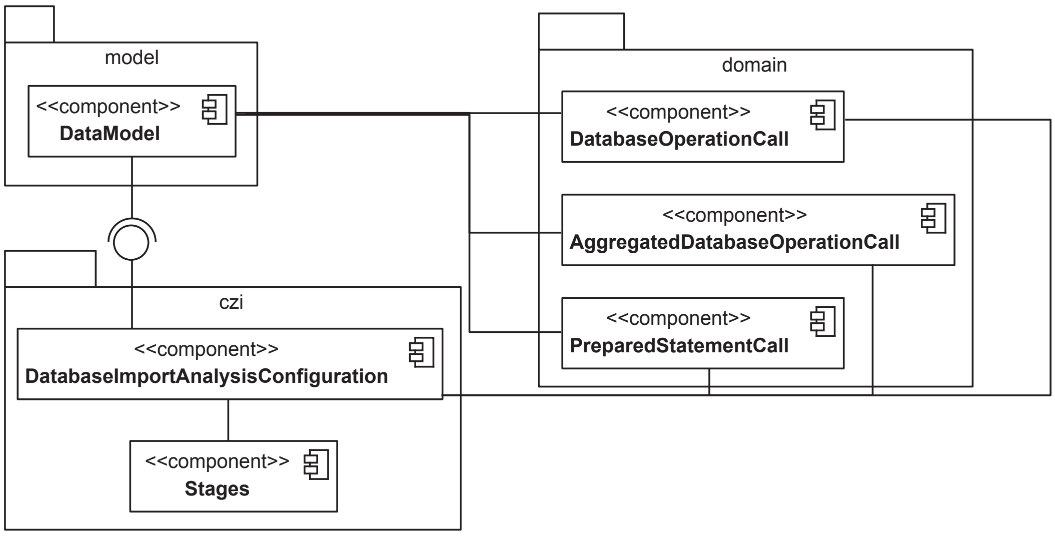


Figure 9.1. Main components of our analysis component

Our implementation of the analysis component is structured into three packages, namely *model*, *domain*, and *czi*. The first one consists of the class *DataModel*, which acts as a data storage for Kieker Trace Diagnosis. The package *domain* contains mapping classes for our previously generated monitoring records. *DatabaseOperationCall* serves as a basis object for database operation calls, *AggregatedDatabaseOperationCall* for aggregated database operation calls, and *PreparedStatementCall* for the same-named calls. The analysis, the related configuration, and the employed filter (stages), are comprised within the classes *DatabaseImportAnalysisConfiguration*, respectively *Stages*, which is an abstract component that includes multiple filter. The presented classes within these three packages are described in the following.

9.1 Monitoring Record Classes

In order to map our previously generated monitoring records based on the monitoring component, we need to define Java objects within Kieker Trace Diagnosis. This allows to process and utilize the monitoring records for our analysis and the visualization. Our monitoring record classes extend the existing class *AbstractOperationCall*, as it already provides useful attributes, e.g., component, operation, and the option to define children objects. The latter one allows to construct (call) trees extending *AbstractOperationCall*. The class is an abstract base for classes, which represent operation calls (also called executions) within Kieker Trace Diagnosis. We describe our three newly introduced monitoring classes in the following.

9.1.1 DatabaseOperationCall

The class *DatabaseOperationCall* is our primary class for storing and processing monitored database operation calls. A reduced class diagram is presented in Figure 9.2. We employ the class *DatabaseOperationCall* within Kieker Trace Diagnosis as a data structure for statements.

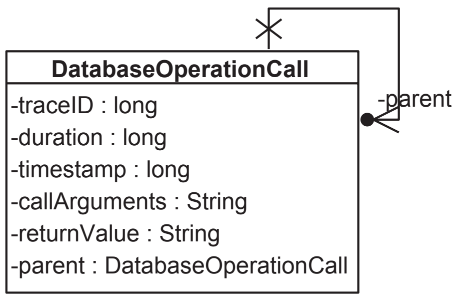


Figure 9.2. Reduced class diagram of DatabaseOperationCall

The presented attributes are based on the record structure, we defined within our monitoring component, as seen in Figure 8.4. The class consists of the following attributes:

▻ traceid: an unique identifier for statements

▻ duration: the duration of the executed statement

▻ timestamp: represents date and time of the record

▻ callArguments: includes the executed SQL statement

▻ returnValue: returned value of an executed operation, formatted based on the data type

▷ parent: contains a reference to a parent object, if one exists

9.1.2 AggregatedDatabaseOperationCall

The class *AggregatedDatabaseOperationCall* is applied as a data structure for aggregated Statements based on their executed SQL statement. We present a reduced class diagram in Figure 9.3.

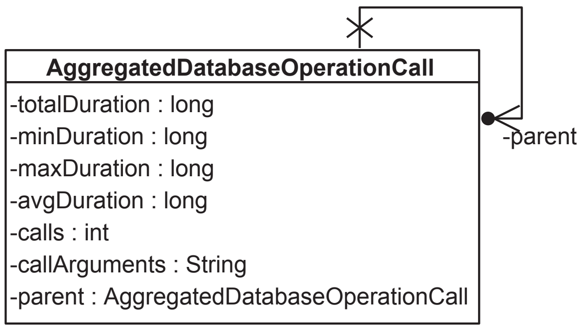


Figure 9.3. Reduced class diagram of AggregatedDatabaseOperationCall

The presented attributes are similar to our previous defined monitoring record structure. The class consists of the following attributes.:

▷ totalDuration: the total used response time, regarding a specific aggregated statement

▷ minDuration: the minimum duration of the executed statement

▷ maxDuration: the maximum duration of the executed statement

▷ avgDuration: the average duration of the executed statement

▷ calls: the number of calls or children of a specific aggregated statement

▷ callArguments: includes the executed SQL statement

▷ parent: contains a reference to a parent object, if one exists

For each SQL statement a parent call is created. If a parent call for a statement call already exists, then the statement call is added as a child to the parent call. As we are interested in conducting a performance analysis of database operations, the response time is very important. In order to find potential performance problems, more precisely concrete statements, we calculate different timings for an aggregated statement. Therefore we note the total, the average, the minimum, and the maximum response time. Additionally, we employ an attribute *calls*, which stores the count of related children calls.

9.1.3 PreparedStatementCall

The last of the three mapping classes is *PreparedStatementCall*. We employ this data structure for our same-named prepared statements. A reduced class diagram is shown in Figure 9.4.

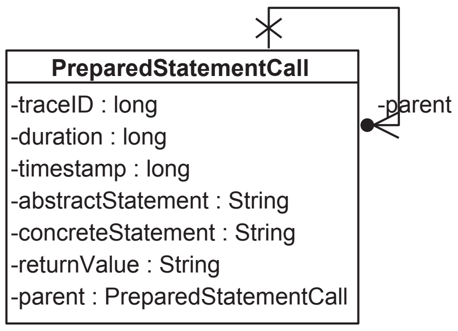


Figure 9.4. Reduced class diagram of PreparedStatementCall

The presented class *PreparedStatementCall* is a variation of the *Statement* class. We list the relevant attributes, which are similar to our previous defined monitoring record structure. The monitoring data structure consists of the following attributes:

- traceID: an unique identifier for a prepared statement
- duration: the duration of the execution
- timestamp: represents date and time of the record
- abstractStatement: includes the precompiled SQL statement, only if the object is a parent object
- concreteStatement: shows the concrete executed SQL statement, only if the object is not a parent object
- returnValue: returned value of an executed operation, formatted based on the data type
- parent: contains a reference to a parent object, if one exists

Each abstract statement defines its own call tree hierarchy. The abstract statement object embodies the parent element and the concrete statements are referenced children nodes. As each concrete statement contains a response time (duration), it is possible to distinguish, which bound parameters may involve a performance problem.

9.2 DataModel

The class *DataModel* is a central container for data, which is used within the application. It manages all necessary information for the visualization and offers data structures for the analysis. A reduced class diagram of the class is displayed in Figure 9.5.

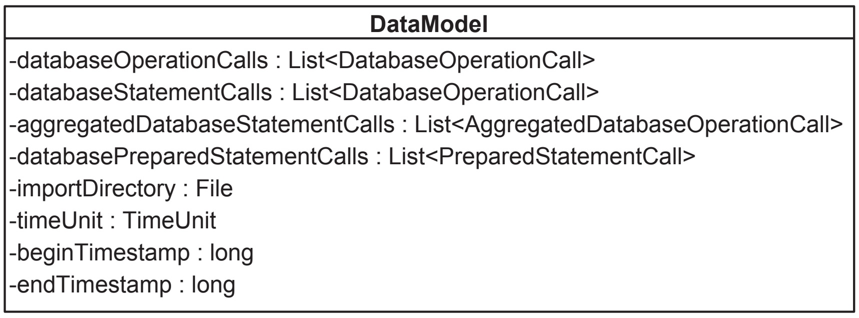

DataModel
-databaseOperationCalls : List<DatabaseOperationCall>
-databaseStatementCalls : List<DatabaseOperationCall>
-aggregatedDatabaseStatementCalls : List<AggregatedDatabaseOperationCall>
-databasePreparedStatementCalls : List<PreparedStatementCall>
-importDirectory : File
-timeUnit : TimeUnit
-beginTimestamp : long
-endTimestamp : long

Figure 9.5. Reduced class diagram of DataModel

The *DataModel* contains four lists for storing the results from the analysis for the database calls. It also employs the analysis and sets the location of the monitoring log, if one is loaded, into the attribute *importDirectory*. Furthermore, it specifies the applied time unit, e.g., nanoseconds, and stores the starting and ending time of an executed analysis (*beginTimestamp* and *endTimestamp*).

9.3 Analysis Configuration

In the previous sections we explained where database operations are stored within Kieker Trace Diagnosis, and what kind of data structure we employ. In this section, we cover the analysis based on the TeeTime[1] Pipes-And-Filter framework. TeeTime offers an abstract class, *AnalysisConfiguration*, for configuring an analysis and setting up pipes and filters. The class represents a configuration of connected stages, which is needed to run an analysis within TeeTime. As the framework utilizes the term *stage* instead of filter, we adopt this naming and employ it in the following. For our purpose, we extend this *AnalysisConfiguration* class and create our own analysis configuration, namely the class *DatabaseImportAnalysisConfiguration*, which is illustrated in Figure 9.6.

The diagram shows our employed sinks within the Pipes-And-Filter architecture. We possess a list for each specific database operation, e.g., *PreparedStatementCall*, and collect them in a similar-named sink, which acts like an endpoint within the analysis. These lists

[1]`http://teetime.sourceforge.net`

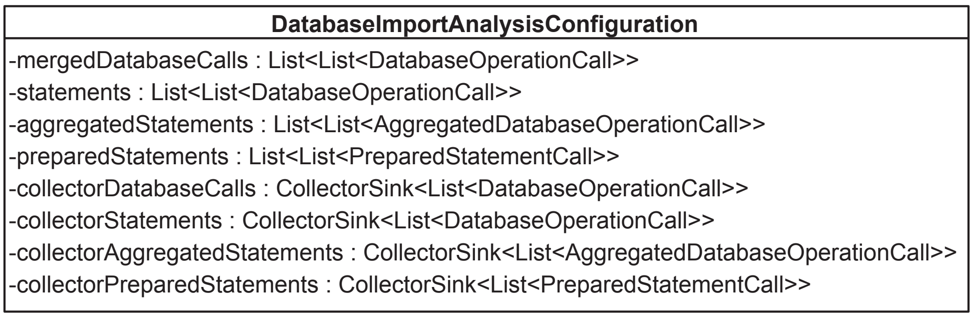

DatabaseImportAnalysisConfiguration
-mergedDatabaseCalls : List<List<DatabaseOperationCall>>
-statements : List<List<DatabaseOperationCall>>
-aggregatedStatements : List<List<AggregatedDatabaseOperationCall>>
-preparedStatements : List<List<PreparedStatementCall>>
-collectorDatabaseCalls : CollectorSink<List<DatabaseOperationCall>>
-collectorStatements : CollectorSink<List<DatabaseOperationCall>>
-collectorAggregatedStatements : CollectorSink<List<AggregatedDatabaseOperationCall>>
-collectorPreparedStatements : CollectorSink<List<PreparedStatementCall>>

Figure 9.6. Reduced class diagram of DatabaseImportAnalysisConfiguration

are an in-between data storage, until the processed data is transfered into the *DataModel*. As described in our approach, we divided the necessary into several tasks, which can be transformed into stages for our analysis. Once again, we split our processing into two domains. Therefore the subsections *generic monitoring record processing* and *specific database call handling* are described in the following.

9.3.1 Generic Monitoring Record Processing

We set-up our analysis configuration and prepare the data structures for storing the results. In order to retrieve these results, we need to further analyze the gathered data. Therefore, we developed a process, which loads our recorded monitoring data and processes it through several, subsequent tasks. In Figure 9.7, the process is illustrated, in form of a component diagram. The diagram is composed of five components, namely the *ReadingComposite*, the *MultipleInstanceOfFilter*, the *DatabaseRecordTransformator*, the *CreateListOfDatabaseCalls*, and the *DatabaseCallMerger*. Once more, we want to focus on flexibility and extendibility. Hence, our components are easy to maintain or replace. We explain the different stages in the following.

ReadingComposite

This stage is a composite stage, which loads monitoring records from a specific directory, deserializes them, and forwards them to an output port. As the stage already existed within Kieker Trace Diagnosis, we applied it for our purpose.

MultipleInstanceOfFilter

The stage filters the deserialized monitoring records and matches them against existing monitoring records, which are derived from the *IMonitoingEvent* record, a basic record type defined in Kieker. As we employed the IRL and generated our own compatible Kieker

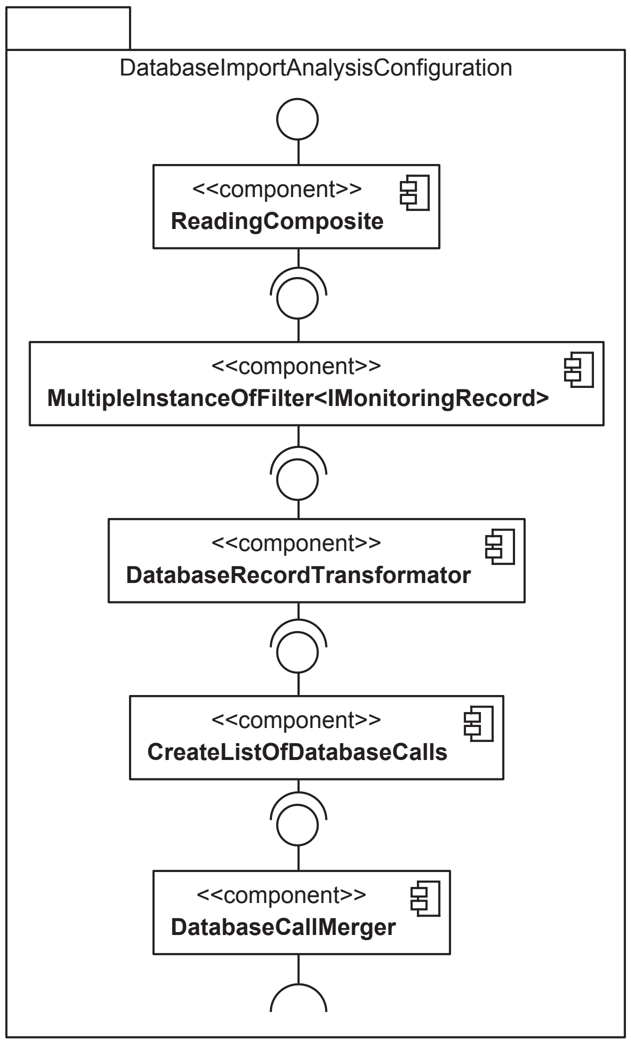

DatabaseImportAnalysisConfiguration
<<component>>
ReadingComposite
<<component>>
MultipleInstanceOfFilter<IMonitoringRecord>
<<component>>
DatabaseRecordTransformator
<<component>>
CreateListOfDatabaseCalls
<<component>>
DatabaseCallMerger

Figure 9.7. Analysis: Generic Monitoring Record Processing

monitoring records, the filter recognizes our records and matches them for the ongoing analysis. We applied this stage for our purpose, as it already existed within Kieker Trace Diagnosis.

DatabaseRecordTransformator

Within this stage, we filter our monitoring records for our record types *DatabaseBeforeEventRecord* and *DatabaseAfterEventRecord*. We distinguish between them, as they offer different information, e.g., the timestamp for calculating the duration (response time) of an executed database operation and the returned value. All matched monitoring records are afterwards transformed into *DatabaseOperationCalls*.

CreateListOfDatabaseCalls

This stage covers only a technical transformation. The processed database operation calls are added to a list. This is a necessary step, as our ongoing stages operate on lists, instead of single monitoring records.

DatabaseCallMerger

The *DatabaseCallMerger* takes the list of processed *DatabaseCalls* and merges each two elements of the list sequentially (before and after events database calls) into one single database call. This is a requirement for our next process, the *Specific Database Call Handling*.

9.3.2 Specific Database Call Handling

Once we handled our records through the *Generic Monitoring Record Processing*, we apply additional filter, in order to extract related database operation call objects. Hence, we focus on identifying specific database operations, e.g., *Statements* and *Prepared Statements*, which we described in Section 4.1. As we continue our analysis using the Pipes-And-Filter framework TeeTime, we present the involved components and stages for the specific database call handling in Figure 9.8.

For our further analysis, we employ four different types of stages, namely *Distributors*, *Extractors*, *Aggregators*, and *Sinks*. Our analysis is divided into three branches, which are used to extract *Statements*, combine these based on their executed SQL statements to *Aggregated Statements*, and process *Prepared Statements*. We describe these three branches in the following.

Statements

In order to extract related database operation calls, we start with the first stage, the *Distributor<List<DatabaseOperationCall>>*, which is located at the top left corner of our diagram. It distributes the merged database operation calls from the generic monitoring record processing to the *StatementExtractor* and the *PreparedStatementExtractor*. The latter one is applied for *Prepared Statements*. The *StatementExtractor*, extracts statements from database calls and refines them for our ongoing analysis. This includes merging several

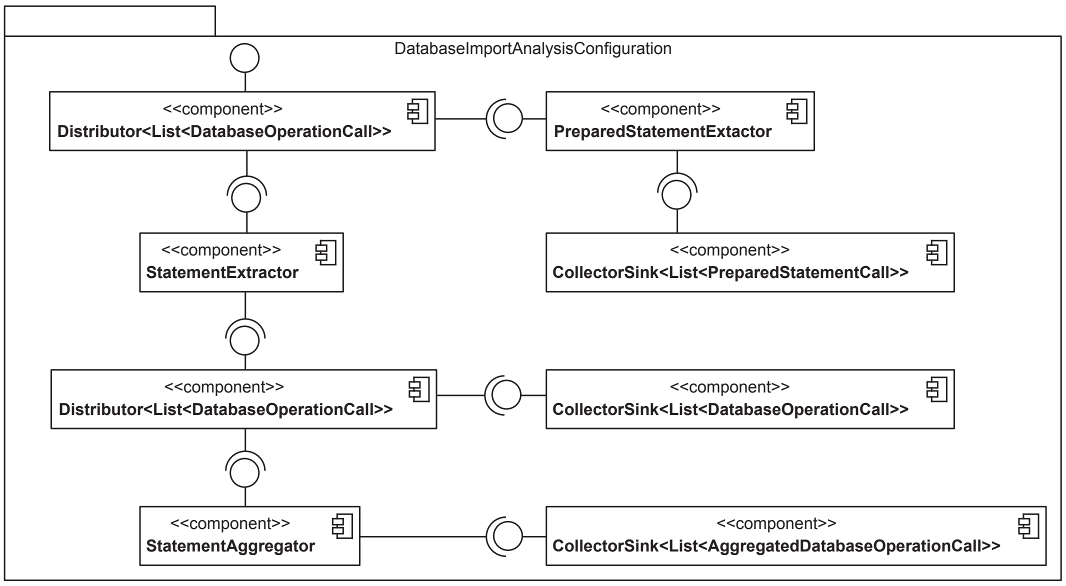

DatabaseImportAnalysisConfiguration
<<component>> Distributor<List<DatabaseOperationCall>>
<<component>> PreparedStatementExtactor
<<component>> StatementExtractor
<<component>> CollectorSink<List<PreparedStatementCall>>
<<component>> Distributor<List<DatabaseOperationCall>>
<<component>> CollectorSink<List<DatabaseOperationCall>>
<<component>> StatementAggregator
<<component>> CollectorSink<List<AggregatedDatabaseOperationCall>>

Figure 9.8. Analysis: Specific database call handling

statements calls (*createStatement()* and related executors) into one single call. Also the overall response time is calculated. Afterwards these statements are forwarded to a sink, which acts as a storage for them (*CollectorSink<List<DatabaseOperationCall>>*).

Aggregated Statements

The second distributor *Distributor<List<DatabaseOperationCall>>* forwards the extracted statements to the *StatementAggregator*. This stage aggregates statement calls based on their operation and executed SQL statement. Furthermore, related response times, e.g., the average and maximum response time, are calculated.

Prepared Statements

The *PreparedStatementExtractor* stage extracts and aggregates prepared statement calls based on our definition in Section 5.1.2. We merge prepared statement calls (*prepareStatement* and related setters and executors) into one call including children, if they exist. Additionally, the *PreparedStatementExtractor* creates an abstract prepared statement with variables, if a precompiled statement is processed for the first time. Afterwards the abstract *Prepared Statement*, which is enriched by setters, is employed to reconstruct concrete executions. These are stored within the call hierarchy of the abstract *Prepared Statement*.

Chapter 10

Visualization Implementation

In the previous section, we described, how we process monitored database operations, analyze them, and store them into the *Datamodel* class, in order to prepare the resulting data for our visualization. In Chapter 6, we decided to employ the *Call Tree View* [De Pauw et al. 2002] representation for our approach. We apply this form of visualization to represent the call hierarchy of a database operation call. Furthermore, we developed, based on the existing underlying visualization architecture within Kieker Trace Diagnosis, which applies the MVC pattern, three different representation views. Additionally, we provide filtering and sorting functions for our recorded monitoring data. In the following, we present an overview of the underlying architecture and describe our three related views.

10.1 Architecture

As mentioned before, Kieker Trace Diagnosis utilizes the MVC pattern for the visualization. We integrate our related views and develop them upon this underlying architecture. In Figure 10.1, the MVC-based visualization architecture, for visualizing database calls, is shown.

Kieker Trace Diagnosis facilitates a *Main* view, which acts as a parent view for several other views. These are the previously existing views for business operation records and our new database operation monitoring records. The MVC architecture allows us to divide our visualization component into three partitions, in order to separate the internal data structure and the processing from the presentation, towards the user of the GUI. We employ three different views, namely the *Statements*, the *Aggregated Statements*, and the *Prepared Statements* view. Each of them also applies the MVC pattern.

Figure 10.2 shows the navigation structure of our available views within the GUI. On the one hand *Business Operation* views are offered, on the other hand our new views for *Database Operations* are shown. Each view represents a sub-view, which is related to the *Main* view. The *Main* view initializes and manages the nodes within the presented navigable tree. We explain our three related database operation views in the following.

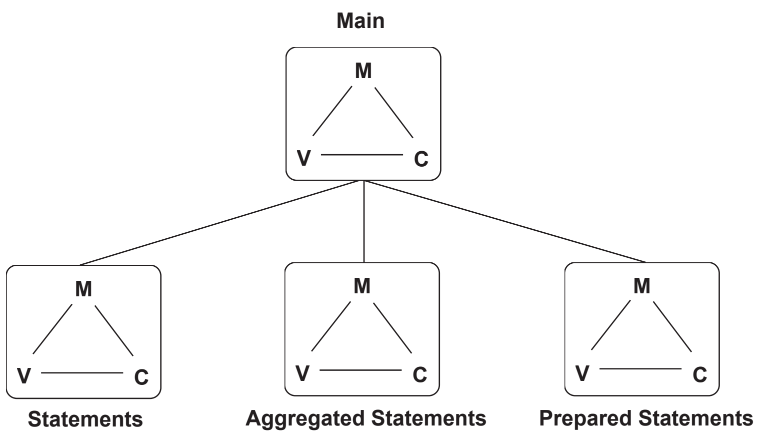


Figure 10.1. MVC architecture of our visualization implementation

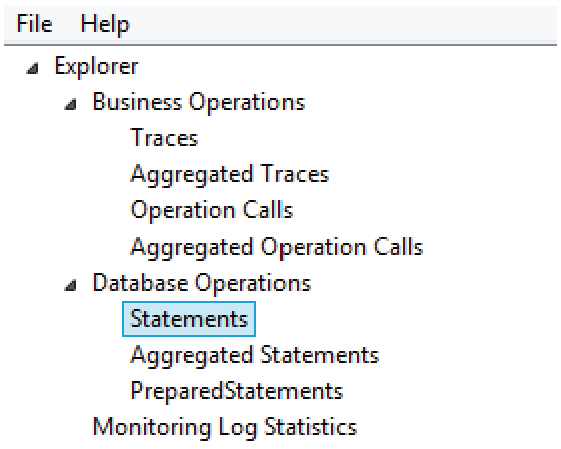
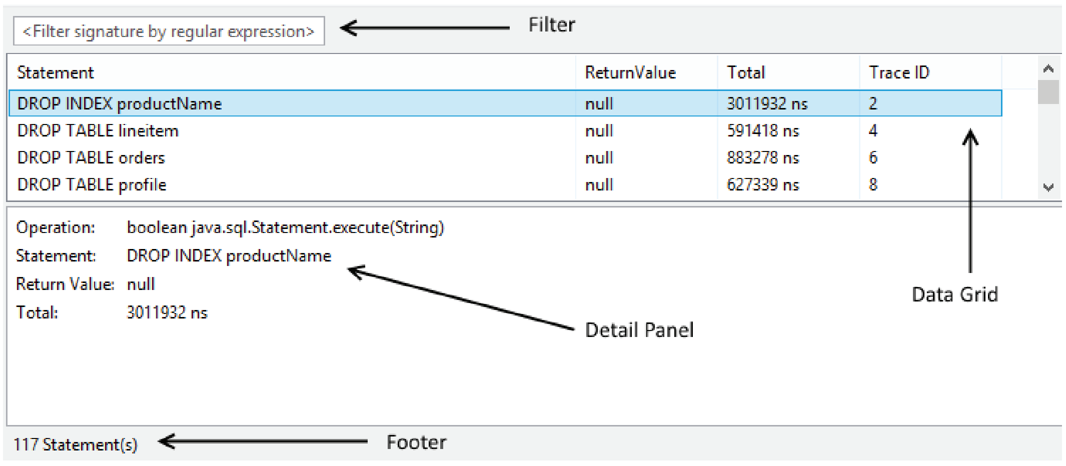


Figure 10.2. Kieker Trace Diagnosis: navigation tree view screenshot

10.2 Statements

We developed our *Statements* view based on our previously presented mock-up, which is shown in Figure 10.3. Therefore, our view is divided into four different components, namely the *Filter*, the *Data Grid*, the *Detail Panel*, and the *Footer*. The *header* is not shown, as the name of the currently selected database operation is included in the navigation tree. Additionally, we created another component, the *Detail Panel*, to enhance the visualization, as we present a second information sub-view for the user.



<Filter signature by regular expression>

Statement	ReturnValue	Total	Trace ID
DROP INDEX productName	null	3011932 ns	2
DROP TABLE lineitem	null	591418 ns	4
DROP TABLE orders	null	883278 ns	6
DROP TABLE profile	null	627339 ns	8

Operation:	boolean java.sql.Statement.execute(String)
Statement:	DROP INDEX productName
Return Value:	null
Total:	3011932 ns

117 Statement(s)

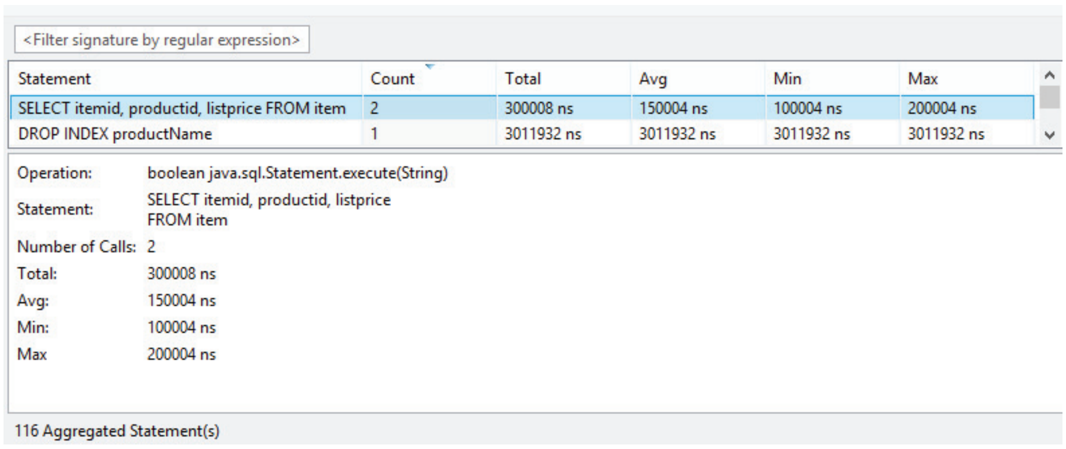
Figure 10.3. Kieker Trace Diagnosis: statement view screenshot

At the top left corner of the view, the input box for filtering the presented statements, the *Filter*, is shown. The filter operates with regular expressions for filtering the database calls. The *Data Grid* component consists of all necessary information about the related statement calls. These are the executed SQL statement string, the returned value, the response time, and the mapped trace id. There are corresponding columns within the *Data Grid*, that allow an ascending or descending sorting of the shown statements. The sorting criterion is chosen by clicking on one of the described columns. Furthermore, the user can use a scrollbar to navigate through the statement items within the list. Our component *Detail Panel* shows the available information for the currently selected statement, e.g., “DROP INDEX productName”, in the screenshot. Hence, we offer a second, more structured perspective for the user. Below the *Detail Panel* the *Footer* component is located, which shows the number of statements within the view.

10.3 Aggregated Statements

In this section, we present our *Aggregated Statements* view, which shows statements, which have been aggregated based on their executed SQL statement. The view is presented in Figure 10.4.

As the layout is similar to the *Statements* view, we do not introduce the components again and describe only the differences instead. In comparison to the first view, we added some time-related columns. More precisely, we present the total, the average, the minimum, and the maximum response time for an aggregated statement. Additionally, each aggregated statement contains the executed SQL statement and the *count*, which represents the number



<Filter signature by regular expression>

Statement	Count	Total	Avg	Min	Max
SELECT itemid, productid, listprice FROM item	2	300008 ns	150004 ns	100004 ns	200004 ns
DROP INDEX productName	1	3011932 ns	3011932 ns	3011932 ns	3011932 ns

Operation:	boolean java.sql.Statement.execute(String)
Statement:	SELECT itemid, productid, listprice FROM item
Number of Calls:	2
Total:	300008 ns
Avg:	150004 ns
Min:	100004 ns
Max	200004 ns

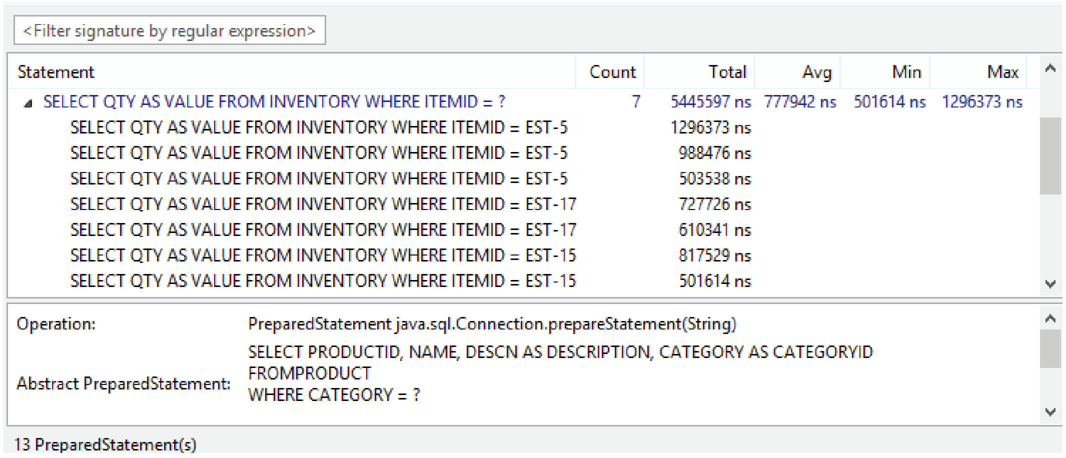
116 Aggregated Statement(s)

Figure 10.4. Kieker Trace Diagnosis: aggregated statement view screenshot

of executions. The *Detail Panel* component is extended towards the added columns and presents all necessary information to a related aggregated statement at a glance.

10.4 Prepared Statements

The third and last view covers our monitored *Prepared Statements*. Once again, we offer a similar view to the previous presented database operation views. The *Prepared Statements* view is shown in Figure 10.5.



<Filter signature by regular expression>

Statement	Count	Total	Avg	Min	Max
◢ SELECT QTY AS VALUE FROM INVENTORY WHERE ITEMID = ?	7	5445597 ns	777942 ns	501614 ns	1296373 ns
SELECT QTY AS VALUE FROM INVENTORY WHERE ITEMID = EST-5		1296373 ns			
SELECT QTY AS VALUE FROM INVENTORY WHERE ITEMID = EST-5		988476 ns			
SELECT QTY AS VALUE FROM INVENTORY WHERE ITEMID = EST-5		503538 ns			
SELECT QTY AS VALUE FROM INVENTORY WHERE ITEMID = EST-17		727726 ns			
SELECT QTY AS VALUE FROM INVENTORY WHERE ITEMID = EST-17		610341 ns			
SELECT QTY AS VALUE FROM INVENTORY WHERE ITEMID = EST-15		817529 ns			
SELECT QTY AS VALUE FROM INVENTORY WHERE ITEMID = EST-15		501614 ns			

Operation:	PreparedStatement java.sql.Connection.prepareStatement(String)
Abstract PreparedStatement:	SELECT PRODUCTID, NAME, DESCN AS DESCRIPTION, CATEGORY AS CATEGORYID FROMPRODUCT WHERE CATEGORY = ?

13 PreparedStatement(s)

Figure 10.5. Kieker Trace Diagnosis: prepared statement view screenshot

The difference to the last view is related to the distinction of an abstract prepared statement and its concrete executions. Prepared statements are precompiled statements with variables, which are bound to later set call parameters, before their execution. For the first time in our visualization, the *Data Grid* component really shows a tree visualization. The previous views also used trees, but they were flat. More precisely, we merged these calls into one single call for the visualization. For our prepared statements we have a tree with nodes, in order to visualize an abstract prepared statement (parent), and its concrete executions (children). This is necessary, as we are interested in concrete executions, that may cause performance problems. Furthermore, as we want to offer the functionality to analyze a concrete prepared statement with its executed parameters and the related response time, we present the executed SQL statement, more precisely we substitute the variables with bonded values, within the SQL statement string. Afterwards, the abstract *Prepared Statement*, which is enriched by setters, is employed to reconstruct concrete executions. These are stored within the call hierarchy of the abstract *Prepared Statement*. Additionally, we present some time-related columns, which are the same, as for the *Aggregated Statements* view.

Chapter 11

Evaluation

In this chapter, we present the evaluation of our approach and its corresponding implementation. We start with a definition of our evaluation goals and our used metrics. Subsequently, we describe our set-up and the execution of our tests within our evaluation environment. We conduct a survey, to verify the functionality and usability of our tool. In the end, we analyze our tests and illustrate the devoted results.

11.1 Goals

Our main goal is to evaluate the functionality and usability of our software, which is divided into the monitoring, and the analysis and visualization component. The latter one is integrated into Kieker Trace Diagnosis. More precisely, we evaluate the functionality and usability of the whole software system. A software engineer should be able to use our approach, in the context of JDBC, to monitor database operations of a specific Java application. Furthermore, the visualization, which is based on the processed analysis, has to be easy-to-use. Filtering and sorting of database operations, like statements and prepared statements, is an essential part.

As we conduct a survey for our evaluation process, the goal is to identify how users interact with our software and how usable it is. This includes also the functionality of our software, the correctness of the presented data, and how useful it is, when the user employs our software, when conducting an analysis on database operations. Additionally, we want to know, if assigned questions within our survey, which contain a single task, are solvable with our software.

11.2 Methodology

This section describes our methodology, which is used in our usability experiment. We employ an usability test based on a survey, respectively questionnaire, we introduced in Chapter 2. Questionnaires have been used a long time to evaluate user interfaces [Root and Draper 1983]. We decided to create a survey based on the SUS questionnaire, and add severals tasks, as we are interested in the overall usability of our software and the usefulness to solve specific questions using it. The participants should verify the usability of our software. Additionally, we measure the overall time, each participant spent on the questionnaire.

11.3 Usability Experiment

Our usability experiment is structured into five, consecutive parts. We start with our developed questionnaire, continue with the experimental set-up, describe how we conduct the experiment, present the results, and close with a discussion.

11.3.1 Questionnaire

We choose a questionnaire, respectively survey, as an evaluation method to verify our software. Therefore, we developed a paper-based questionnaire for our usability testing experiment. We created questions, that may occur during a performance analysis of database operations within an existing software system.

It is important to choose well-defined questions within an usability test, as they directly influence the results of experiment [Nielsen and Landauer 1993]. According to Nielsen and Landauer, we should employ at least 30 test subjects within our experiment to get valid results.

Our questions are result-based tasks, as the user tries to solve the question on his own and we do not provide a fixed, predefined way to find a solution. Our questionnaire begins with less complex questions, e.g., naming the number of presented views, and ends with more complex understanding questions. This allows the test subject to get familiar with our software, so that we can raise the complexity in the latter questions. We divided our questionnaire in six parts. The first part covers personal questions and experiences within the context of database systems and performance analysis. The last part are debriefing questions about the usefulness and behavior of our software. We present the related parts in the following.

Personal Information

Our questionnaire begins with a part for personal information. We are interested in the overall *semester*, the *target degree*, and the experience level, regarding some skills, of our participants. For rating of their experience level, we employ a 5-point Likert Scale [Likert 1932] ranging from 0 (no experience) to 4 (expert with years of experience). The skills, to rate, are listed in the following:

▷ Java Programming

▻ Performance Analysis

▻ Database Systems

▻ SQL

▻ Kieker Trace Diagnosis

We choose these skills, since they are related to the expertise of a real user of our software. In order to compare our results, we need to measure the experience of our participants.

Introduction

In Table 11.1, our defined questions including their id within the questionnaire, their description, and maximum achievable points (score) are displayed. To give the subject time for becoming familiar with the tool, we start with simple questions, which focus on the navigation structure of our software.

Table 11.1. Description of the questions from the introduction part of our questionnaire

ID	Description	Score
2.1	How many different views, regarding database operations, are offered by the program?	1
2.2	How many statements, aggregated statements and prepared statements have been loaded?	3
2.3	How long was the overall analysis duration?	1

Statements

In this part, we start with specific questions according to our software. In our scenario, our participant is a software engineer, who conducts a performance analysis of a software system. He suspects, that database operations may cause performance problems. Therefore, he is interested in a statement, e.g., that has the highest response time. This is often a good starting point for further analysis. The related questions are shown in Table 11.2.

Aggregated Statements

An aggregated statement represents multiple executed SQL statements. One single statements call may not cause a performance issue, even if its response time is high. In contrast, a huge amount of executed similar SQL statements with an inconspicuous response time

Table 11.2. Description of the questions from the statements part of our questionnaire

ID	Description	Score
3.1	Name the trace ID and response time (in ms) of the statement, that has the highest response time.	1
3.2	What is its underlying calling Java operation?	1
3.3	What kind of SQL statement took the lowest amount of time?	1

can cause a performance issue. Therefore, we present related questions in Table 11.3. Especially in larger software systems this sort of questions may occur.

Table 11.3. Description of the questions from the aggregated statements part of our questionnaire

ID	Description	Score
4.1	Name the aggregated statement, its count of invocations, and the total response time (in μs) of the statement, that has the highest total response time.	3
4.2	Name the aggregated statement, its count of invocations, and the average response time (in μs) of the statement, that has the highest average time.	3
4.3	Name the aggregated statement, its count of invocations, and the average response time (in μs) of the statement, that has the lowest minimum response time.	3

Prepared Statements

This part of the questions is related to prepared statements. The user needs to employ our tree navigation within our corresponding view, in order to find a solution. In addition, the user is challenged to distinguish between an abstract and a concrete prepared statement. Altogether, this part is the most difficult one within our questionnaire. We present the related questions in Table 11.4.

Debriefing Questions

Our last part within the questionnaire is feedback oriented. We were interested in their opinion, related to a specific functionality or overall impression, of our software. Again, we employ a 5-point Likert scale, ranging from 0 (very difficult or very bad) to 4 (very easy or very good), and present the related questions in Table 11.5.

Table 11.4. Description of the questions from the prepared statements part of our questionnaire

ID	Description	Score
5.1	Name the abstract prepared statement, that has the highest response time.	1
5.2	Name, referred to the previous question, the concrete prepared statement parameter(s) or variable(s) and corresponding response time(s), that cause the highest response time.	2
5.3	Name the concrete prepared statement, that has the highest count of invocations. Also name its distinct parameters and response times.	3

Table 11.5. Description of the debriefing questions our questionnaire

ID	Description
6.1	How difficult was it to **navigate** through the program?
6.2	How difficult was it to **filter** and **sort** database statements for specific problems?
6.3	Was the program **easy** to use?
6.4	How was your **overall impression** of the tool?
6.5	Do you have any comments/suggestions concerning the usage of the program? Was something ambiguous or not clear?

Related keywords within the questions are marked bold, to emphasis the related functionality, we are interested in. The last question, with id 6.5, does not apply the Likert scale. Instead it uses a free text field.

11.3.2 Experimental Set-up

This section describes the experimental set-up for our evaluation. Prior to the set-up of our experiment, we needed to generate sample data for the usability test. Hence, we employed the aforementioned JPetStore in version 5.0 and executed it on our development system. As we used several machines and software configurations within our experiment, we explain them in the following.

Configuration

As the experiment takes place in an experimental laboratory, we provide an overview of the hardware and software configuration in the following.

Table 11.6. Employed hardware configurations within the experiment

	Machine 1	Machine 2	Machine 3
Model	Dell Optiplex 7010	Dell Optiplex 7010	Lenovo Thinksta-tion S10 6423
CPU	Intel Core i3 3220 2x 3.30GHz	Intel Core i5 3470 4x 3.20GHz	Intel Core 2 Duo E8400 2x 3.00GHz
RAM	8GB	8GB	8GB
Display Size	24 inch (widescreen)	24 inch (widescreen)	19 inch (standard)
Display Resolution	1920 x 1080 pixels	1920 x 1200 pixels	1280 x 1024 pixels

Hardware Configuration We employ three different machines, which are used during our experiment. As the underlying hardware differs, we present their relevant hardware specification in Table 11.6.

The hardware configuration is different, especially in display size and resolution. This circumstance is based on the varying hardware specification of the equipment within our computer lab. This provides not an ideal condition for conducting our experiment. Although our software does not require a high hardware specification and our application does not need a large display size with a resolution greater than 1280 x 1024 pixels, we are optimistic that this fact does not influence our experiment negatively. In an following experiment, we recommend to employ only machines with an identical hardware configuration, especially the used displays.

Software Configuration We employed Windows 7 Professional 64 Bit, including the first Service pack, as operation system for our experiment. Furthermore, we applied the Java Software Development Kit (SDK) in version 1.8, as our software system requires Java for the execution.

11.3.3 Execution of the Experiment

Before the experiment took place, we conducted a small pilot study with two students as participants. Therefore, we received feedback, which helped us to improve our questionnaire. Furthermore, we added hints or rephrased questions, which were mentioned as not clear or difficult. We utilized a paper-based questionnaire, which was described in Section 11.3.1, for our experiment. The test subjects had no time limit, and could work in parallel, as we prepared three machines for our purpose. At the beginning, each participant received a sheet of paper containing a short description of our experiment and the related context. This information sheet can be found in Appendix A. They were given sufficient time to read this paper. Afterwards, we informed the test subjects, that they can ask during their experiment session, if a question is not clear. Subsequently, the questionnaire started with personal questions and experiences within the context of database systems and performance analysis. Thereupon, we assigned tasks, that may occur during conducting a performance analysis of database operations. The experiment session ended with the debriefing questions.

11.3.4 Results

In this section, we present the measured results from our experiment, respectively questionnaire. We start with the personal questions, continue with our application related questions (Q2 - Q5), and close with our debriefing questions.

Personal Information

Our experiment inherited 36 participants, thereof one bachelor student, 32 master students, and 3 PhD students. In the first part of our questionnaire, we asked the test subjects to rate their experience level, regarding the skills *Java Programming*, *Performance Analysis*, *Database Systems*, *SQL*, and *Kieker Trace Diagnosis*. The results, including the specific skill, the mean, and the standard deviation, are shown in Table 11.7.

Table 11.7. Results of the rated experiences within the personal information part

Skill	**Mean**	**SD**
Java Programming	2.6111	0.5491
Performance Analysis	1.0833	0.6917
Database Systems	1.1666	0.6969
SQL	1.3888	0.7663
Kieker Trace Diagnosis	0.4722	0.6540

Introduction

We asked simple questions within this part to get the participants familiar with our software. These were mainly related to navigation structure. The results, including the question id, the mean, and the standard deviation, are displayed in Table 11.8

Table 11.8. Results of the introduction part

ID	Mean	SD
2.1	0.9166	0.2803
2.2	2.9166	0.5
2.3	1	0

Statements

In this part, we were interested in how our participants handle statements. The participants had to find and employ the related views to give a correct answer. The results are shown in Table 11.9.

Table 11.9. Results of the statements part

ID	Mean	SD
3.1	1.3333	0.6761
3.2	0.9722	0.1666
3.3	0.9722	0.1666

Aggregated Statements

As aggregated statements may cause performance issues, we asked the users related questions, which may occur during a performance analysis. The corresponding results are displayed in Table 11.10

Table 11.10. Results of the aggregated statements part

ID	Mean	SD
4.1	2.6388	0.6393
4.2	2.5833	0.7699
4.3	2.5277	0.8101

Prepared Statements

This part of the questions was related to prepared statements. Within the questionnaire, these questions were the most difficult one. The participants had to distinguish between an abstract and a concrete prepared statement, which could be challenging, if they were not familiar with these kind of database operations. We present the related results in Table 11.11.

Table 11.11. Results of the prepared statements part

ID	Mean	SD
5.1	0.9722	0.1666
5.2	1.6111	0.6448
5.3	2.25	1.2507

Debriefing Questions

Our last part within the questionnaire was designed to get feedback towards our developed software solution. We were interested in their opinion of our software. The results are shown in Table 11.12.

Table 11.12. Results of the debriefing questions part

ID	Mean	SD
6.1	3.3611	0.7616
6.2	3.1666	0.6094
6.3	3.2777	0.6594
6.4	3	0.7559

Additionally, the test subjects had to name comments or suggestions concerning the usage of our software. As the text answers are inappropriate to be listed here, they can be found within the raw results in Appendix B.

11.3.5 Discussion of Results

In this section, we discuss the results we presented in the previous section. The time, which the participants spent performing the questionnaire, resulted in 23.58 minutes in the average case, with a standard deviation of 5.69 minutes. Our two participants within the pilot study needed about 30 minutes for the test. This may be attributed to the refinement of our questionnaire based on the feedback of the pilot study. For our concrete task-related questions, which start with the introduction part (Q2) and end with the prepared statements part (Q5), we are interested in the average correctness for each question. In the following, we describe the different parts and analyze and discuss the results.

Personal Information

Based on this data, we can correlate the task-related answers with the experience of the test subjects. The test subjects rated their experience level of *Java Programming* between intermediate and advanced (average), which seems adequate, since most of them may have gained their experience within their bachelor or master course. The result for *Performance Analysis* was located at the beginner level in the average case. This circumstance is hardly surprising, as the students get rarely in touch with this process during their study. The results for *Database Systems* and *SQL* are nearby at the beginner level (average). Although there is a mandatory lecture within the bachelor course, the participants do not feel experienced enough, to rate their experience level as intermediate. Our last question was related to the usage of Kieker Trace Diagnosis, in which we had integrated our developed components. Our test subjects rated their skill between none and beginner, which is very reasonable as the tool is not published at the moment. Additionally, as we integrated the functionality towards database operations within our thesis, the related source code is also not available at this time.

Introduction

Our questions within the introduction part of our questionnaire have been designed with the purpose, to get familiar with our software. Therefore, our questions were simple and could be solved by exploring the navigation structure of our software. The average correctness for the related questions is shown in Figure 11.1.

We measured good average correctness results for our three Questions 2.1, 2.2, and 2.3. Hence, we imply that the first steps within our software are appropriate and our software has a good navigation structure. The latter implication is based on the 100% average correctness rate for Question 2.3. Additionally, we asked how long the duration of the analysis took place.

Statements

To solve our questions, regarding statements, the test subjects were required to apply for the first time our built-in sorting function, as we asked in Question 3.1 for the statement,

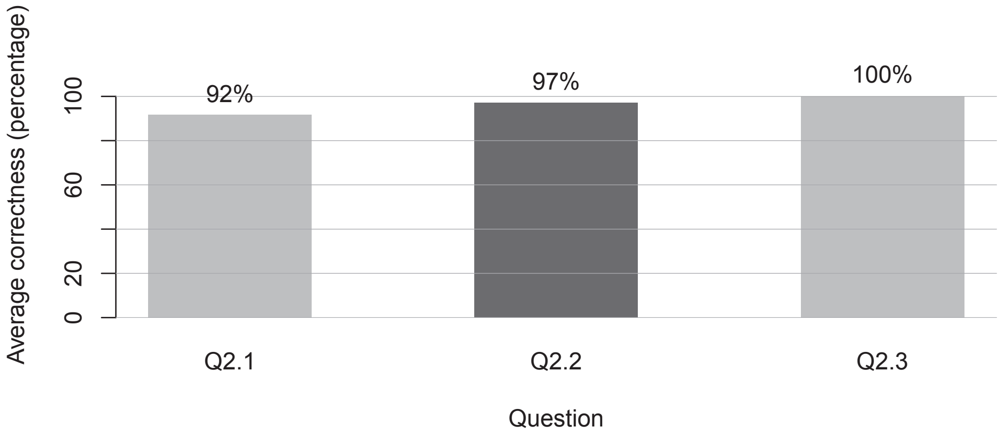


Figure 11.1. Average correctness per question within the introduction part

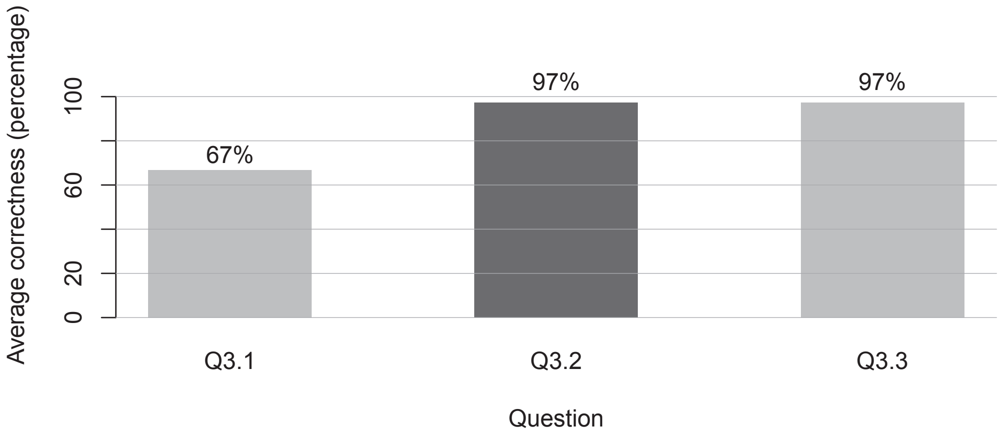


Figure 11.2. Average correctness per question within the statements part

which has the highest (total) response time. We present the average correctness for the related questions in Figure 11.2.

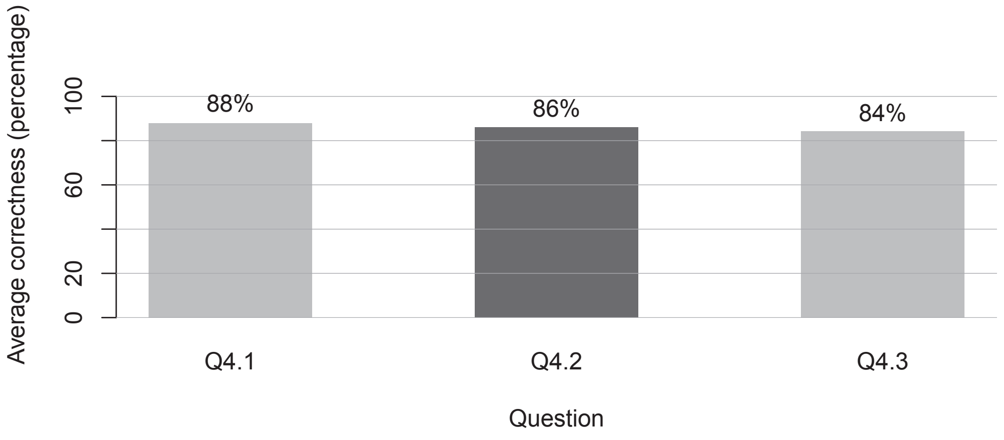


Figure 11.3. Average correctness per question within the aggregated statements part

Surprisingly, many participants were not able to name the correct answer, which is indicated through the low average correctness rate of 67%. One possible explanation may be, that the correct answer for Question 3.2 is an answer, which applies to multiple statements.

Aggregated Statements

The questions within the aggregated statements part had been more difficult than the previous questions. The average correctness for the Questions 4.1, 4.2, and 4.3 is shown in Figure 11.3.

These questions were designed to be similar to questions, which may occur during a performance analysis. We measured very positive average correctness results of 88% (4.1), 86% (4.2), and 84% (4.3). The answer to Question 4.1 and 4.2 were the same. This circumstance was intended to be a pitfall, in order to test the participants. Just a few participants, which answered Question 4.1. correctly, failed the following Question 4.2.

Prepared Statements

Our last task-oriented part within the questionnaire was related to prepared statements. The view, which had to be employed to solve our Questions 5.1, 5.2, and 5.3, was different to the previous ones. For the first time, a call tree structure was visualized and presented

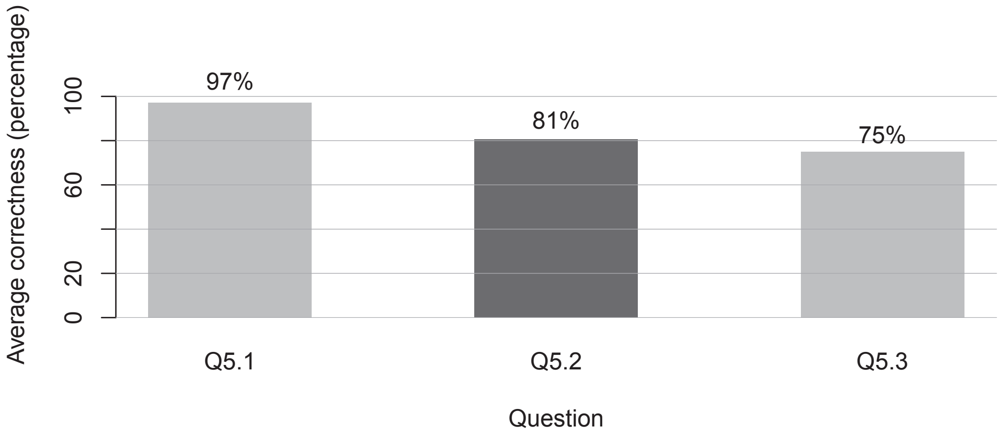


Figure 11.4. Average correctness per question within the prepared statements

a list of abstract prepared statements and their concrete executions, in form of children nodes. We present the average correctness for our questions in Figure 11.4.

As prepared statements are often used within large-scaled software systems, we are especially interested in the correctness of our results. Once again, we measured very positive average correctness rates of 97% (Question 5.1), 81% Question 5.2), and 75% (Question 5.3). We want to name Question 5.1, particularly. We achieved a rate of nearly 100%, which is a overwhelming good value, as we remember, that most participants rated their experience regarding database systems and SQL as beginner. Either the participants did not need to know exactly what was questioned and solved the question anyway, or they were able to deduct. In comparison, the rate of 75% for Question 5.3 seems a little bit left behind. This may be the result of the inaccuracy of this question. Some participants mentioned during the experiment, that this question was not clear. A better formulated question might have resolved in a better rate.

Debriefing Questions

Within the debriefing questions part, we wanted to get feedback from the participants based on their usage of our software during the experiment. In comparison to the previous parts, we were not interested in the average correctness of a question. Instead, we wanted to know how difficult or bad, respectively easy or good, some functionalities or the overall impression of our software were. The results are presented in Figure 11.5. The percentage is ranged from 0% (very difficult or very bad) to 100% (very easy or very good).

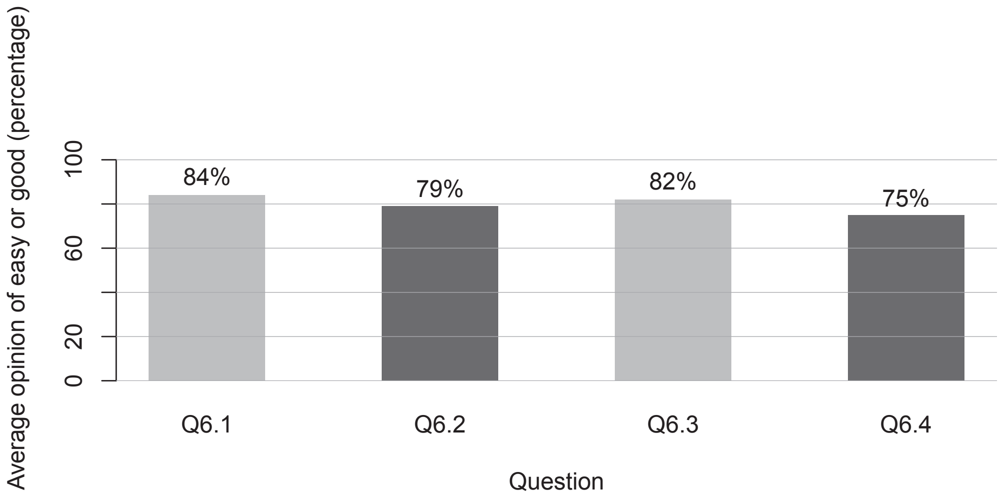


Figure 11.5. Average rating of easy or good per question within the debriefing questions

The participants rated the difficulty to navigate through our software between easy and very easy (84%, Question 6.1). In the same manner our filtering and sorting functionality were rated (79%, Question 6.2). The difference may be related to the limitation of the filtering, as it only accepts a regular expression. In the opinion of the test subjects, our software was easy to use (82%, Question 6.3), also the overall impression (75%, Question 6.4). Therefore, three of four users say, that the overall impression of our software is very good. The presented results are very good and indicate, that our software is applicable for conducting a performance analysis in the context of database operations.

11.3.6 Threats to Validity

Our experiment was performed on three different systems. Although we used more than one system, our experiment is not representative for all available systems. Furthermore, since we could not use the same hardware three times, it is possible, that the results are not completely comparable. Therefore, we suggest to conduct further experiments with the same hardware, for validating our results, and different object systems for a general validation.

Another threat concerns the assigned questions, which might not reflect real performance problems. We defined valid questions concerning a performance analysis of database operations within a software system, in order to verify our software for a real environment. Our test subjects were made up of bachelor, master, and PhD students. Therefore, they might have behave differently in comparison to professional software engineers or developers. Conducting a further experiment, employing only professionals, may quantify this impact.

11.4 Summary

Summarized, we observed positive feedback from our participants, which matches the high correctness rate of our questions (Q2 to Q5), in combination with the positive results from our debriefing questions (Q6). Although the difficulty of our questions increased within the questionnaire, the participants were nearly consistently able to give correct answers. This confirms that our software system have a bearing on conducting a performance analysis of database operations. As we have validated the usability of our developed tool through our conducted experiment, our software system is worthwhile to be further developed. Our raw results and paper-based questionnaire of the experiment is available in Appendix B, respectively Appendix C.

Chapter 12

Related Work

During our literature research, we only found a few suitable approaches within the research body, which were related to our context, monitoring database operations within a performance analysis of a software system. Ma et al. [2012] presented an approach towards monitoring public clouds and created a framework for this purpose. They provide a visualization of observed SQL statements, which shows their number of issues per second. As they are only interested in the kind of database operation, e.g., SELECT, CREATE, INSERT or DELETE, their visualization presents only aggregated calls towards the mentioned level. A more detailed inspection of specific database calls, which we offer in our developed tool, is not possible.

Apart from the research body, there are also exist tools in the industrial context. These are commercial and often very expensive. One of these tools is *InspectIT*, another monitoring framework developed by the company Novatec.[1] Although the source code of the tool is not available in terms of open source, the software can be used, based on the supplied license agreement. The tool was presented at the Symposium on Software Performance 2014 in Stuttgart.[2] It performs a dynamic analysis of Java programs during their execution using an instrumentation of the JVM based on the Java package *java.lang.instrument*.[3] In addition to the capabilities of the presented approaches, InspectIT allows to show executed SQL queries and their corresponding execution times. Our visualization approach was inspired by the tool *InspectIT*, as it offers a simple, similar visualization of information on database operations.

Another tool within this context is the *New Relic APM* tool,[4] supplied by the same-name company. The framework offers a wide range of features, from application response times to security audits. It also features database monitoring. Large companies, e.g., Kickstarter,[5] employ this monitoring framework. Their database monitoring component provides a detailed overview of database performance and indicates critical errors, which are slowing down an application. They are also ranking the most time consuming database calls and

[1] `http://www.inspectit.eu`

[2] `http://www.performance-symposium.org/fileadmin/user_upload/palladio-conference/2014/slides/08_inspectit.pdf`

[3] `http://docs.oracle.com/javase/6/docs/api/index.html?java/lang/instrument/package-summary.html`

[4] `http://newrelic.com`

[5] `http://www.kickstarter.com`

combine this information with aggregated metrics, e.g., response times, throughput, and slow SQL traces. Furthermore, they offer the ability to filter the database communication to show only operations and metrics of a specific type. In comparison to our approach, they additionally offer a visualization for showing slow (prepared) statements. The user can identify slow, problematic statements, without the need to search within a list of statements.

The company AppDynamics[6] offers, beside their APM solution, also a product for database monitoring. Their software is capable of troubleshooting performance issues in production environments, in order to diagnose the root cause of database-related performance issues, using fine-grained historical data. Additionally, they offer functionalities to tune and fix performance issues pro-actively, based on performance metrics, such as the consumption of resources or concrete execution plans. Furthermore, they claim to invoke a monitoring overhead of less than 1%. In comparison to our approach, the tool already offers features to fix performance problems or at least to optimize the performance.

Dynatrace[7] provides another APM solution with their product *Dynatrace Application Monitoring*. Although the details are not presented on their website, their software involves a monitoring of database operations. Furthermore, they visualize calls within the observed software. Additionally, they list response times and highlight potential problems. They also reveal related method arguments, return values, SQL statements, and more. Compared to our approach, their software provides more visualization options, e.g., trend analysis or anomaly detection.

[6] `http://www.appdynamics.com`
[7] `http://www.dynatrace.com`

Chapter 13

Conclusions and Future Work

In this chapter, we summarize our thesis and discuss future work.

13.1 Conclusions

In this thesis, we developed an aspect-oriented approach for performance monitoring of database operations. We described our implementation and conducted an evaluation, in form of an usability test based on a paper-based questionnaire in combination with our developed tool. Our constructed software is composed of three components, namely the *monitoring component*, the *analysis component*, and the *visualization component*. We designed our *monitoring component* to be generic and versatile, in order to allow multiple analysis and visualization tools to use this data as input. We integrated our *analysis component* and *visualization component* into Kieker Trace Diagnosis, which employs the Kieker monitoring record structure. Additionally, Fittkau integrated our *monitoring component* successfully into his monitoring tool ExplorViz [Fittkau 2015]. Based on these two technological integrations, we verified our monitoring tool as being versatile. Additionally, we conducted an experiment with 36 participants, in order to validate the usability of our developed software. In our experiment, we observed positive feedback from our participants, which correlated with the high average correctness rates of the results. Although the difficulty of our questions increased within the questionnaire, the participants were nearly consistently able to give correct answers. This confirms, that our software system has a bearing on conducting a performance analysis of database operations. As we have validated the usability of our developed tool through our conducted experiment, our software system is worthwhile to be further developed.

Our developed software supports a software engineer conducting a performance analysis on database operations. We offer the ability to observe database communication based on JDBC. More precisely, the software engineer is able to inspect executed database operation calls, which are in our case statements, aggregated statements, or prepared statements. We provide three related views within our visualization, which are based on the representation of *Call Tree Views*. This form of visualization allows the representation of the hierarchy of a database operation call. Furthermore, it offers an architecture, which is suitable for functions like filtering and sorting our recorded monitoring data. Our displayed record information includes, in the case of statements, the executed SQL statement and the response time. For aggregated statements we enrich this information with additional timings, such as the average, minimum, or maximum response time. These timings are calculated for a specific aggregated statement, based on the related statements. For prepared statements, we additionally provide the related abstract prepared statement and the concrete executions with bound variables (SQL parameters).

13.2 Future Work

During the development process, we developed new ideas to improve our software. Some points of the future work were identified by ourself, others were provided as feedback through our usability testing experiment. We describe these in the following.

We gather several information when we instrument a software system. Some database operations return the number of database entries, which were affected by this specific executed operation. This allows a software engineer to measure the impact of the concerned database operation. Additionally, we could offer more visualization options, e.g., views based on other approaches, for example 3D visualizations. Some participants of the experiment considered, that the loading process of a monitoring log file, or at least the completion of it, should be displayed in an adequate way. We agree, that an upcoming version should cover this issue. Another feedback is related to the filtering within Kieker Trace Diagnosis. Multiple users mentioned during our experiment, that the filtering functionality is inconvenient and difficult to use due the need to define a regular expression. An improvement towards an easier filtering, e.g., based on substrings, could enhance the usability of our software. With reference to our experiment, we suggest to conduct further experiments to validate the usability of our approach. More precisely, we recommend a controlled experiment, similar to [Fittkau et al. 2015a]. Additionally, the population could be extended, to include professional users, which may use and rate our tool differently, compared to the students we employed in our experiment.

Bibliography

[Ball 1999] T. Ball. The Concept of Dynamic Analysis. In: *Software Engineering*. Edited by O. Nierstrasz and M. Lemoine. Volume 1687. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pages 216–234. (Cited on page 5)

[Canfora Harman and Di Penta 2007] G. Canfora Harman and M. Di Penta. New Frontiers of Reverse Engineering. In: *2007 Future of Software Engineering*. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pages 326–341. (Cited on page 5)

[Chiba 1998] S. Chiba. Javassist - a reflection-based programming wizard for Java. In: *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*. 1998, page 174. (Cited on page 20)

[Chikofsky and Cross 1990] E. Chikofsky and I. Cross J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE* 7.1 (1990), pages 13–17. (Cited on page 5)

[Coble et al. 1997] J. M. Coble, J. Karat, M. J. Orland, and M. G. Kahn. Iterative usability testing: ensuring a usable clinical workstation. In: *Proceedings of the AMIA Annual Fall Symposium*. American Medical Informatics Association. 1997, page 744. (Cited on page 9)

[De Pauw et al. 2002] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of Java programs. In: *Software Visualization*. Springer, 2002, pages 151–162. (Cited on pages 29 and 55)

[Ducasse and Pollet 2009] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *Software Engineering, IEEE Transactions on* 35.4 (2009), pages 573–591. (Cited on page 6)

[Fittkau 2015] F. Fittkau. Live Trace Visualization for System and Program Comprehension in Large Software Landscapes. In: *PhD Topic Presentation*. 2015. (Cited on page 79)

[Fittkau et al. 2013a] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. In: *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. 2013, pages 1–4. (Cited on pages 13 and 28)

[Fittkau et al. 2013b] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and Live Trace Processing with Kieker Utilizing Cloud Computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013*. Volume 1083. CEUR Workshop Proceedings, 2013, pages 89–98. (Cited on pages 13 and 23)

[Fittkau et al. 2015a] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller. Comparing Trace Visualizations for Program Comprehension through Controlled Experiments. In: *23rd IEEE International Conference on Program Comprehension (ICPC 2015)*. 2015. (Cited on pages 28 and 80)

[Fittkau et al. 2015b] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: Visual Runtime Behavior Analysis of Enterprise Application Landscapes. In: *23rd European Conference on Information Systems (ECIS 2015)*. 2015. (Cited on pages 13 and 23)

[Florian Fittkau 2015] Florian Fittkau. ExplorViz Project. 2015. URL: http://www.explorviz.net/ (visited on 06/26/2015). (Cited on page 14)

[Fowler 2002] M. Fowler. Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co., Inc., 2002. (Cited on page 1)

[Froihofer et al. 2007] L. Froihofer, G. Glos, J. Osrael, and K. M. Goeschka. Overview and evaluation of constraint validation approaches in Java. In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society. 2007, pages 313–322. (Cited on page 20)

[Gamma et al. 1994] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: elements of reusable object-oriented software. Pearson Education, 1994. (Cited on page 23)

[Hasselbring 2011] W. Hasselbring. Reverse Engineering of Dependency Graphs via Dynamic Analysis. In: *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*. ECSA '11. Essen, Germany: ACM, 2011, 5:1–5:2. (Cited on pages 11, 27, 28)

[Jung et al. 2013] R. Jung, R. Heinrich, and E. Schmieders. Model-driven Instrumentation with Kieker and Palladio to forecast Dynamic Applications. In: *Proceedings Symposium on Software Performance: Joint Kieker/Palladio Days 2013 (KPDAYS 2013)*. Volume 1083. CEUR Workshop Proceedings. CEUR, 2013, pages 99–108. (Cited on page 11)

[Karat 1997] J. Karat. Evolving the scope of user-centered design. *Communications of the ACM* 40.7 (1997), pages 33–38. (Cited on page 8)

[Kiczales et al. 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. English. In: *ECOOP'97 — Object-Oriented Programming*. Edited by M. Akşit and S. Matsuoka. Volume 1241. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pages 220–242. (Cited on pages 7, 8)

[Kiczales et al. 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. English. In: *ECOOP 2001 — Object-Oriented Programming*. Edited by J. Knudsen. Volume 2072. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pages 327–354. (Cited on page 10)

[Knight and Munro 2000] C. Knight and M. Munro. Virtual but visible software. In: *Proceedings of IEEE International Conference on Information Visualization*. 2000, pages 198–205. (Cited on pages 13 and 27)

[Likert 1932] R. Likert. A technique for the measurement of attitudes. *Archives of psychology* (1932). (Cited on page 62)

[Ma et al. 2012] K. Ma, R. Sun, and A. Abraham. Toward a lightweight framework for monitoring public clouds. In: *Computational Aspects of Social Networks (CASoN), 2012 Fourth International Conference on*. IEEE. 2012, pages 361–365. (Cited on page 77)

[Müller et al. 2000] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. Reverse Engineering: A Roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pages 47–60. (Cited on pages 5, 6)

[Neilson et al. 1995] J. Neilson, C. Woodside, D. Petriu, and S. Majumdar. Software bottlenecking in client-server systems and rendezvous networks. *Software Engineering, IEEE Transactions on* 21.9 (1995), pages 776–782. (Cited on page 7)

[Nielsen and Landauer 1993] J. Nielsen and T. K. Landauer. A mathematical model of the finding of usability problems. In: *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*. ACM. 1993, pages 206–213. (Cited on page 62)

[Oracle 2011] Oracle. JDBC™ 4.1 Specification. 2011. (Cited on pages 1, 9, 17, 18)

[Pooley 2000] R. Pooley. Software Engineering and Performance: A Roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pages 189–199. (Cited on page 7)

[Project 2013] K. Project. Kieker User Guide. Apr. 2013. URL: `http://kieker-monitoring.net/documentation/`. (Cited on page 11)

[Root and Draper 1983] R. W. Root and S. Draper. Questionnaires as a software evaluation tool. In: *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM. 1983, pages 83–87. (Cited on page 61)

[Sabetta and Koziolek 2008] A. Sabetta and H. Koziolek. Measuring Performance Metrics: Techniques and Tools. English. In: *Dependability Metrics*. Edited by I. Eusgeld, F. Freiling, and R. Reussner. Volume 4909. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pages 226–232. (Cited on page 7)

[Sauro and Lewis 2012] J. Sauro and J. R. Lewis. Quantifying the user experience: Practical statistics for user research. Elsevier, 2012. (Cited on page 9)

[Tonella and Ceccato 2004] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In: *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. 2004, pages 112–121. (Cited on page 8)

[Tullis and Albert 2008] T. Tullis and W. Albert. Measuring the User Experience: Collecting, Analyzing, and Presenting Usability Metrics. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. (Cited on page 9)

[Tullis and Stetson 2004] T. S. Tullis and J. N. Stetson. A comparison of questionnaires for assessing website usability. In: *Usability Professional Association Conference*. 2004, pages 1–12. (Cited on page 9)

[Van Hoorn et al. 2009] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Technical report TR-0921. Department of Computer Science, Kiel University, Germany, Nov. 2009. (Cited on page 11)

[Van Hoorn et al. 2011] A. van Hoorn, S. Frey, W. Goerigk, W. Hasselbring, H. Knoche, S. Köster, H. Krause, M. Porembski, T. Stahl, M. Steinkamp, and N. Wittmüss. DynaMod Project: Dynamic Analysis for Model-Driven Software Modernization. In: *Joint Proceedings of the 1st International Workshop on Model-Driven Software Migration (MDSM 2011) and the 5th International Workshop on Software Quality and Maintainability (SQM 2011)*. Volume 708. CEUR Workshop Proceedings. Invited paper. 2011, pages 12–13. (Cited on page 1)

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. 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)*. ACM, Apr. 2012, pages 247–248. (Cited on page 11)

[Wettel and Lanza 2008] R. Wettel and M. Lanza. CodeCity: 3D Visualization of Large-Scale Software. In: *Companion of the 30th international conference on Software engineering*. ACM. 2008, pages 921–922. (Cited on page 27)

[Wulf et al. 2014] C. Wulf, N. C. Ehmke, and W. Hasselbring. Toward a Generic and Concurrency-Aware Pipes & Filters Framework. In: *Symposium on Software Performance 2014: Joint Descartes/Kieker/Palladio Days*. 2014. (Cited on pages 12, 23, 24)

[Xu et al. 2010] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software Bloat Analysis: Finding, Removing, and Preventing Performance Problems in Modern Large-scale Object-oriented Applications. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER '10. Santa Fe, New Mexico, USA: ACM, 2010, pages 421–426. (Cited on page 7)

Kiel University
Department of Computer Science
Software Engineering

Prof. Dr. Wilhelm Hasselbring
M.Sc. Christian Wulf

Assignments for Software Engineering for Parallel and Distributed Systems (SS 15)

Assignment 8

Issued at 26th June 2015 Due to 3th July 2015

Task 1: SQL Database Operations in Distributed Systems 12 points

Based on reverse engineering through dynamic analysis it is possible to perform a software performance analysis in order to detect performance bottlenecks or issues. These may have a negative effect concerning non-functional requirements of the software like increased execution times or memory usage.

Most distributed systems comprise at least one database that is often executed on a dedicated node. As performance problems are in many cases related to databases, it is often worth to take a look at database communication. Common application monitoring frameworks are usually limited to business operations. But for our specific use case, monitoring database operations, there is a lack of freely available tools, which support a developer in conducting a performance analysis. As a result, we developed an approach to aid this process.

In order to further develop our approach, we conduct a study at our working group. The study investigates the usability of our developed tool and should ratify the usefulness of our approach. During the study you will answer a few questions concerning the usage of our tool, while your are performing an analysis of recorded monitoring data. Neither you have to draw diagrams, nor you have to program. The expected solving time is about 30 minutes. Based on the number of correct answers, you can earn a maximum of 12 points.

To get an appointment, please enter your **Stu-Number** in the form of "stuXXXXX" at `http://doodle.com/chktht3uc2fizyge`. Please choose one time slot and remember your choice, because your registration will not be visible afterwards. The study takes place next week, starting on Monday 29.06. in room 1210 (CAP4). If you have further questions, feel free to contact Christian at (czi@informatik.uni-kiel.de).

Experiment: Raw Results

ID	TimeStart	CurrentSemester	TargetDegree	EProgramming	EPerformanceAnalysis	EDatabaseSystems	ESQL	EKTD	Q2.1	Q2.2	Q2.3	Q3.1
1	17:24	1	1	2	1	1	1	0	1	3	1	1
2	10:06	2	1	3	1	2	2	1	0	3	1	2
3	12:36	3	1	3	1	1	1	1	1	3	1	2
4	13:00	11	1	3	1	1	1	0	1	3	1	0
5	12:47	8	1	2	1	2	3	0	1	3	1	2
6	13:34		2	4	3	2	3	1	0	3	1	2
7	14:07	3	1	3	1	0	0	0	1	3	1	1
8	12:32	8	1	3	1	2	3	0	1	3	1	2
9	15:30	8	1	2	0	0	1	1	0	3	1	2
10	15:05	4	1	3	1	1	1	1	1	3	1	2
11	15:00	4	1	3	2	2	2	0	1	3	1	2
12	14:12	4	1	3	0	2	2	0	1	3	1	1
13	13:52	30	2	3	2	2	2	3	1	3	1	1
14	13:36	9	1	3	2	0	1	0	1	3	1	1
15	13:30	2	1	3	1	1	1	1	1	0	1	1

16	09:36	2	1	2	2	2	2	0	1	3	1	1
17	11:04	2	1	2	1	1	1	0	1	3	1	1
18	11:02	4	1	2	2	2	2	1	1	3	1	2
19	10:35	2	1	2	0	0	1	0	1	3	1	2
20	10:30	3	2	3	1	2	3	0	1	3	1	2
21	10:00	1	1	3	1	1	2	0	1	3	1	2
22	09:09	8	1	2	2	2	2	0	1	3	1	1
23	08:00	2	1	2	1	1	1	1	1	3	1	1
24	09:37	6	0	3	1	1	1	0	1	3	1	1
25	08:04	2	1	3	1	1	1	0	1	3	1	1
26	15:59	3	1	3	2	1	1	0	1	3	1	1
27	09:30	1	1	3	0	0	1	1	1	3	1	1
28	09:34	2	1	2	1	1	0	1	1	3	1	1
29	10:11	8	1	2	1	1	1	1	1	3	1	0
30	10:20	2	1	2	0	1	1	0	1	3	1	0
31	10:38	2	1	2	1	1	1	1	1	3	1	2
32	14:57	3	1	2	0	1	1	0	1	3	1	2
33	11:00	3	1	3	1	1	1	1	1	3	1	2

34	12:17	2	1	3	1	2	1	0	1	3	1	1
35	12:34	2	1	3	1	1	1	1	1	3	1	0
36	14:00	3	1	2	1	0	1	0	1	3	1	2

Q3.2	Q3.3	Q4.1	Q4.2	Q4.3	Q5.1	Q5.2	Q5.3	DQ6.1	DQ6.2	DQ6.3	DQ6.4	TimeEnd	DQ6.5
1	1	0	0	0	1	2	3	3	3	3	2	17:44	enable selection of multiple items in the list to display the number of selected items. suggestion according to previous question about numbers of items
1	1	3	3	3	1	2	2	4	3	4	4	10:29	filter allows just regular expressions, simple search not possible. Syntax of regular expression not clear
1	1	3	3	3	1	2	3	4	4	4	2	13:11	Time is floored to its unit resulting in response time 0 ms without floating point precision
0	1	2	2	3	1	2	3	1	2	3	2	13:27	Sorting was easy, but didn't use the filter (caused by reguglar expressions). The tool locks very boring (only text, no use of colors). Ambiguity – the survey is in english, but the program had a german translation. This was confusing. Dind't know whether I had to name java-methods or SQL
1	1	3	3	3	1	1	3	3	3	2	3	13:14	Is there an option to get the names/caluclation of the prepared statements?
1	1	3	3	3	1	2	0	4	3	3	4	13:50	not apparent: monitoring log statistics view contains database operations statistics. Aggregation of the same concrete prepared statement would be helpful
1	1	2	2	3	1	2	3	3	4	3	3	14:23	
1	1	3	3	3	1	1	2	3	2	2	2	12:59	sorting was easy, filtering not (not clear, how to use the filter). Ambigious how to change from milliseconds to microseconds. Clear and cleaned up window. Would be nice to have an easier way to identify statements than copy the whole statement.
1	1	3	3	3	1	1	3	3	3	4	3	15:55	
1	1	3	3	3	1	2	3	4	3	4	4	15:39	
1	1	3	3	3	1	2	3	3	4	3	3	15:20	wasn't clear that the log have to be loaded manually. So it was a little confusing for the first minute. After figuring this out, the rest was easy to use.
1	1	3	3	0	1	2	0	3	3	2	1	14:28	
1	1	2	2	3	1	0	0	1	3	4	2	14:10	There should be response to loading of data. Not necessary a dialog. Preferably, it should switch to „DB>Statements“
1	1	3	3	3	1	2	3	3	4	4	4	14:08	instead of microseconds, milliseconds
1	1	2	0	3	1	1	3	3	4	3	4	13:54	

1	1	3	3	2	1	2	0	4	3	4	4	09:50	columns are insufficient labeled. Avg, min and max inclu-de some time unit, but there is no repsonse time
1	1	2	2	3	1	1	3	3	3	3	3	11:25	
1	1	2	2	3	1	2	3	4	3	3	3	11:17	PreparedStatement view, bottom half, shows durations of 0 microseconds for concrete statements, unlike the tabular view
1	1	3	3	3	1	2	3	4	3	3	3	11:00	
1	1	3	3	2	1	1	3	3	3	3	3	10:56	concrete prepared statements bottom half view everytime 0 microseconds duration. After sorting the item with the previous index stays marked, the bottom half then not shows the marked item. Both cases may lead to errors
1	1	3	3	3	1	2	3	4	4	4	3	10:24	A pop-op after loading a monitoring log would be nice
1	1	2	2	2	1	1	2	4	3	4	3	09:34	It was unclear, if the loading of the monitoring data suc-ceeded, s start-screen would be nice.
1	1	3	3	2	1	2	3	3	3	3	2	08:23	
1	1	2	2	2	1	2	3	4	4	4	3	10:00	The last question was unclear for me. I like this classic, easy design. No fancy stuff, very good. A maybe for the design you could use a modern color etc. But that's no a must have.
1	1	3	3	3	1	2	3	4	4	4	3	08:20	
1	1	3	3	3	1	2	3	3	3	2	2	16:15	bad GUI design. Too much to write.
1	1	3	3	1	1	2	0	4	4	4	3	09:54	Maybe highlighting SQL or making them bold would be useful for a better distinction between them and tables
1	1	2	2	2	1	0	3	3	3	3	3	10:00	How can I change the time unit from ns to ms?
1	0	3	3	3	1	2	0	4	3	3	3	10:43	It took a bit to find out that you can change the timeunit. The sorting in the last task was a bit confusing. It seemes that you can only sort for the times of invocations of the abstract statement and not the concrete one.
1	1	2	2	3	0	2	3	3	3	3	3	10:44	The search is not working like expected
1	1	3	3	3	1	2	0	4	2	3	3	11:11	Question 5.3 was unclear for me. Concerning if it is still about the statement from 5.1
1	1	3	3	3	1	2	0	3	2	3	3	15:25	
1	1	3	3	3	1	1	3	4	3	3	3	11:24	Filtering should have example to show the structure of a regex pattern. Bottom view sometimes show 0s total dura-tion, although above it is not 0

1	1	3	3	2	1	2	3	4	4	4	4	12:39	Didn't see I could expand prepared statements at first. Otherwise fine, very intuitive
1	1	3	3	2	1	2	3	3	3	3	4	12:52	Everything looked fine to me
1	1	3	3	2	1	0	3	4	3	4	4	14:30	prepared statements: sorting of concrete statements not possible. Duration = 0 ms for all concrete statements in the bottom panel

1 Personal Information

1.1 Please note the **time**:

☐

1.2 Please name your **Stu-Number**:

☐

1.3 Please name your overall **semester** (“Fachsemester”):

☐

1.4 Target degree:

☐ Bachelor ☐ Master ☐ Ph.D

1.5 Please rate your experience level:

	None	Beginner	Intermediate	Advanced	Expert
Java Programming	☐	☐	☐	☐	☐
Performance Analysis	☐	☐	☐	☐	☐
Database Systems	☐	☐	☐	☐	☐
SQL	☐	☐	☐	☐	☐
Kieker Trace Diagnosis	☐	☐	☐	☐	☐

2 Introduction

2.1 How many different views, regarding database operations, are offered by the program?

2.2 How many statements, aggregated statements and prepared statements have been loaded?

2.3 How long was the overall analysis duration?

3 Statements

3.1 Name the Trace ID and response time (in ms) of the statement, that has the highest response time.

3.2 What is it's underlying calling Java operation?

3.3 What kind of SQL statement took the lowest amount of time?

4 Aggregated Statements

4.1 Name the aggregated statement, it's count of invocations, and the total response time (in μs) of the statement, that has the highest total response time.

4.2 Name the aggregated statement, it's count of invocations, and the average response time (in μs) of the statement, that has the highest average time.

4.3 Name the aggregated statement, it's count of invocations, and the average response time (in μs) of the statement, that has the lowest minimum response time.

5 Prepared Statements

5.1 Name the abstract prepared statement, that has the highest response time.

5.2 Name, referred to the previous question, the concrete prepared statement parameter(s) or variable(s) and corresponding response time(s), that cause the highest response time.

5.3 Name the concrete prepared statement, that has the highest count of invocations.
Also name it's **distinct** parameters and response times.

6 Debriefing Questions

6.1 How difficult was it to **navigate** through the program?

very difficult ☐ ☐ ☐ ☐ ☐ very easy

6.2 How difficult was it to **filter** and **sort** database statements for specific problems?

very bad ☐ ☐ ☐ ☐ ☐ very good

6.3 Was the program **easy to use?**

very difficult ☐ ☐ ☐ ☐ ☐ very easy

6.4 How was your **overall impression** of the tool?

very bad ☐ ☐ ☐ ☐ ☐ very good

6.5 Do you have any comments/suggestions concerning the usage of the program?
Was something ambigious or not clear?

6.6 Please note the **time**: