

Uncovering Performance Antipatterns by Systematic Experiments

Diploma Thesis of

Alexander Wert

At the Department of Informatics
Institute for Program Structures
and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr. Walter F. Tichy
Advisor:	Dr.-Ing. Jens Happe
Second advisor:	Dipl.-Inform. Dennis Westermann

Duration:: 01. October 2011 – 19. April 2012

I declare that I have developed and written the enclosed Diploma Thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 19. April 2012

Acknowledgements

I would like to thank my advisors Dr.-Ing. Jens Happe and Dipl.-Inf. Dennis Westermann for their great support and valuable feedback. It was a pleasure to work under your supervision.

My thanks go also to Prof. Dr. Ralf H. Reussner and Prof. Dr. Walter F. Tichy for the opportunity to write this thesis and for reviewing this work.

Furthermore, I would like to thank the team at SAP for active discussions, helpful ideas and the great working atmosphere. Especially, I thank Christoph for the great teamwork.

I would like to express my thanks to Olga for her detailed proof-reading of this thesis.

Finally, a very special word of thanks goes to my family. In particular, I thank my wife Oxana, my mother Irina and my brother Andreas for your enduring and essential support. Without your help this thesis would not have been possible!
Thank You!

Abstract

As the size and complexity of enterprise applications increase, it becomes more and more challenging to develop software systems exhibiting a satisfactory performance behaviour. Software Performance Engineering (SPE) aims for addressing this problem by applying engineering principles during software development. Software Performance Antipatterns are an established SPE concept describing recurrent design and development flaws leading to low software performance. Detecting Software Performance Antipatterns in present software artifacts can serve as a feedback mechanism for software architects and developers.

Existing approaches for detecting Software Performance Antipatterns are either model-based or monitoring-based concepts. While model-based approaches can be used in early development phases, they are not suited for capturing performance flaws made during implementation. Monitoring-based approaches overcome this problem by searching for antipatterns in software systems during runtime. However, the antipatterns which can be found by monitoring-based approaches depend on the actual workload submitted to the system under test.

In this thesis, we introduce a performance antipattern detection approach which is based on systematic measurement experiments. Through systematic measurement experiments we are able to conduct goal-oriented search for Software Performance Antipatterns and their root causes in an effective way. For this purpose, we combine the concept of dynamic instrumentation with an adaptive approach of executing measurement experiments. Based on a hierarchy of performance problems we define a detection process which is guided by a decision tree. Using a decision tree increases the efficiency of the search process as unnecessary measurement steps can be avoided. For selected Software Performance Antipatterns we introduce and compare different detection techniques selecting the best ones for the overall approach.

We evaluated our detection approach on an implementation of the TPC-W Benchmark. In this evaluation scenario, we discovered a performance antipattern leading to a scalability problem. Moreover, we were able to identify the root cause responsible for the detected problem. These results show the applicability of the detection approach introduced in this thesis.

Zusammenfassung

Da die Größe und Komplexität von Geschäftsanwendungen stetig steigt, wird es zunehmend schwieriger Software-Systeme zu entwickeln, die ein zufriedenstellendes Performance-Verhalten aufweisen. Ingenieurmäßige Software-Entwicklung geht dieses Problem an, indem ingenieurmäßige Prinzipien während der Software-Entwicklung eingesetzt werden. Software-Performance-Anti-Patterns stellen ein etabliertes Konzept in der ingenieurmäßigen Software-Entwicklung dar und beschreiben wiederkehrende Entwurfs- und Entwicklungsfehler, die zu schlechter Software-Performance führen. Die Erkennung von Software-Performance-Anti-Patterns in bestehenden Software-Artefakten kann Entwicklern und Software-Architekten als ein Feedback-Mechanismus dienen.

Bestehende Ansätze zur Erkennung von Software-Performance-Anti-Patterns sind entweder Modell-basiert oder Monitoring-basiert. Zwar können Modell-basierte Ansätze in einer frühen Phase der Software-Entwicklung eingesetzt werden, jedoch sind diese Ansätze nicht geeignet, um Fehler zu erkennen, die während der Implementierung entstehen. Monitoring-basierte Ansätze lösen dieses Problem durch eine messbasierte Suche nach Anti-Patterns während dem Betrieb der Zielanwendung. Allerdings hängt die Anzahl an Anti-Patterns, die durch Monitoring-basierte Ansätze erkannt werden können, stark von der tatsächlichen Last ab, die am Zielsystem anliegt.

In dieser Arbeit stellen wir einen Ansatz zur Erkennung von Software-Performance-Anti-Patterns vor, der auf systematischen Messexperimenten beruht. Durch systematische Messexperimente sind wir in der Lage eine zielorientierte Suche nach Software-Performance-Anti-Patterns und deren Ursachen auf eine effektive Weise durchzuführen. Hierfür kombinieren wir das Konzept der dynamischen Instrumentierung mit dem Ansatz einer adaptiven Ausführung von Messexperimenten. Wir definieren einen Detektionsprozess, der basierend auf einer Hierarchie von Performance-Problemen einen Entscheidungsbaum zur Steuerung des Ablaufs verwendet. Die Verwendung eines Entscheidungsbaums erhöht die Effizienz des Detektionsprozesses, da unnötige Messungen vermieden werden können. Wir stellen für einige der betrachteten Software-Performance-Anti-Patterns Erkennungsverfahren vor und vergleichen diese, um die besten Verfahren für den Gesamtansatz zu selektieren.

Wir haben unseren Ansatz an einer Implementierung des TPC-W Benchmarks evaluiert. In diesem Evaluationszenario haben wir ein Software-Performance-Anti-Pattern und dessen tatsächliche Ursache erkannt welche zu einem Skalierbarkeitsproblem der Anwendung führt. Die Evaluierung zeigt die praktische Anwendbarkeit des vorgestellten Erkennungsansatzes.

Contents

1. Introduction	1
1.1. Software Performance Engineering	1
1.2. Performance Antipatterns as an SPE Concept	2
1.3. Idea	3
1.4. Overview	3
2. Fundamentals	5
2.1. Software Performance Antipatterns	5
2.1.1. The Nature of Software Performance Antipatterns	5
2.1.2. Performance Antipattern Categorization Template	7
2.1.2.1. Observable Behaviour	7
2.1.2.2. Scope of Observation	7
2.1.2.3. Indicators	8
2.1.3. Description and Categorization of Software Performance Antipatterns	8
2.1.3.1. The Blob Antipattern	8
2.1.3.2. The Empty Semi Trucks Antipattern	9
2.1.3.3. The Stifle Antipattern	11
2.1.3.4. The Ramp Antipattern	11
2.1.3.5. The Traffic Jam Antipattern	12
2.1.3.6. The One Lane Bridge Antipattern	13
2.1.3.7. The More is Less Antipattern	14
2.1.3.8. Unbalanced Processing Antipattern	14
2.1.3.9. The Dormant References Antipattern	15
2.1.3.10. Session as a Data Store Antipattern	16
2.1.3.11. The Sisyphus Database Retrieval Performance Antipattern	16
2.1.3.12. The Circuitous Treasure Hunt Antipattern	17
2.1.3.13. The Tower of Babel Antipattern	17
2.1.3.14. Unnecessary Processing Antipattern	18
2.1.3.15. Excessive Dynamic Allocation Antipattern	18
2.1.3.16. Spin Wait Antipattern	18
2.1.4. Software Performance Antipatterns at a Glance	19
2.2. Instrumentation	21
2.2.1. Aspect Oriented Programming	21
2.2.2. Structural Reflection and HotSwap	22
2.2.3. SIGAR API	23
2.2.4. Kieker	24
2.3. Mathematical Foundations	25
2.3.1. Central Tendency	25
2.3.2. Measures of Dispersion	26

2.3.3.	Comparing two Samples	26
2.3.3.1.	Confidence Interval	27
2.3.3.2.	T-Test	27
2.3.4.	Linear Regression	28
2.4.	Foundations on Performance Evaluation	28
2.4.1.	Performance Modeling	29
2.4.2.	Measurement-Based Performance Evaluation	30
2.5.	Software Performance Cockpit	31
3.	Related Work And Contribution	33
3.1.	Related Work	33
3.2.	Contribution	35
4.	Approach	37
4.1.	Big Picture On The Antipattern Detection Approach	37
4.2.	The Adaptive Measurement Approach	39
4.3.	Antipattern Detection Architecture and Process	41
4.4.	General Assumptions	45
4.4.1.	Monitoring Overhead	45
4.4.2.	Knowledge About Usage	45
4.4.3.	No Disturbing Sources	45
4.4.4.	Fixed Environment	45
4.4.5.	Standard Technologies	45
4.4.6.	Byte Code Analysis	46
4.5.	Summary	46
5.	Instrumentation	47
5.1.	Full Instrumentation	47
5.2.	Dynamic Instrumentation	49
5.2.1.	Dynamic Instrumentation with AOP and Kieker	49
5.2.2.	Dynamic Instrumentation using Javassist, HotSwap and Kieker	50
6.	Antipattern Detection in Detail	53
6.1.	Validation System and Scenarios	55
6.1.1.	Online Banking System	55
6.1.2.	Scenarios	56
6.2.	The Varying Response Times Detection	60
6.2.1.	Experiment Setup	60
6.2.1.1.	Problem Specific Instrumentation	60
6.2.1.2.	Proper Workload for Experiments	60
6.2.2.	Detection Techniques	61
6.2.2.1.	Fixed COV Threshold	61
6.2.2.2.	Adapted COV Threshold	61
6.3.	The Ramp Detection	63
6.3.1.	Detection	63
6.3.1.1.	Analyses on Chronologically Continuous Measurement Data	63
6.3.1.2.	Separated Time Windows Analysis	67
6.3.2.	Root Cause Analysis	68
6.3.2.1.	Realization	68
6.3.2.2.	Evaluation	70
6.3.3.	Summary	71
6.4.	The Dormant References Detection	71
6.4.1.	Experiment Configuration	71

6.4.2. Analyzing Memory Consumption	72
6.4.3. Evaluation	73
6.5. The One Lane Bridge Detection	74
6.5.1. Detection	74
6.5.1.1. Experiment Configuration	74
6.5.1.2. Direct Blocking Times Analysis	74
6.5.1.3. Approximated Blocking Times Analysis	76
6.5.2. Root Cause Analysis	78
6.5.2.1. Synchronized Methods Root Cause	78
6.5.2.2. Database Lock Root Cause	80
6.5.3. Summary	82
7. Evaluation	83
7.1. TPC-W Benchmark	83
7.1.1. System Under Test: Bookstore Emulator	84
7.1.2. Workload Specification	85
7.1.3. TPC-W Solution Model	86
7.2. Experiment Setup	86
7.3. Results	88
7.3.1. The Varying Response Times Problem	88
7.3.2. The Ramp	90
7.3.3. The One Lane Bridge	90
7.3.4. Interpreting the Result	91
7.4. Summary	93
8. Conclusion	95
8.1. Summary	95
8.2. Future Work	96
Bibliography	99
Appendix	105
A. Validation Scenarios	105
B. TPC-W Measurements	106
B.1. Varying Response Times	106
B.2. The Ramp	106
B.3. One Lane Bridge Detection	107
B.4. One Lane Bridge Root Cause Analysis	108
List of Figures	109
List of Tables	111
Abbreviations	113

1. Introduction

1.1. Software Performance Engineering

Software performance is an important aspect of software systems influencing the success of software vendors as well as of their customers. Bad software performance causes higher costs, as it results in higher consumption of employees' working time, lower production rates and power consumption inefficiencies. In addition, bad software performance might lead to high service response times, bad scalability and impair the system's reliability. This deteriorates customer relationships as customer satisfaction decreases. For software vendors, software performance is a decisive competitive factor. In any case, ensuring satisfactory software performance is an important task.

For this purpose, the field of Software Performance Engineering (SPE) has been established. In contrast to the "Fix It Later" approach, where performance is dealt with only when performance problems occur, SPE aims for evaluating and managing software performance during the entire software lifecycle. As in general, late problem discovery and solution is quite expensive, an important goal of SPE is to detect and fix performance problems as early as possible. Therefore, it is important to gather performance requirements and design the software architecture with respect to these requirements in early phases. During development and test phases, the developer has to be provided with performance feedback, so that performance problems can be discovered early. Finally, during operation and maintenance phases it is important to keep performance at a satisfactory and stable level.

We distinguish two conceptually different approaches for software performance evaluation: model-based and measurement-based approaches. The former use an abstracted representation of the real system (a model) to evaluate the performance characteristics of the real system. Performance models can be solved either by mathematical analysis or by simulation. While analytically solving a model is cheaper, simulations provide more accurate results as less assumptions have to be met. Model-based performance evaluation is relatively cheap as for this purpose the real system does not have to be implemented or realized. However, the accuracy of model-based performance evaluation depends on the detail level of the model. Often, it is difficult to build detailed, representative models which results in inaccurate performance predictions. Measurement-based performance evaluation approaches promise more accurate results as they capture the real system's behaviour. However, measurement-based approaches assume implemented, running systems or fragments to be measured. Thus, while model-based performance analysis can be

applied in early development phases, measurement-based approaches can be used at least when some implementations are present.

Despite the great progress of SPE in research, to this day, SPE has been relatively neglected in development. This is due to limiting factors like available budgets, negotiated development time horizons and personnel restrictions. Furthermore, software developers focus on realizing additional features rather than evaluating the performance of existing services. Despite the high importance of performance, conducting performance evaluation manually is a time-consuming and expensive task yielding no directly visible result compared to implementing new features [Sma06]. In addition, performance evaluation presumes expertise knowledge, which is another obstacle for software developers to perform SPE. Thus, SPE is still a challenging research area. In particular, it is crucial to make SPE easily applicable for software developers as well as reduce costs, knowledge and effort required to conduct performance evaluation.

1.2. Performance Antipatterns as an SPE Concept

As a reaction to the software crisis [NR69] in the mid-sixties, the software engineering (SE) discipline has been founded. The aim of SE is to apply engineering approaches to software development in order to produce software in more structured, effective and efficient manner resulting in high quality software systems.

For instance, the concept of *Design Patterns* [AIS⁺77, Gam95] is an important engineering approach software engineers have learned from construction engineering. In SE, design patterns describe common, well-approved solutions to recurring software design problems. Thus, design patterns describe best practices for structuring software. The concept of *Antipatterns* is closely related to design patterns. However, antipatterns describe recurring problem solutions which should be avoided as they have a negative impact on the software's quality attributes. Thus, instead of characterizing best practices, antipatterns describe recurring mistakes. While design patterns are wide-spread, well-known and often used in the area of SE, antipatterns are less known as they can not be applied as simple as design patterns. In contrast to design patterns, which are used consciously, in most cases mistakes leading to antipatterns are made unconsciously. Compared to design patterns, the antipattern concept is a reactive rather than a proactive approach. Thus, one way to take advantage of the antipattern concept is to detect antipatterns in developed architectures or software fragments in order to provide feedback to the software architect, developer respectively.

While different kinds of design antipatterns affect different software quality attributes, in this thesis, we focus on design antipatterns which impair the performance of the software under development (*software performance antipatterns* (SPA) [SW00]). In particular, we design a detection approach for SPAs which serves as a performance feedback mechanism informing the software architects and developers about performance flaws caused by bad design or implementation. In recent work, approaches have been proposed for detecting SPAs in architectural models based on UML [CME10] or the Palladio Component Model (PCM) [TK11]. While these approaches are suitable for detecting high level architecture antipatterns at design time, many performance mistakes unconsciously made during the implementation phase cannot be captured. Existing measurement-based approaches (e.g. [PM08]) for performance antipattern detection apply monitoring techniques during system operation in order to gather system behaviour information. Based on this information rules are defined which are used for the detection of antipatterns. As these approaches depend on monitoring during operation, the set of SPAs which can be detected depend on the actual workload. Furthermore, excessive monitoring causes high measurement overhead. In order

to overcome these problems, in this thesis, we introduce an SPA detection approach based on systematic measurement experiments.

1.3. Idea

As mentioned before, monitoring-based approaches for performance problem detection entail some disadvantages. Firstly, these approaches depend on the actual workload submitted to the system under test during operation. Thus, the search for performance problems is randomized in some manner. Secondly, for meaningful measurement results excessive monitoring is required which causes high monitoring overhead. However, high monitoring overhead impairs measurement accuracy. On the other hand, selective monitoring reduces the amount of available information which is required for analysis. Finally, monitoring-based approaches are mostly applicable only during operation. However, detecting performance problems lately during the software development process causes high costs for solving these problems.

In this thesis, we introduce and combine some concepts to overcome these problems. Instead of randomly searching for performance problems, we apply systematic measurement experiments allowing us to control which workload is submitted to the system under test. As we perform target-oriented experiments, the probability to find a specific performance problem is higher than the one of an approach which relies on a random workload. Furthermore, for each experiment we can specifically determine which data should be collected. In this way, we reduce the monitoring overhead during measurements without impairing the quality of information captured through measurements. Coupling the systematic measurement concept with dynamic instrumentation [MCC⁺95] allows us to find performance problems not only effectively but also efficiently. In particular, a search process guided by a decision tree enables the avoidance of unnecessary measurements which increases the efficiency significantly. Finally, our approach is intended to be used during the development phase rather than operation. In this way, costs can be avoided for solving performance problems lately.

1.4. Overview

In this section, we introduce the structure of this thesis:

- **Chapter 2:**

In Chapter 2, we introduce some foundations required for the understanding of the concepts describes in this thesis. In particular, we describe the concept of antipatterns and give an overview of existing software performance antipatterns (cf. Chapter 2.1). In Chapter 2.2, we introduce techniques and tools for instrumenting code in order to gather measurement data. Furthermore, we provide fundamentals on mathematical concepts in Chapter 2.3. In Chapter 2.4, we introduce approaches for performance evaluation. Finally, we explain the concept of the Software Performance Cockpit in Chapter 2.5.

- **Chapter 3:**

In Chapter 3, we give an overview on related work and describe the contribution of this thesis.

- **Chapter 4:**

In Chapter 4, we introduce the antipattern detection approach. We explain the main idea in Chapter 4.1. Furthermore, we describe the adaptive measurement approach in Chapter 4.2, the architecture of the detection concept in Chapter 4.3 and important assumptions in Chapter 4.4.

- **Chapter 5:**

As code instrumentation forms the basis of the detection approach, in Chapter 5, we discuss different instrumentation techniques comparing the full instrumentation technique (5.1) with dynamic instrumentation (5.2).

- **Chapter 6:**

In Chapter 6, we examine different detection techniques for individual software performance antipatterns. In particular, we investigate the Varying Response Time problem in Chapter 6.2, the Ramp antipattern in Chapter 6.3.1.1, the Dormant References antipattern in Chapter 6.4 and the One Lane Bridge antipattern in Chapter 6.5.

- **Chapter 7:**

We evaluate our approach in Chapter 7 on the TPC-W Benchmark. We introduce the TPC-W Benchmark in Chapter 7.1 and describe the experiment setup in Chapter 7.2. Finally, we present the detection results in Chapter 7.3.

- **Chapter 8:**

In Chapter 8, we conclude this thesis summarizing and discussing the proposed concepts and define issues for future work.

2. Fundamentals

2.1. Software Performance Antipatterns

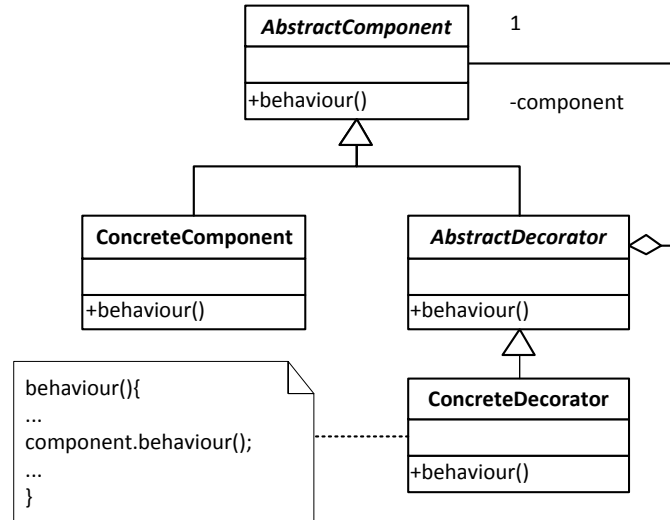
2.1.1. The Nature of Software Performance Antipatterns

As the term *Software Performance Antipattern* (SPA) is a key-element in this thesis, in this section, we introduce the concept of software performance antipatterns and describe some performance antipatterns found in literature.

Design Patterns [GHJ⁺02] are a well known concept in the area of software engineering (SE). Software engineers use design patterns for structuring software and creating software architectures for modularity, maintainability, efficiency and other extra-functional attributes. Design patterns are widely spread because they provide good and well-tried solutions to recurrent design problems. Design patterns originated from civil engineering [AIS⁺77] where design patterns describe common solutions to recurring problems when designing buildings. Based on the model of civil engineering, software engineers use design patterns for solving recurring problems. Using design patterns in software engineering was an important step towards making software development an engineering discipline.

All design patterns are described by a name, problem description and a solution. Thus, design patterns are not only common solutions to recurring problems, but form a language which can be used to describe a complex software architecture in a more abstract and simple way. In Figure 2.1 the *Decorator Pattern* is depicted as an example for a design pattern. The Decorator Pattern addresses the problem of dynamically modifying the behaviour of existing objects and provides a dynamic alternative to inheritance. The general solution to this problem is depicted in Figure 2.1 where a *ConcreteDecorator* modifies the behaviour a *ConcreteComponent* by referencing it and decorating the *behaviour()* method call.

While design patterns represent best practices for structuring software, *Design Antipatterns* describe common problems which should be avoided as they have a negative effect on certain extra-functional attributes of the software. Similar to design patterns, antipatterns are characterized by a name, a problem description and a solution. Here, the problem description refers to a recurring, bad approach to manage, structure and develop software. An antipattern's solution suggests better alternatives to the problematic approach or suggests steps to reduce the negative impact of the problematic approach. According to Brown [Bro98], antipatterns can be classified into three groups: *Development Antipatterns*, *Architecture Antipatterns* and *Software Project Management Antipatterns*. A fragmental antipattern hierarchy is depicted in Figure 2.2. While architectural antipatterns

Figure 2.1.: Decorator Pattern, [GHJ⁺02]

impair extra-functional issues like maintainability, modularity, etc., project management antipatterns have negative effects on issues like productivity during a software development process, complexity or project costs. Development antipatterns affect negatively some QoS attributes of the software. Thus, we further distinguish between *Performance Antipatterns*, having a negative effect on performance, and antipatterns affecting other QoS attributes. In this thesis we consider only performance antipatterns. These can be either technology specific (EJB, .NET, ...) or technology independent. At the bottom level, we classify antipatterns by their cause. Thus, we distinguish between performance antipatterns caused by bad design (*Performance Design Antipatterns*), improper deployment of the software system (*Performance Deployment Antipatterns*) or inefficient implementation (*Performance Implementation Antipatterns*).

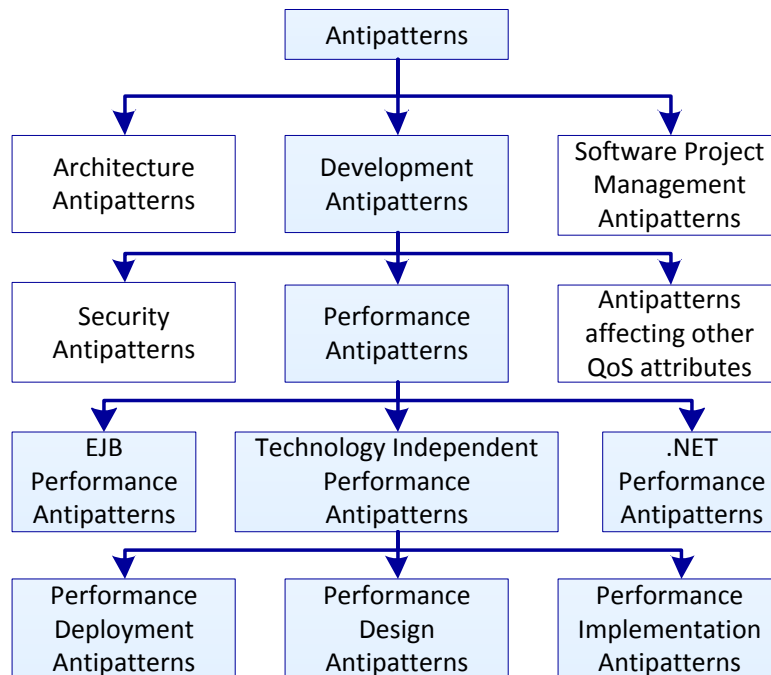


Figure 2.2.: Antipattern hierarchy (following and extending [PM08])

2.1.2. Performance Antipattern Categorization Template

In Section 2.1.3, we describe a compilation of software performance antipatterns (SPA) found in literature (cf. Table 2.1). These SPAs differ from each other not only in their negative impact on performance but also in metrics and information characterizing them. Thus, for a systematic approach to antipattern detection we have to extract common behaviour and characteristics of individual SPAs. For this purpose, we develop a categorization template which we apply on each antipattern in Section 2.1.3. For each antipattern we provide a description of its main characteristics and evaluate each antipattern in regard to the feasibility of detecting them by systematic measurement experiments. Then, we categorize each SPA we evaluated as feasible using the three categorization variables described in the following.

2.1.2.1. Observable Behaviour

As SPAs negatively affect the performance of the considered software system, they all exhibit a certain dynamic behaviour leading to the negative effect. Thus first, we group the SPAs by their observable behaviour. For this purpose, we distinguish four groups of behaviour: *calling behaviour*, *response time progression*, *threading behaviour* and *memory consumption behaviour*.

- **Calling Behaviour**

Calling behaviour means that the considered software system or software component exhibits an inefficient interaction behaviour. In particular, this might be excessive method calling, database accessing or messaging behaviour. While in practice excessive method calling is less critical, inefficient remote communication, messaging or database accessing may have a considerable impact on the performance.

- **Response Time Progression** A scalable and good performing software system should have little variation in response times as greatly varying response times indicate instability and unreliability leading to unsatisfied customers. Thus, we speak of bad response time progression if the response times vary greatly.

- **Threading Behaviour** A bad threading behaviour addresses the problem of inefficiently or ineffectively utilizing concurrent processing. That includes unnecessary synchronization points, unbalanced distribution of resources to threads and thrashing. Thus, bad threading behaviour is the primary reason for limited scalability of software systems leading to bad performance while increasing the workload.

- **Memory Consumption Behaviour** Antipatterns which lead to a high memory consumption impair the software performance. High memory consumption leads to excessive memory swapping or slows down algorithms which work on data structures. In both cases the performance and the scalability of the target software system decrease.

2.1.2.2. Scope of Observation

Besides their difference in behaviour, SPAs differ in the scope where individual antipatterns can be observed. Although some SPAs exhibit the same class of observable behaviour they might be observed in different scopes. For instance, antipatterns exhibiting an excessive *Calling Behaviour* can be observed in the scope of messaging or in the scope of accessing a database. For the categorization of SPAs we use the following five scopes:

- **messaging:** We assign the messaging scope to an SPA if the observable behaviour can be recognized just by observing the usage of a messaging service.

- **database access:** This scope is assigned to database related SPAs.
- **service execution:** SPAs which can be recognized by observing the execution behaviour of top level system services are assigned the service execution scope.
- **concurrency:** The concurrency scope describes antipatterns whose observable behaviour is caused during concurrent execution of several execution threads.
- **memory consumption:** Finally, SPAs exhibiting a peculiar memory consumption behaviour are assigned the memory consumption scope.

2.1.2.3. Indicators

The third categorization variable (*Indicators*) addresses the problem of recognizing circumstances in observations which indicate certain SPAs. We distinguish between high level indicators abstracting from individual antipatterns and specific indicators pointing to the existence of single SPAs. High level indicators are not sufficient to decide whether a certain SPA exists or not, but serve as simple “direction signs” for further detection processing. Low level indicators are used to differentiate between single SPAs in an SPA category and between different root causes of an SPA. We formulate indicators as formal terms describing circumstances which can be observed by measuring the execution of the software system under test.

2.1.3. Description and Categorization of Software Performance Antipatterns

In this section we apply the categorization template on each SPA we found in literature. Table 2.1 gives an overview of all considered SPAs. The left part comprises all SPAs we evaluated as feasible for detecting them through systematic measurement experiments.

Feasible SPAs		Infeasible SPAs	
Antipattern Name	Source	Antipattern Name	Source
The Blob	[SW00]	Sisyphus Database Retrieval	[DGS02]
Empty Semi Trucks	[SW03a]	Circuitous Treasure Hunt	[SW00]
The Stifle	[DAKW03]	The Tower of Babel	[SW03a]
The Ramp	[SW03b]	Unnecessary Processing	[SW03b]
The Traffic Jam	[SW02b]	Excessive Dynamic Allocation	[SW00]
The One Lane Bridge	[SW00]	Spin Wait	[BPSH05]
More is Less	[SW03b]		
Unbalanced Processing	[SW03b]		
Dormant References	[Ray07]		
Session as a Data Store	[sap]		

Table 2.1.: Overview on software performance antipatterns

2.1.3.1. The Blob Antipattern

Description

The Blob [SW00] (also known as the “God Class”) describes a problem in structuring software. The Blob is a class or component which is responsible for the entire processing using other classes or components only for retrieving required information. As information is separated from the processing unit, the Blob causes high messaging overhead while retrieving required information during processing. However, excessive messaging results

in bad performance, in particular if messaging is conducted in a distributed environment. Another form of the Blob is a class or component which contains the entire information other classes or components need for processing. The negative effect on performance is the same as with the first case, only that in this case other classes (or components) access the Blob in order to retrieve information.

In both cases, the root cause for bad performance is the separation of data and the behaviour related to this data. Thus, the solution of the Blob antipattern is to keep processing units and related information together. Instead of separating data from processing units, structuring should be carried out by separating semantic units. This solution increases not only the maintainability of the software, but improves the performance by reducing the messaging overhead, as well.

Evaluation

- **Observable Behaviour:** The Blob antipattern leads to inefficient messaging producing too many calls which impair the performance of the software system. Thus, the Blob exhibits a striking *Calling Behaviour*.
- **Scope of Observation:** The scope for observing the Blob is the *messaging scope*. In this context messaging comprises all kinds of interaction between software components, in particular remote communication.
- **Indicators:** A software system exhibiting the characteristics of the Blob antipattern inevitably generates a high messaging overhead. Thus, if the proportion of the overall residence time the system spends with messaging (p_{msg}) exceeds a certain threshold T_{msg} , then high response times might be caused by the Blob antipattern. A possible high level indicator is:

$$T_{msg} < p_{msg} = \frac{R'_{msg}}{R'} \quad (2.1)$$

Here, R' is the overall residence time while R'_{msg} is the time the system performs messaging. The absolute value of T_{msg} has to be derived empirically.

If the high level indicator shows a high messaging overhead, a specific indicator is required which differentiates the Blob antipattern from other SPAs causing high messaging overhead. According to the description of the Blob, we can suspect the Blob antipattern if we identify a component which sends much more requests ($\#msg_{OUT}$) as it receives ($\#msg_{IN}$) or vice versa. This indicates that the considered component has a close dynamical dependency to other components. In particular, this applies to situations where the considered component interacts with many different components. Thus, a possible specific indicator is the following:

$$\#msg_{IN} << \#msg_{OUT} \vee \#msg_{IN} >> \#msg_{OUT} \quad (2.2)$$

2.1.3.2. The Empty Semi Trucks Antipattern

Description

Similar to the Blob antipattern, the *Empty Semi Trucks* [SW03a] is a performance antipattern concerning messaging behaviour. Sending a message from system A to system B is always entailed by an overhead like meta-data or processing tasks required to send and receive a message. As this overhead is often nearly constant for one message independent of the actual message size, it is obvious that sending a larger payload in one message is cheaper than splitting it into several messages. The Empty Semi Trucks antipattern addresses the problem of sending data from system A to system B in many small messages instead of aggregating it into a few bigger messages. The negative performance effect

entailed by the Empty Semi Trucks antipattern is a high messaging overhead resulting in higher response times. Root causes for this antipattern can be either inefficient use of the available bandwidth or an inefficient interface which does not allow for sending aggregated messages. In Figure 2.3, an example for an improper interface is depicted. In that example we want to send person data (name, age and gender) from one system to another using a custom messaging interface. We consider two possibilities to design this messaging interface. Using *MessagingInterface A* implies that three messages have to be sent in order to transmit data for one person. However, using *MessagingInterface B* allows us to first aggregate person data before sending it as one message. Thus, using *MessagingInterface A* implies a messaging overhead which is up to three times bigger than it is the case with *MessagingInterface B*.

MessagingInterface A	Person	MessagingInterface B
+ sendName(String name) + sendAge(int age) + sendGender(String gen)	+ setName(String name) + setAge(int age) + setGender(String gen)	+ sendPersonInfo(Person p)

Figure 2.3.: Empty Semi Trucks antipattern: example of an improper interface

Hence, sending aggregated messages (Batching performance pattern [SW02a]) and providing proper interfaces (Coupling performance pattern [SW02a]) solves the performance problem entailed by the Empty Semi Truck antipattern.

Evaluation

As the effect of the Empty Semi Trucks antipattern is quite similar to the one of the Blob antipattern, classes for the observable behaviour and the scope of observation are the same as for the Blob. Thus, the high level indicator pointing to a high messaging overhead is the same as for the Blob, too. However, the specific behaviour of the Empty Semi Trucks antipattern differs from the Blob resulting in another specific indicator.

- **Observable Behaviour:** As it was the case with the Blob, the Empty Semi Trucks antipattern exhibits a striking *Calling Behaviour*.
- **Scope of Observation:** The Empty Semi Trucks antipattern can be observed in the scope of *messaging*.
- **Indicators:** The high level indicator is the same as for the Blob:

$$T_{msg} < p_{msg} = \frac{R'_{msg}}{R'} \quad (2.3)$$

The Empty Semi Trucks antipattern is characterized by a high message frequency and a small average size of sent messages. Thus, the specific indicator for this antipattern is:

$$f_{msg} = \frac{\#msg}{t} > T_{freq} \wedge \overline{size(msg)} < T_{size} \quad (2.4)$$

Whereby, f_{msg} is the message frequency defined by the number of messages $\#msg$ per time unit t and T_{freq} is a threshold to be defined. T_{size} is a threshold for the average message size $\overline{size(msg)}$.

2.1.3.3. The Stifle Antipattern

Description

In general, the *Stifle* antipattern [DAKW03] is a special case of the Empty Semi Trucks antipattern applied on database accesses. Equivalent to messaging, a database access entails processing overhead. Thus, executing a series of SQL statements is significantly cheaper if it is performed in a batch rather than as a series of single database accesses. This performance problem is addressed by the Stifle antipattern entailing the performance problem of a relatively high database access overhead. High overhead caused by the database management system can be reduced if database statements and queries are aggregated at the application layer before they are sent to the database layer in order to be executed in a batch process.

Evaluation

As a special case of the Empty Semi Trucks antipattern, the Stifle can be treated in a similar way.

- **Observable Behaviour:** Equivalent to the Empty Semi Trucks antipattern, the Stifle exhibits a striking *Calling Behaviour*.
- **Scope of Observation:** The Stifle is a database related antipattern. Thus, the striking behaviour can be observed within the scope of *accessing a database*.
- **Indicators:** Similar to the high level indicator of the Blob and the Empty Semi Trucks antipattern, we define the high level indicator for the Stifle. While the Empty Semi Trucks causes a high messaging overhead, the Stifle entails a high database overhead. If the proportion of database residence time p_{db} exceeds a threshold T_{db} , the Stifle antipattern might be one possible reason for bad performance. Thus, we define the following high level indicator for the Stifle antipattern:

$$T_{db} < p_{db} = \frac{R'_{db}}{R'} \quad (2.5)$$

Whereby, R' is the overall residence time while R'_{db} is the time the system accesses a database. Again, a proper value for T_{db} has to be determined empirically.

A high number of database calls per service request $\#calls_{DB}$ paired with a small average number of result rows $\#rows$ specifically indicates the Stifle antipattern:

$$\#calls_{DB} > T_{call} \wedge \overline{\#rows} < T_{rows} \quad (2.6)$$

2.1.3.4. The Ramp Antipattern

Description

The Ramp [SW03b] is a rather general performance antipattern characterized by increasing response times (memory consumption, etc.) over operation time. The behaviour of the Ramp can be generalized to the time behaviour of any resource consumption. For example, another form of the Ramp can be observed if the memory consumption increases while the system is used. The Ramp behaviour is depicted in Figure 2.4 where the actual response time (or memory consumption) increases with operation time indicated by the linear regression function. Such behaviour leads to impaired performance although the workload was not increased significantly over time. As the Ramp performance problem is described at a high abstraction level, there might be different causes for this behaviour. Two possible reasons are the Sisyphus Database Retrieval Performance Antipattern (described in Section 2.1.3.11) and the Dormant References Antipattern (described in Section 2.1.3.9). Due to the fact that the Ramp is a rather general performance problem, the solution to this performance antipattern depends on the actual cause.

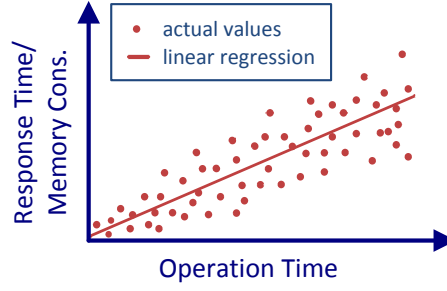


Figure 2.4.: Illustration of the Ramp antipattern, following [SW03b]

Evaluation

- **Observable Behaviour:** The response time variant of the Ramp leads to increasing response times. Observed over time, this results in high response time variance. In this case, one can observe a striking *Response Time Progression Behaviour*. In the case of the memory consumption variant of the Ramp, we observe a striking *Memory Consumption Behaviour* as memory consumption increases over time. However, as growing memory consumption leads to increasing response times we focus on the *Response Time Progression Behaviour*.
- **Scope of Observation:** In both cases, the striking behaviour can be noticed by observing the system execution at service level. Thus, the scope of observation is the *service execution scope*.
- **Indicators:** The *Response Time Progression Behaviour* is specified by a high variance in response times. As the coefficient of variance is a normalized metric, we use $COV(R)$ as an indicator for the *Response Time Progression Behaviour*. Thus, if $COV(R)$ exceeds a threshold T_{COV} , the Ramp might be a possible cause for bad performance. Furthermore, we can assume the Ramp antipattern if the response times increase significantly over time:

$$R(t + x) > R(t) \quad (2.7)$$

2.1.3.5. The Traffic Jam Antipattern

Description

Similar to the Ramp antipattern, the *Traffic Jam* [SW02b] antipattern describes a general response time behaviour of a software system. The Traffic Jam behaviour is characterized by a temporary overload situation of the software system (or a part of the software system) leading to long queues at some (passive or active) resources. Such overload situations often result in big variance of response times as some requests get stuck in congestion while others do not. However, a big variance of response times is a negative performance characteristic of a software system as the system users perceive rather high response times than the average response time. There might be different reasons for a Traffic Jam, one of them is the One Lane Bridge antipattern (described in Section 2.1.3.6). Another cause for the Traffic Jam antipattern is a temporarily high workload resulting in the described overload situation. Depending on the root cause there are different solutions to the Traffic Jam antipattern. In the case of a temporarily high workload it is advisable to utilize load balancer in order to distribute the workload uniformly among available resources or to perform admission control in order to guarantee a feasible workload.

Evaluation

- **Observable Behaviour:** Similar to the Ramp antipattern the Traffic Jam leads to highly varying response times exhibiting a striking *Response Time Progression Behaviour*.
- **Scope of Observation:** The *Response Time Progression Behaviour* can be observed at the *service execution* level.
- **Indicators:** The high level indicator is the same as for the Ramp antipattern:

$$COV(R) > T_{COV} \quad (2.8)$$

In the case of the Ramp antipattern the response time variance is based on increasing response times over time. Thus, the Ramp occurs even under a low workload. However, a Traffic Jam occurs only under high traffic volumes. In the case of the Traffic Jam antipattern, the variance in response times depends on the workload intensity. Thus, we can presume a Traffic Jam antipattern if the high response time variance is paired with the following specific indicator:

$$COV(R) \approx WL \quad (2.9)$$

Here, “ \approx ” means that the variance increases with the workload.

2.1.3.6. The One Lane Bridge Antipattern

Description

The *One Lane Bridge* [SW00] typically occurs in multi-threaded software systems, where only few threads can resume execution while most threads have to wait. Synchronization points in the program flow or database locks are often reasons for the One Lane Bridge. At a One Lane Bridge only few threads are active concurrently. Thus, less work can be processed resulting in higher response times. Furthermore, as mentioned before the One Lane Bridge can lead to a congestion resulting in the Traffic Jam antipattern. Solving the One Lane Bridge antipattern is not always possible as in many cases synchronization points and database locks are inevitable. However, one can try to reduce the impact of the One Lane Bridge by reducing the time required to “overcome” the synchronization phase. In particular, synchronizations should be avoided if they are not really necessary. For example, it is not necessary to synchronize concurrent reading accesses to a variable.

Evaluation

- **Observable Behaviour:** As the One Lane Bridge occurs in multi-threaded systems, this antipattern exhibits a striking *Threading Behaviour*.
- **Scope of Observation:** Depending on the root cause of the One Lane Bridge the scope of observation might be either the *concurrency scope* (in the case of synchronized methods) or the *database access scope* (in the case of database locks).
- **Indicators:** As the One Lane Bridge mostly leads to the Traffic Jam antipattern the high level indicator is the same as for the Ramp and the Traffic Jam antipatterns:

$$COV(R) > T_{COV} \quad (2.10)$$

If the difference between the response time R and the CPU time t_{CPU} of a service request is quite large, we can assume large waiting times. However, large waiting times indicate the presence of the One Lane Bridge. Thus, the specific indicator for the One Lane Bridge is the following:

$$R \gg t_{CPU} \quad (2.11)$$

2.1.3.7. The More is Less Antipattern

Description

If a software system is occupied with management tasks rather than performing actual work, we speak of *threshing*. The *More is Less* antipattern [SW03b] occurs when too many tasks are executed concurrently resulting in threshing due to high management overhead. Some reasons for the More is Less antipattern are too many concurrent threads, database connections, network connections or pooled resources. Figure 2.5 illustrates exemplary the throughput behaviour depending on the number of concurrent threads. The More is Less antipattern occurs when the number of concurrent threads exceeds n^* causing higher management overhead and, thus, decreasing the throughput of the system. In order to avoid the More is Less antipattern it is important to determine n^* experimentally or by simulation. Knowing n^* allows for creating thread and connection pools of proper size so that threshing can be avoided. Smith et al. [SW03b] describe further (case specific) solutions to this antipattern, like using a dedicated thread for background processing.

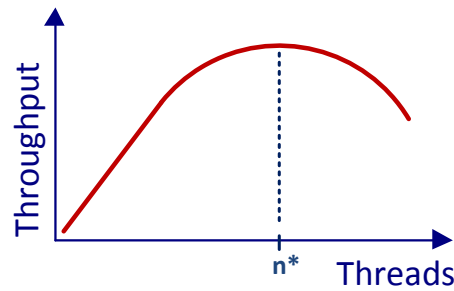


Figure 2.5.: More is Less antipattern: example throughput behaviour

Evaluation

- **Observable Behaviour:** Most variants of the More is Less antipattern depend on concurrent execution of several threads. For instance, holding several database connections is reasonable only if there are several concurrent threads using these connections. Thus, we assume that the number of considered resources (causing the More is Less antipattern) increases with the number of concurrent service requests. Consequently, the behaviour we can observe in the case of a More is Less antipattern is a striking *Threading Behaviour*.
- **Scope of Observation:** The scope of observation of the More is Less antipattern depends on the actual root cause. If threshing is caused by too many threads the scope is *concurrency*. If threshing is caused by too many database connections, the behaviour can be observed in the scope of *database accessing*.
- **Indicators:** Independent of the actual root cause, the More is Less antipattern results in response times R growing over linear with the number of concurrent service requests X :

$$R \propto X^a; a \gg 1 \quad (2.12)$$

The over linear growth is caused by progressively increasing overhead for resource management.

2.1.3.8. Unbalanced Processing Antipattern

Description

The *Unbalanced Processing* antipattern [SW03b] occurs in multi-threaded systems when scalability is impaired by unbalanced execution of concurrent tasks. There are three kinds

of unbalanced processing. i) If scalability is impaired due to single-threaded code or because all processors are dedicated to other tasks, one observes the *Concurrent Processing Systems* antipattern. ii) In a *Pipe and Filter Architecture* the slowest filter determines the throughput of the entire system. iii) Finally, one speaks of *Extensive Processing* if several processes compete for a processor, whereby, a long running process employs the processor for a long fraction of time blocking short processes.

In all three cases, there is no balanced distribution of running processes (or threads) to available processors resulting in bad scalability which is a negative performance property. Solutions to these balancing problems are described in [SW03b].

Evaluation

- **Observable Behaviour:** The Unbalanced Processing antipattern is another multi-threading related SPA. Thus, this antipattern can be recognized by its *Threading Behaviour* and its utilization of available resources.
- **Scope of Observation:** Consequently, the scope of observation is the *concurrency scope*.
- **Indicators:** On the top level, the Unbalanced Processing antipattern is indicated by an imbalance of resource utilization paired with a long queue of waiting threads:

$$U(CPU_A) \gg U(CPU_B) \wedge QL(T) > T_{QL} \quad (2.13)$$

The Extensive Processing manifestation of this antipattern is specifically indicated if a thread T^* exists which employs a CPU (t_{CPU}) much more than other threads:

$$\exists T^* : \forall T : t_{CPU}(T^*) \gg t_{CPU}(T) \quad (2.14)$$

Here, $t_{CPU}(T)$ is the CPU time of thread T . Indicators for the Concurrent Processing Systems variant depend on the actual root cause.

2.1.3.9. The Dormant References Antipattern

Description

The *Dormant References* antipattern [Ray07] can be a root cause for the Ramp antipattern. A *dormant reference* is a reference that points to an object which has been used in the past which, however, will not be used in the future anymore. The Dormant References antipattern is a specific memory leak which can affect the performance negatively in two ways: Either the memory leak results in high memory consumption forcing the operating system to additional swapping or the antipattern occurs in a distending repository impairing processing times on that repository. The latter case might lead to the Ramp antipattern, as repository operations like searching or sorting will traverse dormant references as well, although these are not relevant anymore. As new entries are added to the repository, the response times for these operations increase. Thus, missing clean-up operations are the root causes for this problem. Hence, the Dormant References antipattern can be solved by ensuring to remove all references to an object which will not be used anymore.

Rayside et al. [Ray07] describe other memory antipatterns as well, however, we focus on the Dormant References antipattern as it might be a reason for the Ramp antipattern.

Evaluation

- **Observable Behaviour:** As the Dormant References antipattern results in increasing memory consumption, this antipattern exhibits a striking *Memory Consumption Behaviour*.
- **Scope of Observation:** In order to recognize this antipattern, data has to be collected in the scope of *memory consumption*.
- **Indicators:** A high level indicator for high memory consumption is the *memory footprint* of a service request. A memory footprint is defined by the difference between used memory space before and after a service request:

$$mf = M(t_{after}) - M(t_{before}) > T_{mf} \quad (2.15)$$

Thus, if the memory footprint exceeds a certain threshold T_{mf} , high memory consumption can be assumed.

If the Dormant References antipattern is the root cause for the Ramp antipattern, this can be indicated by increasing memory consumption over time:

$$M(t+x) > M(t) \quad (2.16)$$

2.1.3.10. Session as a Data Store Antipattern

Description

The *Session as a Data Store* antipattern [sap] occurs if sessions are used as a kind of data store. Misusing sessions as data stores hurts scalability of the software system since sessions become heavy weight objects consuming much memory and hindering the system to execute many concurrent sessions efficiently. Instead of storing data directly into sessions, data should be persisted, for example in a database.

Evaluation

- **Observable Behaviour:** Equivalent to the Dormant References the Session as a Data Store antipattern results in a striking *Memory Consumption Behaviour*.
- **Scope of Observation:** Again, we have to observe the *memory consumption scope*.
- **Indicators:** If a session is misused as a data store, the memory consumption increases with a growing number of concurrent service requests / sessions. Thus, we can presume this antipattern if the memory consumption M is proportional to the number of concurrent sessions $\#sessions$:

$$M \propto \#sessions \quad (2.17)$$

However, if the system runs out of memory, this leads to thrashing and an over linear growth of response times. Thus, a high level indicator for the Session as a Data Store antipattern is the same as for the More is Less antipattern:

$$R \propto X^a; a \gg 1 \quad (2.18)$$

2.1.3.11. The Sisyphus Database Retrieval Performance Antipattern

Description

As mentioned in Section 2.1.3.4, the *Sisyphus Database Retrieval Performance* antipattern [DGS02] is a special case of the Ramp antipattern. The Sisyphus antipattern describes a performance problem which occurs when retrieving an entire set of data from a database although only a subset is needed. Even though the size of the requested subset is constant over time, the response times of the requests grow as the size of the base set increases. The solution to this antipattern is based on advanced search algorithms retrieving only the required subset. Solutions to this antipattern are described in more detail in [DGS02].

Evaluation

For detecting the Sisyphus antipattern the detection program would have to know whether the entire result of a query is used or only a subset. Thus, semantics about the usage of the query result is required. However, this knowledge is not available.

2.1.3.12. The Circuitous Treasure Hunt Antipattern

Description

The *Circuitous Treasure Hunt* antipattern [SW00] is an extension of the Stifle antipattern. While the Stifle antipattern addresses the problem of executing a series of SQL statements as single database accesses, the Circuitous Treasure Hunt antipattern additionally emphasizes the semantic dependency between these individual SQL statements. More precise, the Circuitous Treasure Hunt antipattern points out the problem of structuring a database in a way that a block of information can only be retrieved by executing single requests where one request depends on the result of the previous request. The negative performance impact of this antipattern is the same as it is the case with the Stifle antipattern. In order to solve the Circuitous Treasure Hunt antipattern the database structure has to be refactored (cf. [SW00]).

Evaluation

Although the Circuitous Treasure Hunt antipattern is related to the Stifle antipattern, which we think of to be detectable by a measurement-based approach, it is a difficult task to decide whether a detected Stifle antipattern actually is a Circuitous Treasure Hunt. The only difference between the Stifle and the Circuitous Treasure Hunt are the dependencies between single database queries (cf. Section 2.1.3.12). Detecting these dependencies requires either semantic knowledge about the queries content or the ability to trace information flow. Both issues are beyond the scope of this thesis.

2.1.3.13. The Tower of Babel Antipattern

Description

The *Tower of Babel* antipattern [SW03a] concerns transformation processes of data representations. If two communicating systems use different representations for the same information, it is necessary to convert one representation to another before data is interchanged. The same problem may also occur if two systems have the same data representations but the exchange format defers from this representation. Format conversions are expensive tasks, in particular if data is exchanged often and in significant amounts. Thus, recurrent transformation steps may impair the system's performance significantly. Eliminating unnecessary transformation steps and using the *Fast Path* pattern [SW03a] are proper solutions to this antipattern.

Evaluation

In order to detect this antipattern the detection program would have to recognize automatically code fragments where data is transformed to another representation. As there is no standard code for data transformation, there is an endless number of possible transformation implementations sharing no common characteristics. Thus, recognizing such code fragments is not a trivial task and requires semantics about corresponding code fragments, which is beyond the scope of this thesis.

2.1.3.14. Unnecessary Processing Antipattern

Description

As the name suggests, the *Unnecessary Processing* antipattern [SW03b] addresses the problem of executing program code which is of no use for the actual work or is not needed at the current processing point. For example, creating a database connection is unnecessary processing if it is not clear yet, whether the database will be used at all. The database connection could be created if it is sure that the connection will be of any use. Obviously, unnecessary processing extends the response times unnecessarily. Solving the Unnecessary Processing antipattern is case specific and requires code refactorings or restructurings in the most cases.

Evaluation

The problem to detect Unnecessary Processing is the same as with the Tower of Babel antipattern. It is not possible to decide automatically whether a code fragment is necessary or not, as it again requires knowledge about the semantics of the corresponding code fragment.

2.1.3.15. Excessive Dynamic Allocation Antipattern

Description

The *Excessive Dynamic Allocation* [SW00] is a quite low level performance antipattern concerning dynamic allocation of objects. Dynamic allocation is often used to keep a system flexible as objects are “alive” only for the time window they are really needed. However, allocating and cleaning up memory space might be expensive if it is done excessively. Recycling objects and use resource pools [SW00] can be some solutions to this antipattern.

Evaluation

Under the assumption that a programming language allows only to create objects but not to destroy them, high memory consumption of a function (*memory footprint*) seems to be an indicator for Excessive Dynamic Allocation as it indicates that many (or few big) objects have been created. However, the memory consumption of a function is not a reliable indicator for this antipattern. As already mentioned, there is no guarantee that many objects have been created, it could have been only a few big object, too. Furthermore, with systems like Java or .Net, there is no guarantee that dynamically created objects have not been removed from memory by the garbage collector. Thus, the function’s memory footprint is not sufficient for detecting the Excessive Dynamic Allocation antipattern. We do not see adequate alternatives to detect this pattern and for that reason we will not consider this antipattern in the following.

2.1.3.16. Spin Wait Antipattern

Description

The concern of the *Spin Wait* antipattern [BPSH05] is a multi-threading system where threads perform active waiting for a condition instead of going to sleep until they are notified about a state update. Active waiting implies bad performance as threads consume resources (CPU) although they do not perform any useful work. As mentioned before, it is more efficient if threads are deactivated (sleeping) while they wait for a condition. Threads can be notified and requested to resume as soon as the corresponding condition is fulfilled.

Evaluation

As the Spin Wait antipattern refers to actively waiting threads, for detecting this antipattern it is crucial to distinguish actively waiting threads from working threads. However, in general, this distinction is not possible. One could argue that active waiting is implemented as an empty loop, however, this is not always the case. For example, active waiting could also be implemented in a way that certain conditions are calculated in the loop-body. In such a case, again, knowledge about the code's semantics is needed in order to decide whether a loop is an implementation of active waiting. Furthermore, there is no significant difference in resource utilization between active and active waiting threads. Hence, these metrics cannot be used for Spin Wait detection, too.

2.1.4. Software Performance Antipatterns at a Glance

In this section, we described sixteen SPAs we found in literature. Ten of them we evaluated as feasible to detect by measurement-based approaches. We applied the categorization template to these ten SPAs and summarized the categorization results in Figure 2.6. So far, we considered each SPA individually. However, some antipatterns show dependencies among each other while others have several manifestations and different possible root causes. In order to illustrate these dependencies we depicted the hierarchy of performance problems in Figure 2.7.

	Observable Behaviour	Scope of Observation	High Level Indicator
The Blob	CB	messaging	high messaging overhead $p_{msg} = \frac{R'_{msg}}{R'} > T_{msg}$
Empty Semi Trucks			
Stifle	CB	database access	high database overhead $p_{db} = \frac{R'_{db}}{R'} > T_{db}$
The Ramp	RTPB, MCB	service execution	high variety in response times $COV(R) > T_{COV}$
Traffic Jam	RTPB		
One Lane Bridge	TB	concurrency, database access	
More is Less	TB	concurrency, database access	over linear growth of response times $R \propto X^a; a \gg 1$
Extensive processing	TB	concurrency	unbalanced resource utilizations $U(CPU_A) \gg U(CPU_B) \wedge QL(T) > T_{QL}$
Concurrent Processing Systems			
Dormant References	MCB	memory consumption	high memory consumption $mf = M(t_{after}) - M(t_{before}) > T_{mf}$
Session as a Data Store	MCB	memory consumption	over linear growth of response times $R \propto X^a; a \gg 1$

CB: Calling Behaviour
TB: Threading Behaviour

RTPB: Response Time Progression Behaviour
MCB: Memory Consumption Behaviour

Figure 2.6.: Overview on the categorization

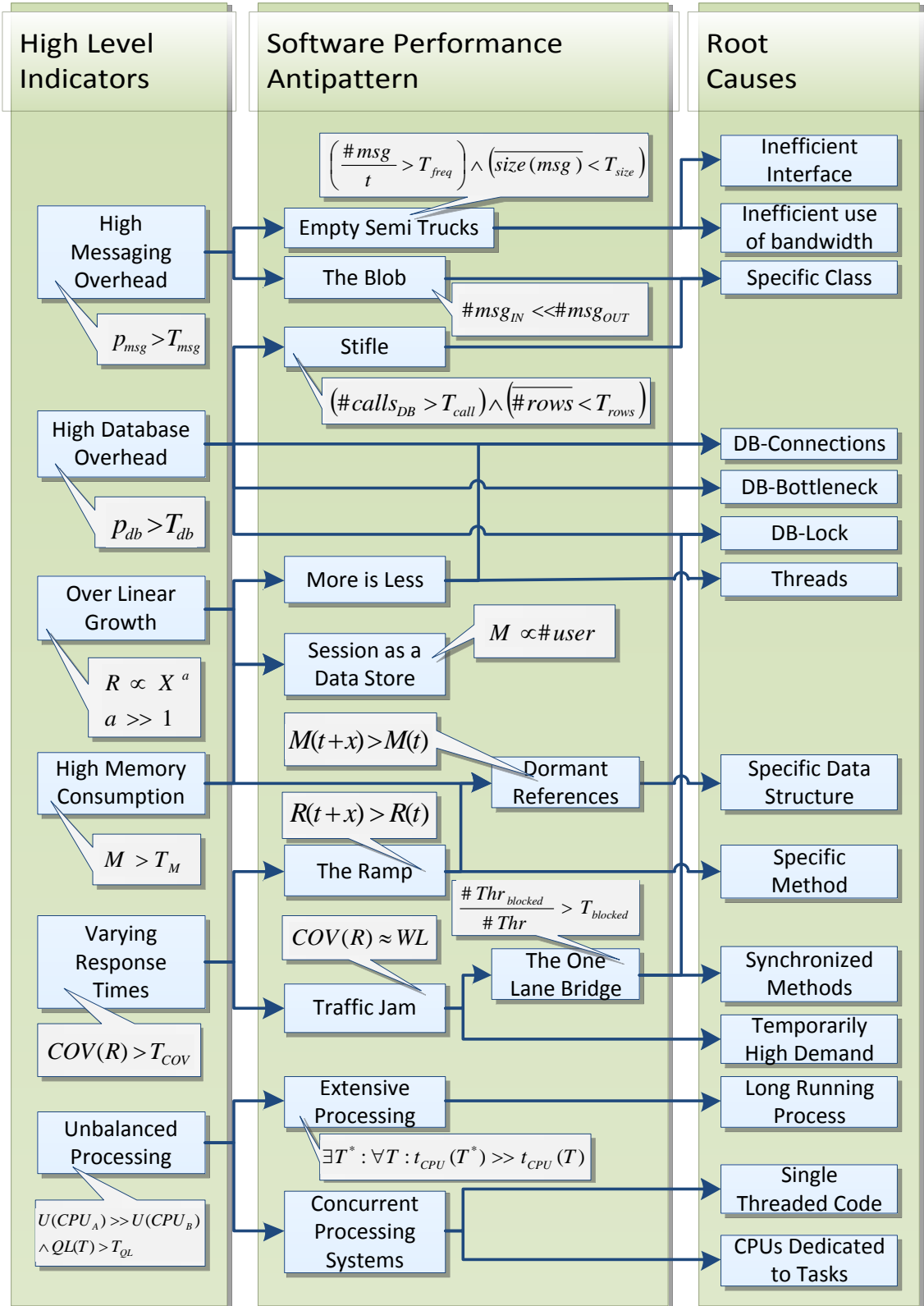


Figure 2.7.: Performance Problem Hierarchy

The left column embraces the high level indicators which are the most abstract performance problems. The causes for these problems are the single SPAs depicted in the middle column. SPAs are indicated by more specific indicators visualized as white speech bubbles in Figure 2.7. The arrows in Figure 2.7 illustrate a “possible cause” relationship. Finally, the right column embraces possible root causes for the individual SPAs.

2.2. Instrumentation

In this work, we develop a concept for detecting performance antipatterns based on systematical measurements. In order to make decisions on the existence of certain antipatterns, high level measurement data are insufficient. Therefore, we instrument the code of the software in order to gather more detailed and meaningful measurement data. In the following, we give an overview on some concepts, technologies and tools we will use for instrumenting code.

2.2.1. Aspect Oriented Programming

Aspect Oriented Programming (AOP) is a concept introduced by Kiczales et al. [KLM⁺97] for structuring software. While traditional structuring concepts like methods, classes, modules or components are proper for structuring basic functionality, they are not suitable for capturing issues concerning several modularization entities (*crosscutting concerns*). For instance, with traditional approaches enriching an application with logging information requires to modify each module, class or method. AOP provides means for encapsulating crosscutting concerns as aspects for the sake of modularity, and join these aspects with the functionality entities during execution of the application. Thus, the programmer is able to specify crosscutting concerns as one entity (aspect) instead of spreading this information among a large set of functionality entities. In this way, AOP increases the maintainability and clarity of application code when crosscutting concerns are required.

Several implementations of the AOP concept have been developed for different programming languages, including AspectJ [HH04], the AOP framework for Java. In general AOP comprises two components: a language for defining aspects and the so called Aspect Weaver responsible for joining aspects with executable code. The functionality of the Aspect Weaver is based on three important AOP concepts: *join points*, *pointcuts* and *advices*. Join points define points in the code where aspects can be woven in. In most AOP implementations method calls and field accesses are valid join points. Pointcuts are used by the Aspect Weaver for culling certain join points. Therefore, join points encountered by the Aspect Weaver are matched against pointcut definitions. For example, if the developer defines the following pointcut

Listing 2.1: Example for a pointcut

```
pointcut myPointcut(): call(void MyClass.do(int))
```

the Aspect Weaver picks out all method calls in the execution flow with the specified signature. For defining pointcuts, one can use wildcards - for instance, the following pointcut matches to all public methods returning an integer value:

Listing 2.2: Example for wildcards in a pointcut definition

```
pointcut anotherPointcut(): call(public int *.*(..))
```

Finally, the developer has to specify the behaviour of an aspect defining an advice. Advices extend the method concept, as they not only allow to describe a behaviour using a (high level) programming level (Java in the case of AspectJ), but also provide means

for defining when to execute the behaviour. There are three advice types differing in the execution time of the advice behaviour: before advice, after advice and around advice. As the names suggest, advices can be executed before, after or around a pointcut. Thus the advice in Listing 2.3 forces the Aspect Weaver to print the message “Calling a method!” before a method with the given signature is executed.

Listing 2.3: Example for an advice

```
before () : myPointcut () {
    System.out.println("Calling a method!");
}
```

AspectJ provides two ways of using the Aspect Weaver. Aspects can be woven into the code either at compilation time or at runtime. The first approach avoids overhead of weaving at runtime, but is less flexible as the target code has to be recompiled before using AOP. In AspectJ, dynamic weaving is realised by an agent which injects aspects into the bytecode when classes are loaded at runtime. This alternative is more flexible, but entails a higher runtime overhead for aspect weaving.

As code instrumentation is a crosscutting concern, in this work, we will use AOP and tools based on AOP (e.g. Kieker 2.2.4) for instrumenting code.

2.2.2. Structural Reflection and HotSwap

Standard AOP techniques like AspectJ are statical concepts, as code weaving is performed during class loading and loaded classes cannot be modified dynamically. The concepts *Structural Reflection* [Chi00] and *HotSwap* [CST03] allow to extend the AOP concept by the ability to dynamically changing behaviour of classes. In the following we describe these concepts and explain how they can be used to realize dynamic class modifications [CST03].

For introspecting data structures the Java language provides the reflection mechanism which allows to access class members, read class definitions or invoke certain class methods. However, the Java Reflection API does not allow modifying the behaviour or structure of classes. Techniques like AOP realize behavioural reflection, which allows to modify behaviour of certain operations through method call interception, but do not provide means to alter the class definition. In [Chi00] Chiba et. al introduce a framework called Javassist which supports structural reflection. This allows to modify data structure definitions, such as adding class members to a class definition. Therefore, Javassist extends the Java Reflection API and provides an implementation of structural reflection which does not depend on modifying existing runtime systems or compilers. Rather, Javassist realizes structural reflection by altering the bytecode of compiled Java classes before they are loaded into a Java Virtual Machine (JVM).

In a standard Java application Javassist can be used in two ways. The first alternative is to use a custom class loader for loading classes. Java allows for defining user class loader which extend the class *ClassLoader*. In order to apply structural reflection one could define and use a class loader similar to the example in Listing 2.4. Before a class X is actually loaded to the JVM by the methods *resolveClass()* and *defineClass()*, the defined class loader *CustomLoader* modifies the bytecode using Javassist. Note that all classes referenced from the class X are loaded by the same class loader on demand. The second possibility to use Javassist is to use it without a user class loader, but to overwrite the actual class file after the bytecode has been modified. Then, the standard class loader can be used for loading the stored class file.

For modifying class definitions Javassist provides meta-classes (*CtClass*, *CtMethod*, etc.) for each real class, method, etc., which represent the bytecode of the target class. The

Listing 2.4: Example for a custom class loader, (following [Chi00])

```

class CustomLoader extends ClassLoader {
    public Class loadClass(String name){
        byte[] bytecode = readClassFile(name);
        // modify class definition
        ...
        return resolveClass(defineClass(bytecode));
    }

    private byte[] readClassFile(String name){
        // read bytecode from class file
    }
}

```

meta-class provides methods for modifying the target class' definition by adding class members or altering method bodies. However, Javassist does not allow to remove class members as that would change the class type and violate the type safety principle of Java. For the sake of convenience, the modification methods provided by Javassist abstract from the actual bytecode, allowing the user to modify class definitions on a high abstraction level.

To sum up, Javassist enables users to conveniently modify class definitions before they are loaded into the JVM. However, an important restriction of Javassist is that classes cannot be modified anymore, as soon as they are loaded into the JVM. In order to overcome this limitation, in another work [CST03] Chiba et. al combine a concept called *HotSwap* with Javassist. Generally, Java does not allow a class to be loaded twice by the same class loader. Thus, the single way to use a dynamically modified class is to create a new custom class loader for loading the modified class. In Java classes are identified by the class identifier plus the class loader which loaded the class. Hence, the same class might exist several times in a Java program, if it is loaded by different class loaders. In Java Development Kit (JDK) 1.4 the HotSwap mechanism has been introduced, which is an extension of the Java Platform Debugger Architecture (JPDA) [jpd]. In debug mode, the HotSwap mechanism allows to dynamically reload classes as long as no schema changes on class definitions have been made. While method body modifications are allowed, adding or removing class members are not. Thus, as long as only method bodies are modified, using HotSwap allows us to dynamically change and reload classes without having to use a custom class loader.

In the context of this thesis, we will use Javassist and HotSwap for dynamically change instrumentation of classes (Chapter 5.2.2).

2.2.3. SIGAR API

The System Information Gatherer (SIGAR) is an API for retrieving general system information independent of the underlying operating system. SIGAR is provided by Hyperic [hyp], a subdivision of vmware [vmw]. SIGAR is used for retrieving general monitoring data such as memory consumption, cpu utilization, network information, etc. As every operating system provides this information, the main task of SIGAR is to provide a common, platform independent interface for accessing this information. Besides the platform independence, SIGAR provides interfaces for several programming languages like Java, C# or Perl.

Listing 2.5: Excerpt from `OperationExecutionAspectFull` aspect, (cf. [EHJ11])

```

aspect OperationExecutionAspectFull {
  pointcut allOperations() : execution(* *.*(..));

  around() : allOperations(){
    // measure, use Kieker framework
    ...
    proceed();
    ...
  }
}

```

2.2.4. Kieker

Kieker ([EHJ11], [kie]) is a framework for monitoring and analysing the runtime behaviour of complex software applications. In particular, Kieker is suitable for monitoring distributed software systems. Furthermore, the Kieker framework is quite flexible by providing several extension points for adapting the framework for custom use. The Kieker architecture comprises two main components: the monitoring component and the analysis component (cf. Figure 2.8). While the monitoring component is responsible for gathering

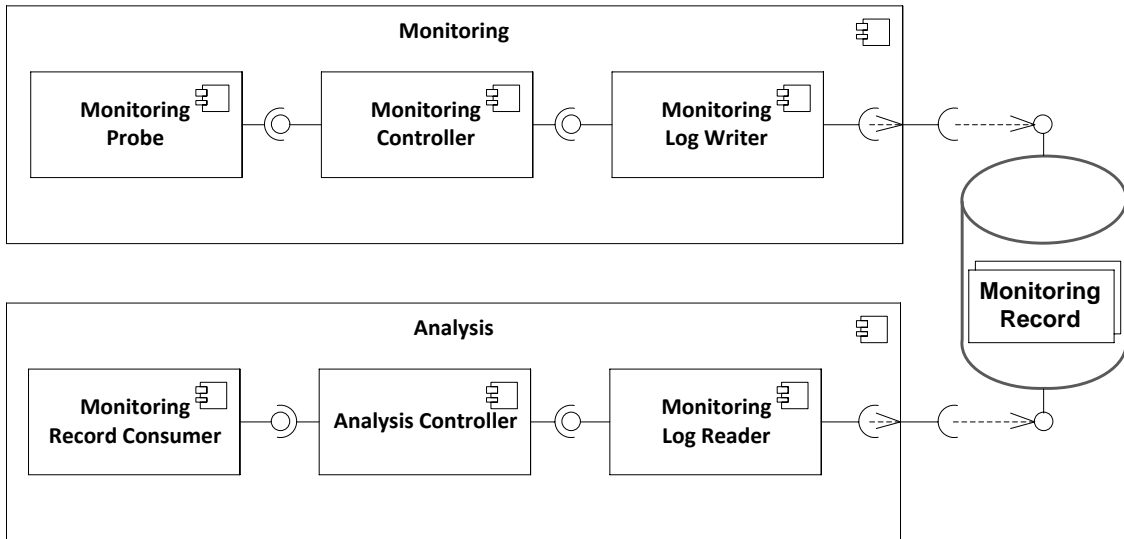


Figure 2.8.: Kieker architecture, following [kie]

monitoring data, the analysis component uses this data to perform different analyses. The monitoring component consists of three sub-components. Different types of monitoring probes are used to capture data at different points in the target system. For instance, probes might be SIGAR based providing general system information or they might use AOP for providing tracing information and reveal the internal behaviour. In fact, the monitoring probe component provides extension points allowing the user to implement custom monitoring probes. Kieker provides two types of monitoring probes which use AOP: *OperationExecutionAspectFull* and *OperationExecutionAspectAnnotation*. While the former intercepts every method call, the latter instruments only methods which are annotated with a certain annotation. The realization of the *OperationExecutionAspectFull* is illustrated in Listing 2.5. The `around` advice using the `allOperations` pointcut intercepts every method execution in order to monitor the executed methods. In contrast, the *annotate-*

Listing 2.6: Excerpt from `OperationExecutionAspectAnnotation` aspect, (cf. [EHJ11])

```

aspect OperationExecutionAspectAnnotation {
  pointcut annotatedOperations() :
    execution( @OperationExecutionMonitoringProbe * *.*(..) );

  around() : annotatedOperations() {
    // measure, use Kieker framework
    ...
    proceed();
    ...
  }
}

```

dOperations pointcut of the *OperationExecutionAspectAnnotation* aspect is restricted to methods which are annotated with *@OperationExecutionMonitoringProbe* (cf. Listing 2.6). Monitoring probes are coordinated by the monitoring controller which collects measured data from probes and passes it to the monitoring log writer. Again, there might be different log writer implementations for persisting monitoring data. For example, Kieker provides database writer, file system writer, in memory writer and java messaging (JMS) writer. Monitoring data is persisted by the writers as monitoring records which are created by the monitoring probes. The monitoring record concept is another extension point where the user can customize the usage of Kieker. In particular, one can define new records which, for example, are created by custom monitoring probes. Monitoring records are structured hierarchically, thus, each monitoring record must be derived from the most top record “AbstractMonitoringRecord” containing a logging timestamp.

The analysis component is structured symmetrically to the monitoring component. Thus, for each type of the monitoring log writer Kieker provides a monitoring log reader in order to read persisted monitoring records. The analysis controller implements a pipe and filter framework retrieving data from the readers and passing it from one analysis strategy to the next. Analysis strategies are implemented as plug-ins allowing the user to provide custom analysis strategies. Kieker provides analysis plug-ins for reconstructing the software architecture of the monitored system, performing trace analysis and visualizing the monitored and processed data, for instance in form of call graphs or dependency graphs. Call graphs and dependency graphs capture the calling behaviour between individual components of the monitored system and can be used to detect certain patterns. That is particularly interesting for the performance antipattern detection in this thesis.

2.3. Mathematical Foundations

In this section, we provide mathematical foundations required for the understanding of the concepts in this thesis. In the following, we focus only on certain mathematical aspects abstaining from profound explanations. Our descriptions are based on [HEK05], thus, for more comprehensive explanations we refer to [HEK05].

2.3.1. Central Tendency

In statistics there are different terms describing a central tendency of a sample (x_1, x_2, \dots, x_n) . In the following, we explain the *arithmetical mean* (also called *average*) and the *median*. For a set of values x_1, x_2, \dots, x_n the *arithmetical mean* \bar{x} is defined as:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i \quad (2.19)$$

While the *arithmetical mean* is susceptible to outliers another central value, the *median*, is more stable. The *median* is a special case of a *quantil*. For a sorted set of values x_1, x_2, \dots, x_n , the α -quantil \tilde{x}_α is a value for which $\alpha \cdot 100\%$ of values are smaller or equal \tilde{x}_α :

$$\tilde{x}_\alpha = \begin{cases} x_{\lceil n \cdot \alpha \rceil} & , n \cdot \alpha \notin \mathbb{Z} \\ \frac{1}{2}(x_{n \cdot \alpha} + x_{n \cdot \alpha + 1}) & , n \cdot \alpha \in \mathbb{Z} \end{cases} \quad x_1 \leq x_2 \leq \dots \leq x_n \quad (2.20)$$

The *median* \tilde{x} is a 0.5 – *quantil*:

$$\tilde{x} = \begin{cases} x_{\lceil \frac{n}{2} \rceil} & , n \text{ odd} \\ \frac{1}{2}(x_{\frac{n}{2}} + x_{\frac{n}{2} + 1}) & , n \text{ even} \end{cases} ; x_1 \leq x_2 \leq \dots \leq x_n \quad (2.21)$$

2.3.2. Measures of Dispersion

For the evaluation of samples, the dispersion is an important aspect. In evaluating statistics there are different measures for describing dispersion. In the following we describe only the *variance*, the *standard deviation* and the *coefficient of variance*. Further measures are described in [HEK05].

One of the most used measures for describing dispersion is the *variance*. The *variance* s^2 of a sample $X = (x_1, \dots, x_n)$ is defined as follows:

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2 \quad (2.22)$$

The square root s of the variance is called *standard deviation*. An important property of the *standard deviation* is the fact that it has the same dimension as the arithmetical mean \bar{x} of the sample. The *standard deviation* is defined as follows:

$$s = \sqrt{s^2} = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2} \quad (2.23)$$

The *coefficient of variance* (*COV*) is a dimensionless measure, describing the dispersion of a sample. If \bar{x} is the mean and s^2 is the variance of the sample $X = (x_1, \dots, x_n)$, the *coefficient of variance* for sample X is defined as follows:

$$COV = \frac{\sqrt{s^2}}{\bar{x}} = \frac{\sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2}}{\frac{1}{n} \sum_{i=1}^n x_i} \quad (2.24)$$

2.3.3. Comparing two Samples

Often, it is necessary to compare two independent samples. For this purpose, statistical tests can be used. In this section, we describe the nature of *confidence intervals* and the *t-test* which is a statistical test for comparing the mean values of two independent samples.

2.3.3.1. Confidence Interval

If X_1, \dots, X_n is a sample from a normal distributed random variable $X \in N(\mu, \sigma^2)$, the arithmetical mean \bar{x} is a point estimate for the expected value μ . A *confidence interval* (CI) $[c_1, c_2]$ is an interval estimate for μ :

$$P(c_1 \leq \mu \leq c_2) = 1 - \alpha \quad (2.25)$$

The probability $1 - \alpha$ that μ lies within the confidence interval is called *confidence level*. α is called significance level. If X_1, \dots, X_n are identically, normally distributed ($X_i \in N(\mu, \sigma^2)$), the mean

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i \quad (2.26)$$

is normally distributed, too:

$$\bar{X} \in N\left(\mu, \frac{\sigma^2}{n}\right) \quad (2.27)$$

The transformation $Z = \frac{\bar{X} - \mu}{\frac{\sigma^2}{n}} \in N(0, 1)$ follows a standard normal distribution. For a random variable Z following a standard normal distribution and a significance level α the critical value $v_{1-\frac{\alpha}{2}}$ is known, such that:

$$P(-v_{1-\frac{\alpha}{2}} \leq Z \leq v_{1-\frac{\alpha}{2}}) = 1 - \alpha \quad (2.28)$$

Substituting Z and solving the equation for μ yields the following:

$$P(c_1 = \bar{X} - v_{1-\frac{\alpha}{2}} \cdot \sqrt{\frac{\sigma^2}{n}} \leq \mu \leq \bar{X} + v_{1-\frac{\alpha}{2}} \cdot \sqrt{\frac{\sigma^2}{n}} = c_2) = 1 - \alpha \quad (2.29)$$

If $n \geq 30$, we can estimate σ^2 with the empirical variance s^2 . Thus, we get the confidence interval $[c_1, c_2]$ with:

$$c_1 = \bar{X} - v_{1-\frac{\alpha}{2}} \cdot \sqrt{\frac{s^2}{n}} \quad c_2 = \bar{X} + v_{1-\frac{\alpha}{2}} \cdot \sqrt{\frac{s^2}{n}} \quad (2.30)$$

If $n < 30$, σ^2 cannot be estimated accurately with s^2 . However, if $\forall i : X_i \in N(\mu, \sigma^2)$ then the transformation

$$T = \frac{\bar{X} - \mu}{\frac{s^2}{n}} \in N(0, 1) \quad (2.31)$$

is *student-t* distributed with $n - 1$ degrees of freedom. In this case, the confidence interval $[c_1, c_2]$ is:

$$c_1 = \bar{X} - t_{1-\frac{\alpha}{2}, n-1} \cdot \sqrt{\frac{s^2}{n}} \quad c_2 = \bar{X} + t_{1-\frac{\alpha}{2}, n-1} \cdot \sqrt{\frac{s^2}{n}} \quad (2.32)$$

Whereby, $t_{1-\frac{\alpha}{2}, n-1}$ is the critical value of the t-distribution for significance level α and $n - 1$ degrees of freedom.

2.3.3.2. T-Test

The *t-test* utilizes the confidence interval in order to compare two independently normally distributed samples $X = (x_1, \dots, x_n)$ and $Y = (y_1, \dots, y_m)$. In particular, the t-test checks whether X and Y originate from the same population. If two samples originate from the same population their expected values μ_X and μ_Y are equal. The t-test checks the following *null hypothesis* H_0 :

$$H_0 : \mu_X - \mu_Y = 0 \quad (2.33)$$

The alternative hypothesis is:

$$H_1 : \mu_X - \mu_Y \neq 0 \quad (2.34)$$

Assuming that X and Y are normally distributed, the means \bar{X} , \bar{Y} and the difference $\bar{D} = \bar{X} - \bar{Y}$ are normally distributed as well:

$$\bar{X} \in N(\mu_X, \frac{\sigma_X^2}{n}) \quad \bar{Y} \in N(\mu_Y, \frac{\sigma_Y^2}{m}) \quad \bar{D} \in N(\mu_D, \sigma_D^2) \quad \mu_D = \mu_X - \mu_Y \quad (2.35)$$

For a confidence level α the confidence interval $[c_1, c_2]$ for \bar{D} can be calculated. If $0 \notin [c_1, c_2]$ then $\mu_X - \mu_Y \neq 0$ with a probability of $p = (1 - \alpha) \cdot 100\%$. In this case the null hypothesis H_0 can be rejected and the alternative hypothesis H_1 applies with a probability p .

2.3.4. Linear Regression

For a two-dimensional sample $S = ((x_1, y_1), \dots, (x_n, y_n))$, it is often desirable to find a functional dependency between the two dimensions. Regression techniques provide means for deriving this functional dependency. For our purposes in this thesis, we use *linear regression* which provides a function $f(x) = y$ describing a linear dependency between x and y . Because of measurement errors, the single points of sample S are not located on a straight. Thus, for each point (x_i, y_i) we assume the following dependency:

$$y_i = f(x_i) + e_i \quad (2.36)$$

Here, f is the regression line and e_i the measurement error for the point (x_i, y_i) . We want to find a regression line $y = f(x) = a + bx$ which minimizes the errors $e_i = y_i - f(x_i)$ over all sample points (x_i, y_i) . For this purpose, one can use the *least squares method*. Thus, our goal is to find values for a and b such that the sum of squared errors SSE is minimized:

$$SSE(a, b) = \sum_{i=1}^n e_i^2 = \sum_{i=1}^n (y_i - a - bx_i)^2 \xrightarrow{!} \min \quad (2.37)$$

In order to find values a and b minimizing $SSE(a, b)$, we have to determine the partial derivatives for $SSE(a, b)$ and equate them to zero:

$$\begin{aligned} \frac{\partial SSE(a, b)}{\partial a} &= -2 \sum_{i=1}^n (y_i - a - bx_i) \stackrel{!}{=} 0 \\ \frac{\partial SSE(a, b)}{\partial b} &= -2 \sum_{i=1}^n (y_i - a - bx_i) \cdot x_i \stackrel{!}{=} 0 \end{aligned} \quad (2.38)$$

These two terms define a system of equations. The values for a and b can be determined by solving this system of equations.

2.4. Foundations on Performance Evaluation

In this section, we introduce some foundations on software performance evaluation which are important to understand the concepts described in this thesis. In general, there are two different approaches for performance evaluation: model-based and measurement-based approaches. In the following, we describe some aspects of both approaches. The descriptions in this section are based on [Lil00]. For further reading and deeper understanding we recommend [Lil00].

2.4.1. Performance Modeling

The advantage of model-based approaches is that it can be applied in early development phases when no runnable artifacts are available. However, performance models have to be created which are used for evaluation. A wide spread approach for modeling software performance is based on the *Queueing Theory*, which is fragmentary described in the following.

Queueing Networks

An established way to model performance aspects of software systems is to use *queueing networks*. For this purpose, resources (processors, disks, network connections, etc.) are modeled as *queues*. A queue consists of a *waiting line* and one or more *servers*. Servers are intended to process requests. Each server can process only one system request at a time. Other requests acquiring the same server have to wait in the *waiting line* of the corresponding queue. Servers and queues can form a complex network whose connectors describe the transition probabilities from one server to another. Figure 2.9 depicts an exemplary queueing network for an environment comprising a load balancer, a set of application servers and a database.

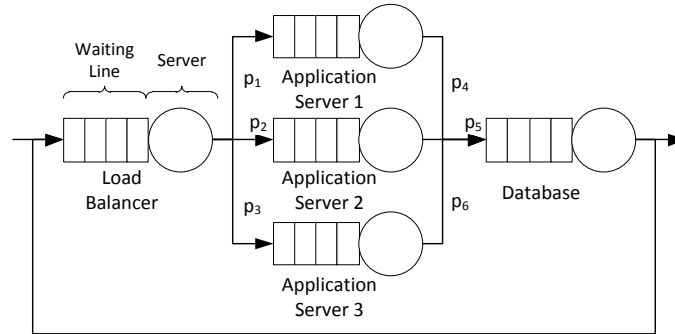


Figure 2.9.: Example for a queueing network

In the following, we list some important definitions concerning the description of queueing networks:

- The *arrival rate* λ is the number of requests which arrive per unit of time at the system or a single queue.
- The *throughput* X is the number of requests which are completed per unit of time.
- The *service time* S is the average time a server needs to service a request.
- The *response time* R is the service time plus the time the corresponding requests is waiting in the queue.
- A system is in *steady state* if considered over a long period of time the following applies: $X \approx \lambda$. For a system which is not in a steady state, the arrival rate is higher than the throughput. In this case, the system is overloaded and the queue length grows infinitely.
- An *active resource* has a finite speed and processes requests. CPUs, hard disks, network connections, etc. are active resources.
- A *passive resource* does not perform any work itself, however it is required for the execution of a request. Typical passive resources are software threads, connection pools, software locks, etc.

Workload

Queuing networks can be solved either analytically or by simulation. Both approaches require a specification of the *workload* for the examined system. A workload can comprise one or more *workload classes*. A workload class represents a certain group of system requests. Each workload class is specified by a *workload type*, a *workload intensity* and for each queue in the system a *service demand*. There are two possible types of workload: *open workload* and *closed workload*. The workload intensity of an open workload is specified by the arrival rate λ . Open workloads can bring the system to an unsteady state if the arrival rate is too high. If the arrival rate is permanently higher than the throughput of the queueing network, the queue length grows steadily bringing the system in an unsteady state. Thus, with an open workload the number of concurrent requests in the system is not limited. A closed workload is specified by a fixed population size. The workload intensity is determined by a number of concurrent users and a *think time*. The think time describes for each user how long the user waits after completion of a request before he places a new request. Systems under a closed workload are inherently in a steady-state as the system population is limited. Thus, with a closed workload the arrival rate λ equals the throughput X .

Little's Law

Operational Analysis is a discipline which describes the relationships between certain metrics of a system in operation. Operational Analysis comprises some laws which describe these relationships. *Little's Law* is one of the most important laws. This law describes the relationship between throughput X , system population N and the response time R of any system which is in a steady state. This law can be applied without knowing the internals of a system. If for any system in a steady state the mean throughput is X and the mean response time of requests is R , then the following applies for the mean number of concurrent requests in the system:

$$N = R \cdot X \quad (2.39)$$

2.4.2. Measurement-Based Performance Evaluation

As model-based performance evaluation is based on abstractions, in general, model-based performance evaluation approaches are less accurate than measurement-based approaches. Measurement experiments are the core of measurement-based performance evaluation approaches. Data captured during measurements is used to derive *performance metrics* which serve for evaluation of the performance behaviour of the examined software systems. Performance metrics depend on input parameters which describe the system state and usage behaviour of the system under test during measurements. Let $P = (P_1, \dots, P_n)$ be the set of input parameters describing the configuration of the system and the usage behaviour. An *experiment* is defined by a configuration vector $V^i = (V_1^i, \dots, V_n^i)$ of values for P and a set $Q = (Q_1, \dots, Q_k)$ of parameters for which observations are taken. In order to increase the accuracy of measurements, experiments should be repeated several times. An *experiment series* is a set of m experiments with different configurations V^1, \dots, V^m for P capturing data for the same observation parameters Q_1, \dots, Q_k . Thus, the result of an experiment series is a discrete relation $r(P)$ describing the dependency between input values for P and observation values for Q . Often, regression techniques are used to derive a functional dependency (called Performance Curve [WHW12]) between P and Q . Measurement results can be used for many different purposes like bottleneck analysis, capacity planning, performance problem detection and others.

2.5. Software Performance Cockpit

Similar to the Kieker framework (cf. Section 2.2.4), the *Software Performance Cockpit* (SoPeCo) [WHHH10] is an approach and tool for software performance evaluation. Unlike the Kieker framework which is used for monitoring software systems during operation, SoPeCo is intended for conducting systematic measurement experiments in order to capture the performance behaviour of a software system using goal-oriented measurements. In particular, SoPeCo can be used for evaluating the performance of distributed systems. Originally, SoPeCo was designed for measuring the performance of target systems in order to infer mathematical models (*Performance Curves* [HWSK10], [WHW12]) of the system's performance behaviour. However, currently, extensions for SoPeCo are being developed, e.g., that execute and analyse JUnit performance regressions [Heg12] or antipattern detection in the context of this thesis. As the SoPeCo architecture allows for deploying the main measurement control part on a dedicated *Measurement Control Node* (cf. Figure 2.10), the SoPeCo influences the performance of the target system only to a little degree. In Figure 2.10, the architecture of the Software Performance Cockpit is depicted. In general,

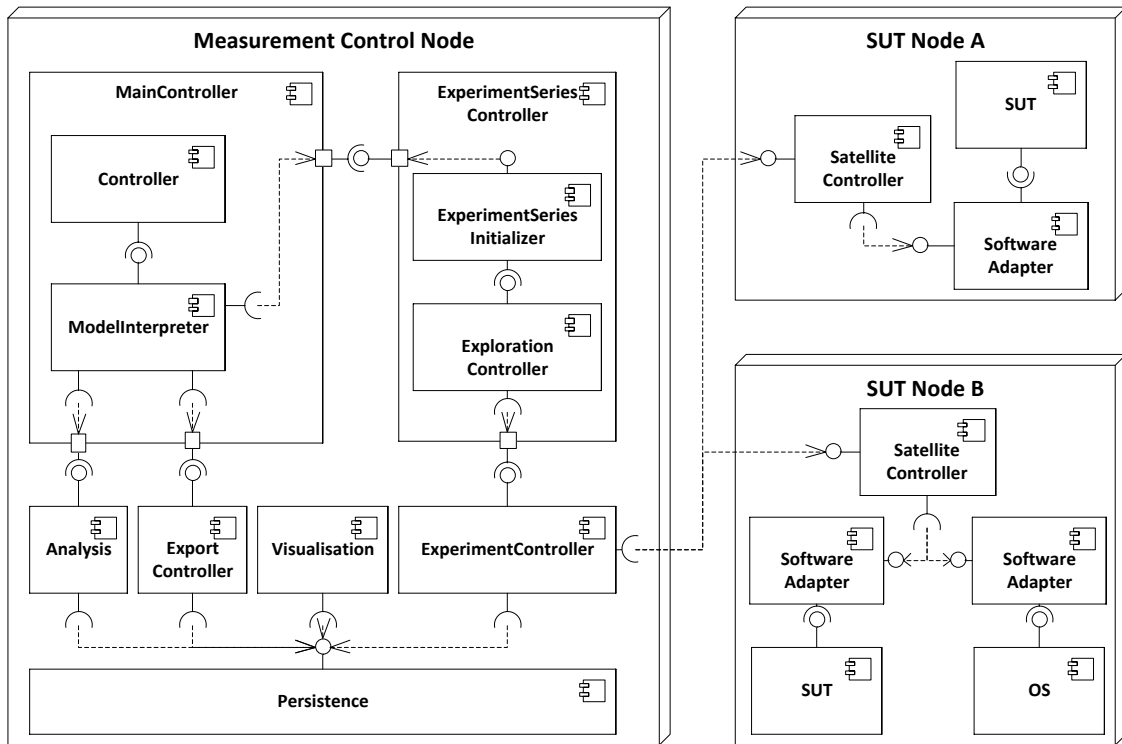


Figure 2.10.: Architecture of the Software Performance Cockpit

the SoPeCo system can be distributed among one *Measurement Control Node* and several system under test (SUT) nodes. While the main part of the SoPeCo framework can be deployed on a dedicated node, one needs to deploy lightweight satellites on SUT nodes in order to capture measurement data on these nodes. The *MainController* is responsible for coordinating the overall process. Furthermore, the *MainController* sets up the measurement environment by interpreting the measurement configuration model provided by the performance analyst. The measurement configuration model allows the performance analyst to specify the resource environment, define experiments and experiment series, select strategies for exploring the parameter space as well as analysis and export strategies. In the first step, the *MainController* utilizes the *ExperimentSeriesController* for coordinating the execution of all experiment series. The *ExperimentSeriesController* configures the measurement environment, prepares all measurement components for experiment execu-

tion and finally uses the *ExplorationController* for exploring the specified parameter space triggering single experiments. As the SUT might be distributed among several nodes, each node contains a *SatelliteController* instance responsible for gathering measurement data on the according node. The *ExperimentSeriesController* utilizes the *ExperimentController* for conducting single experiments. Therefore, the *ExperimentController* coordinates the interaction between individual satellites and correlates measurement data provided by the satellites after experiment execution. Finally, measurement data is passed to the *Persistence* component responsible for storing data and making it available for further processing. Considering the SUT nodes (cf. Figure 2.10), one can see that a *SatelliteController* uses *SoftwareAdapters* for accessing the actual SUT. As *SoftwareAdapters* are software components responsible for invoking the SUT and gathering certain performance metrics, they have to be provided by the performance analyst (or domain expert). In general, one can define several *SoftwareAdapters* per *SatelliteController*, each responsible for collecting metrics concerning different performance aspects. For example, consider the *SUT Node B* in Figure 2.10. There, the *SatelliteController* comprises two *SoftwareAdapters* gathering different measurement data. While the left *SoftwareAdapter* instruments the target application (SUT) measuring metrics like response time, the right *SoftwareAdapter* accesses the operating system (OS) in order to retrieve general metrics like resource utilization.

Besides the measurement execution, another important step in the SoPeCo overall process is the analysis phase. The *Analysis* component serves for inferring *Performance Curves* from measurement data. Therefore, SoPeCo provides a set of different analysis strategies, for instance *Linear Regression*, *MARS* [CW00], *Kriging* [OW90] or *Genetic Programming* [FH12]. Based on measurement data retrieved from the Persistence Layer, the Analysis component calculates a curve. For exporting measurement data or inferred curves, the *MainController* utilizes the *ExportController* which allows for generating CSV or image files. Furthermore, the performance analyst can use the *Visualisation* component for interactive analysis of measured data.

In the scope of this thesis, we will extend the Software Performance Cockpit concept by the ability to detect performance antipatterns in target software systems.

3. Related Work And Contribution

In this chapter, we give an overview of recent work on performance engineering, design patterns and antipatterns, as well as antipattern detection techniques. In the second part, we describe the contribution of this thesis and differentiate our concepts from related work.

3.1. Related Work

Koziolok [Koz10] provides a survey on software performance evaluation for component-based software systems. His survey comprises both model-based as well as measurement-based performance evaluation approaches. Some research have been made in modeling software systems based on observation of measurement data ([HWSK10], [WVCB01], [PVR95], [SKK⁺01] and [XH96]).

While software patterns have been investigated for many years ([GHJV93], [YB98], [AZ05], [Mes96], [PS97], [RCS08], [BHS07]) research in the field of antipatterns is relatively young. Software antipatterns have been defined for different contexts. Hallal et al. [HAT⁺04] define antipatterns concerning Java-based multi-thread systems. Information security antipatterns are introduced in [Kis02]. Dudney et al. [DAKW03] describe technology specific antipatterns related to J2EE systems.

Smith and Williams ([SW00], [SW02b], [SW03b] and [SW03a]) define and describe a set of performance antipatterns which are independent of any technology. In their articles, the authors describe 10 antipatterns in detail, explain the symptoms and propose solutions for each antipattern. Much of later research articles in the context of technology independent performance antipatterns are based on the work of Smith and Williams. Smaalders [Sma06] describes some recurring performance problems (antipatterns) experienced at Sun Microsystems during the refactoring process of the Solaris operation system.

Patterns and antipatterns definitions themselves are of little use as long as they are not used for designing and analysing software architectures. While design patterns can be used relatively straight forward during the software design process, antipatterns serve as a feedback concept indicating flaws in present design models or implementations. Thus, antipattern detection is crucial for practical use of antipatterns. However, detecting patterns in existing software fragments might be of high use, too, especially in the context of re-engineering and refactoring legacy systems. Heuzeroth et al. [HHHL03] apply static and dynamic code analysis in order to detect patterns in legacy code. For static analysis, an attributed abstract syntax tree (AST) is generated. Heuzeroth et al. define patterns as relations over the AST and compare the AST with a list of relations in order to identify a set of pattern candidates. In the second step, the candidates are monitored during

execution checking for dynamic pattern rule violations. Candidates, which do not violate pattern rules represent detected patterns. In a similar way, Antoniol et al. [AFC98] apply statical analysis to ASTs in order to detect patterns in object oriented software.

While many patterns can be detected by statical structure analysis, discovering performance antipatterns is mostly based on detecting symptoms characterizing certain antipatterns. Cortellessa et al. [CMR10] introduce a process for identifying antipatterns which actually negatively affect the performance of the considered system. Therefore, detected antipatterns are compared with violated requirements in order to select 'guilty' antipatterns. As only 'guilty' antipatterns are detected, this process reduces the effort for refactoring the design. In another work, Cortellessa et al. [CME10] introduce a concept for UML-based detection of technology independent performance antipatterns. For this purpose, the UML software design model is annotated with performance properties using the UML MARTE profile. The next step is a transformation of the annotated UML model to a performance model in form of a Queueing Network (QN). The QN is then solved using standard analysis methods like Mean Value Analysis (MVA) in order to derive service quality (QoS) attributes like response time, utilization and throughput. Cortellessa et al. formalize antipatterns as a set of OCL rules encoding antipattern symptoms as formal OCL conditions. The computed QoS attributes are checked against the OCL conditions in order to detect antipatterns. Xu [Xu09] uses Layered Queueing Networks (LQN) as performance models in order to differentiate between design flaws and configuration (resource allocation) flaws during antipattern detection. However, according to Cortellessa et al. [CME10], using LQNs as performance models restricts the amount of possible refactoring solutions during the transformation of performance attributes back to the design model. While the concept of Cortellessa et al. is based on UML models, Trubiani and Koziolok [TK11] describe a method for detecting technology independent performance antipatterns in software system models specified using the Palladio modelling language. Trubiani and Koziolok define a set of rules characterizing different antipatterns. Similar to the concepts of Cortellessa et al. and Xu, the Palladio software design model is checked statically against the defined rules for antipattern detection. The Palladio modelling language provides modeling elements for performance properties, allowing the user to consider the system design from a performance perspective.

The antipattern detection concepts of Cortellessa et al., Xu and Trubiani et al. are (partly) based on software design models. Parsons et al. [PM08] introduce a measurement-based antipattern detection concept. However, their concept is restricted to component based Java Enterprise Edition (JEE) software systems. Parsons et al. introduce the Performance Antipattern Detection (PAD) tool comprising three components. The Monitoring component is responsible for capturing and gathering run-time data, such as run-time paths, resource info and component meta-data. This data is then used by the Advanced Data Analysis component to reconstruct a run-time design model. Using a set of antipattern rules the Rule Engine component analyses the run-time design model in order to detect JEE specific performance antipatterns. Compared to the concept proposed in this thesis, the approach of Parsons et al. is based on monitoring of systems in operation rather than systematic measurement experiments.

In the mid-nineties Miller et al. [MCC⁺95] introduced a tool named *Paradyn*. Instead of detecting performance antipatterns, Paradyn searches fully automated for general performance problems. In particular, Paradyn focuses on evaluating parallel and distributed, long-running software. In order to realize fully automated problem detection, Miller et al. apply two essential concepts. The first is a dynamic instrumentation concept allowing to modify the instrumentation during execution time. Secondly, for guidance of the Paradyn process Miller et al. define a hierarchical model called *W³ Search Model*. The *W³ Search Model* comprises three dimensions: *why*, *where* and *when*. The *why* dimension addresses the problem of detecting the existence of a performance problem. The *where* dimension

is used to locate the resource where the performance problem occurs. Finally, the *when* dimension is used to constrict the time interval of the problem’s occurrence. Both, the *why* and the *where* dimensions exhibit a hierarchical structure. Searching along the *why* dimension is done by traversing a hypothesis hierarchy. Whereby, the most top hypothesis represents the most general reason for the considered performance problem, becoming more specific when digging deeper into the hierarchy. For each hypothesis Miller et al. define tests used to evaluate measurement data against the hypotheses. The *where* dimension is structured in a similar hierarchical way. Miller et al. organize resources (system nodes, synchronization objects, disks, CPUs, etc.) in a hierarchical structure. Isolating performance considerations to certain groups of resources allows to find the actual resource where the problem occurs. When traversing these dimension Paradyn adapts the instrumentation of the target system in order to collect only data required for the current performance considerations. As Miller et al. use dynamic instrumentation, the Paradyn tool provides an elegant concept for automating the search for performance problems. However, the Paradyn tool is based on monitoring rather than executing systematic measurement experiments. Therefore, the Paradyn tool is limited by the assumption that the target system is running long enough to observe all relevant performance cases.

3.2. Contribution

In this section, we explain the contribution of this thesis and differentiate it from the related work described in the previous section.

In this thesis, we introduce a measurement-based detection approach for software performance antipatterns. Based on the idea of systematic measurement experiments, we develop an adaptive measurement approach which reduces the monitoring overhead and increases the efficiency of the detection process. For some selected antipatterns we investigate different experiment execution and data analysis techniques in order to find best suited concepts to detect the considered antipatterns. Finally, we evaluate the described approach on an official benchmark scenario showing the effectiveness of the described approach.

While in [HHHL03] and [AFC98] common design patterns are extracted from existing systems, we detect performance related antipatterns. When performance antipatterns are detected through statical code or model analysis ([CME10] and [TK11]) there is no guarantee that the detected antipatterns actually impair the performance. Therefore, concepts are needed to rank detected antipatterns and find “guilty” antipatterns as proposed in [CMR10]. Moreover, it is difficult to discover performance antipatterns through statical analysis as performance is a dynamic property of a software system. As our approach is based on measurements, the performance antipatterns detected by our approach inherently impair the performance of the examined software system. Thus, the probability to find false positives (“non-guilty” performance antipatterns) is quite low. Although [PM08] and [MCC⁺95] are measurement-based approaches, they utilize monitoring for gathering measurement data. Thus, the amount of detected antipatterns depends on the actual workload. As we apply systematic measurement experiments, we are able to control the search process and the workload submitted to the examined software system during measurements. In this way, antipatterns can be detected more effectively. While it is difficult to derive refactoring solutions when detecting performance antipatterns based on performance models ([Xu09]), through systematic measurement experiments it is possible to search for the actual root causes of detected performance problems.

4. Approach

In this chapter, we describe our systematic, measurement-based approach for automatically detecting software performance antipatterns during development. For general understanding, in Section 4.1, we first present the general idea and motivation for automatically detecting performance antipatterns during the development phase. In particular, we illustrate the benefits for the software development process and software quality. In Section 4.2, we consider different measurement approaches. Extending and applying the *Dynamic Instrumentation* approach from [MCC⁺95], we develop and describe the *Adaptive Measurement* approach. Moreover, we derive an overall process model from the adaptive measurement approach. In Section 4.3, we integrate this process with the *Antipattern Detection Architecture*, a software architecture based on the SoPeCo framework (cf. Section 2.5). Finally, in Section 4.4, we meet some assumptions which serve as little simplifications for the realization of the proposed detection concept.

4.1. Big Picture On The Antipattern Detection Approach

In the following, we present the idea behind our antipattern detection approach. In contrast to many other approaches working on system and performance models, with our approach we focus on detecting performance antipatterns in existing software systems. This includes, in particular, software systems under development. In this way, our approach aims to support developers and architects to build high quality software with regard to software performance. In particular, the main goal of our antipattern detection approach is to provide valuable performance feedback to developers and architects indicating performance problems introduced during development or designing. As refactoring and restructuring software can become expensive and time-consuming tasks if they are dealt with lately, it is important to provide feedback as early as possible. Thus, instead of monitoring a fully developed software system during operation, our approach is based on systematic measurement experiments which are conducted periodically during development. If developers and architects are informed about performance problems and their causes during an early development phase, they are able to fix that problems with little effort and low costs. Due to its complexity and effort, performance evaluation is omitted in many development projects. The automation of performance evaluation tasks and, in particular, antipattern detection can overcome this problem. Following this idea, the SoPeCo framework (cf. Section 2.5) provides means for performing systematic experiments on (distributed) software systems and enables the automated execution of measurements. Based on SoPeCo, we

develop analyses of measurement data, which automatically decide whether certain antipatterns are present in the system under test.

In Figure 4.1, we depicted the antipattern detection process integrated with the software development task. The result of each development iteration is a partly completed software

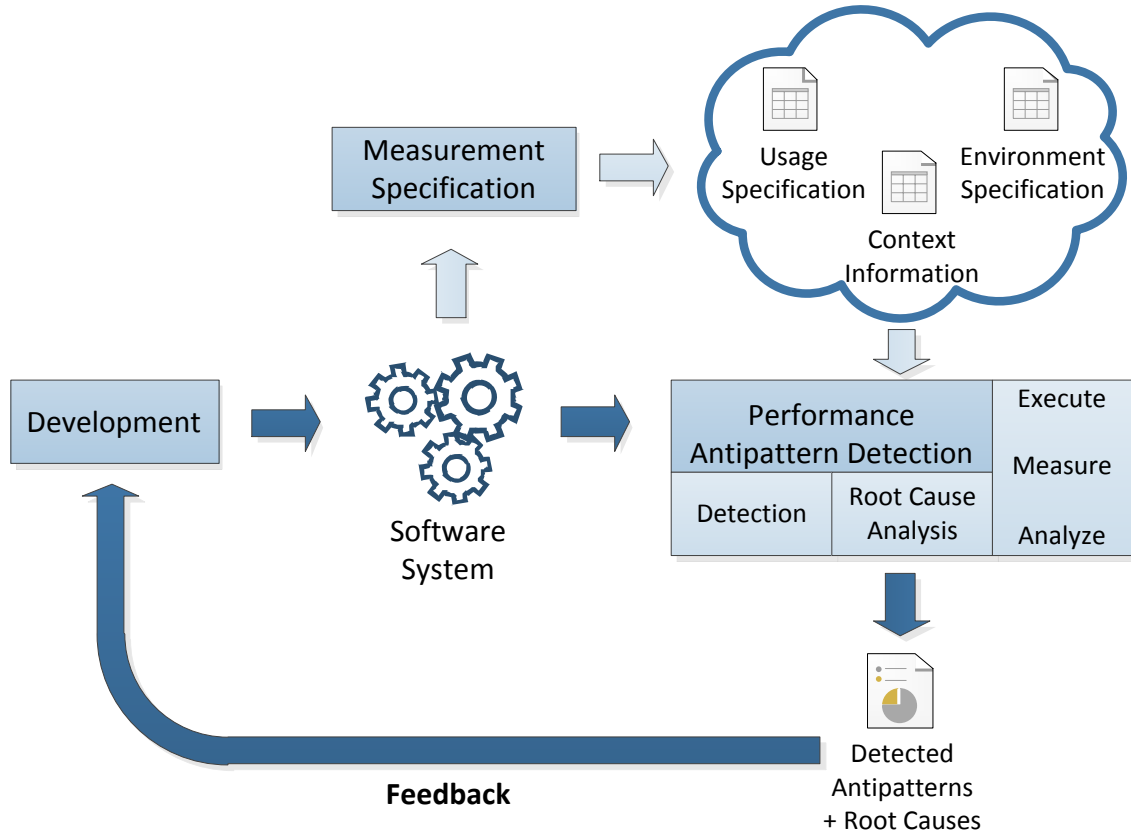


Figure 4.1.: Antipattern detection integrated with development

system which can be evaluated with regard to performance. For this purpose, the developer has to provide measurement specifications. In particular, these are an environment specification describing the distribution of the software system under test (SUT), an usage specification describing the typical workload for the SUT, and finally, additional context information required for analysis. Then, these specifications are used to perform antipattern detection. While the tasks *Development* and providing *Measurement Specifications* are conducted by the developer himself, the antipattern detection is a fully automated task which can be executed for example during a nightly build. Performance antipattern detection includes gathering data about performance and dynamic behaviour of the SUT while it is executed, and analyzing this data in order to detect possible performance antipatterns. The result of the performance antipattern detection task serves as performance feedback to the developer. For a valuable feedback two things are important. Firstly, the performance antipattern detection should provide hints on which performance antipatterns occur. Secondly, the performance feedback can provide additional value by indicating the root causes for each antipattern. Thus, the antipattern detection task is logically divided into two main sub-tasks: detecting the occurrence of certain antipatterns and finding the root causes for these antipatterns. Providing this information to the developer, the automatic antipattern detection task decreases the required effort for the developer to find performance lacks.

4.2. The Adaptive Measurement Approach

As suggested in the previous section, detecting SPAs and their root causes includes measuring and analyzing measurement data. While analysis approaches have to be designed individually for each antipattern, defining the measurement approach is a rather generic task. In general, two different measurement approaches are possible.

We call the first alternative *general monitoring*, which is a quite general measurement approach. Applying general monitoring means gathering all data of interest while the system is set under a random or statistically specified workload. A statistically specified workload should simulate the real usage behaviour of the target system. Applying this measurement approach for software performance antipattern (SPA) detection implicates that all data required for detecting all SPAs has to be gathered during one measurement run. The simplicity of gathering data is an advantage of this approach as no filtering or preprocessing is required. Furthermore, the general monitoring approach implies a quite simple overall process. As all required data is gathered during one measurement run, this measurement approach allows to divide the antipattern detection task into two separated sub-tasks: *measure* and *analyze*. The resulting overall process is depicted in Figure 4.2. It is a sequential process consisting of three parts. Firstly, the system under test is fully instrumented in order to gather all required data. The second part comprises execution and monitoring of the system under test. Finally, the measured data is analyzed in order to detect performance antipatterns. However, gathering all required data for all SPAs

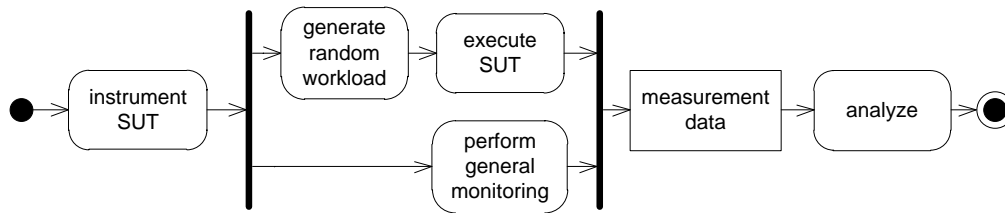


Figure 4.2.: Overall process applying general monitoring approach

results in a huge amount of unfiltered data. This circumstance complicates the analyses of measured data. Besides the huge amount of data, a high measurement overhead distorting the measurement results (cf. Section 5.1) is the second disadvantage of this approach. High overhead is caused by excessive data gathering implicated by the general monitoring approach. While the former problem affects only the performance of the analysis phase, the second disadvantage of this measurement approach affects the accuracy of the analysis results. Due to these disadvantages this measurement approach is not applicable for SPA detection.

The second alternative, the *adaptive measurement approach*, is based on systematic measurements. In order to overcome the problems of the first approach we execute target-oriented measurements for each aspect we want to examine. With target-oriented measurement experiments we are able to specify for each aspect to be examined the input and workload parameters for the system under test and define which observation data we are interested in. Thus, we gain more control and systematics over the measurement experiments. Due to selective, target-oriented data gathering, this approach allows us to reduce the amount of data and the overhead for gathering this data (cf. Chapter 5.2). Iteratively performing target-oriented measurements for each SPA we want to examine is a simple way to apply this target-oriented measurement approach to SPA detection. However, such an application of target-oriented measurements is rather time-consuming as advanced experiments and analyses have to be performed for each antipattern. Instead, we utilize common characteristics and dependencies of SPAs (cf. Chapter 2.1) for applying target-oriented measurements in an adaptive and systematic way. For this purpose, we

use a hierarchical approach similar to the approach presented in [MCC⁺95]. Instead of executing experiments on a high detail level for each known antipattern, we start with executing generic, high-level experiments. The results of the high-level experiments are analyzed and used to decide how to continue. Generally speaking, we use a decision tree as procedure model for antipattern detection. In Figure 4.3, a simple example for a deci-

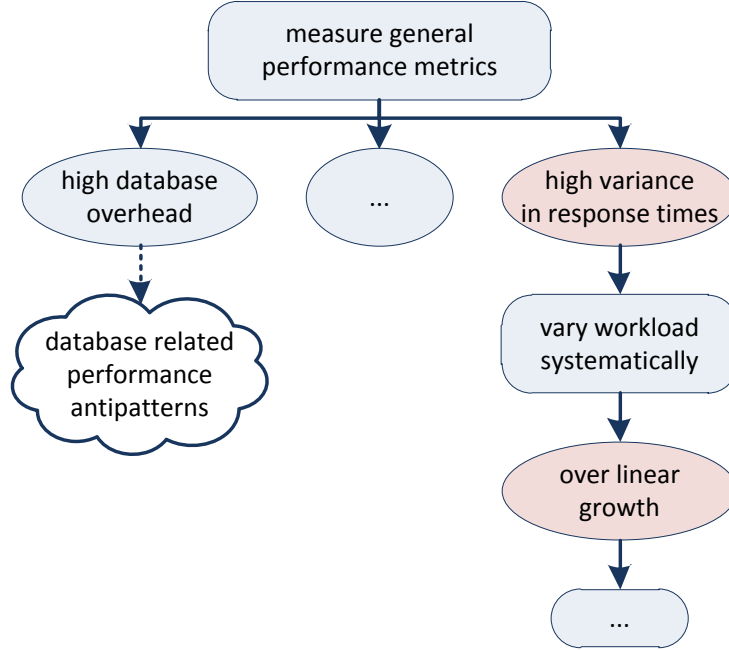


Figure 4.3.: Example for a decision tree

sion tree is depicted. At the most top level, we measure general performance metrics we use to derive the first decision. For instance, if we discover that service response times vary greatly but the database overhead is quite low, we do not examine database related antipatterns. Rather, we analyze more precisely antipatterns based on greatly varying response times. For this purpose, we execute more detailed experiments. For instance, we could measure the response times under systematical variation of the workload in order to detect over linear growth of response times. Applying this hierarchical approach, we can avoid the effort for examining antipatterns which can be excluded at a high abstraction level. For instance, there is no need to analyze database related performance antipatterns, if no database overhead could be detected with a high-level experiment. The hierarchy depicted in Figure 2.7 (Chapter 2.1.4) is a potential decision tree for SPA detection.

The overall process resulting from the adaptive measurement approach (depicted in Figure 4.4) is more complex than the sequential process (cf. Figure 4.2). As input we need a decision tree guiding the overall process. Based on the decision tree and the analysis results from previous examined aspects the next aspect to be examined is selected. The selection is performed similarly to the example in Figure 4.3. As long as further aspects can be selected, the next aspect is examined individually. For this purpose, the SUT is instrumented specifically for the considered aspect in order to enable selective, target-oriented data gathering. In contrast, the instrumentation task of Figure 4.2 instruments the SUT completely rather than selectively. Then, a series of measurement experiments with different workload parameters is executed. The workload is varied until the input parameter space is fully explored. Finally, measurement data is passed to the analysis for evaluation of the considered performance aspect. Applying the adaptive measurement approach allows us not only to reduce the measurement overhead for higher accuracy of detection results (cf. Section 5.2), but increases the efficiency of the SPA detection process.

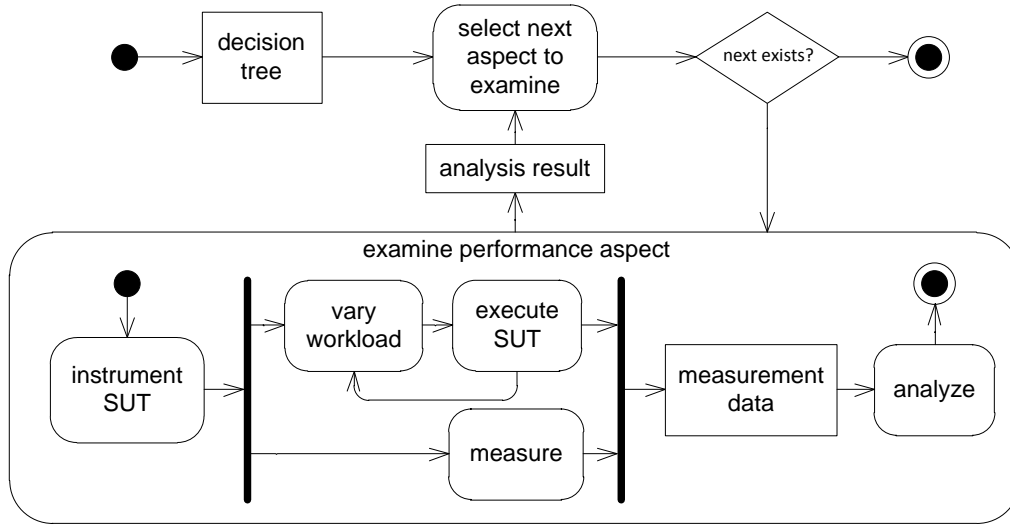


Figure 4.4.: Overall process applying adaptive measurement approach

4.3. Antipattern Detection Architecture and Process

Up to now, we considered antipattern detection from the measurement and procedural point of view. In this section, we introduce the general antipattern detection architecture and integrate it with the detection process described in the previous section (cf. Figure 4.4).

As mentioned in Section 4.1, we extend the SoPeCo (cf. Chapter 2.5) framework in order to realize the measurement-based antipattern detection approach. Consequently, the architecture for SPA detection (cf. Figure 4.5) exhibits some similarities to the SoPeCo architecture described in Section 2.5. In particular, we separate the experiment execution and analysis components from the system under test by defining a dedicated *Measurement Control Node* for experiment coordination and analysis tasks. The overall process for antipattern detection is coordinated by the *APD MainController* which gets two artifacts as input. The first artifact is a *Performance Problem Model* serving, inter alia, as a decision tree (cf. Section 4.2) for the coordination of the overall process. As the SUT might be distributed among several system nodes, the second artifact is an *Environment Specification* describing the resource environment of the SUT. The main task of the APD MainController is to traverse the performance problem model and to decide for each considered performance problem whether it has to be examined or not. With this component we realize the coordination of the adaptive approach described in Section 4.2. Thus, a performance problem is examined only when its hierarchical predecessors could be detected (cf. Figure 4.3). Besides the decision tree, the Performance Problem Model defines for each performance problem a specific *Detection Component*. A Detection Component is responsible for investigating one specific performance problem, performance antipattern respectively. For this purpose, a Detection Component comprises at minimum four sub-components. The *Detection Controller* is triggered by the APD MainController and is responsible for coordinating the antipattern specific detection and root cause analysis processes. The *Instrumentation* component encapsulates antipattern specific knowledge about instrumenting the SUT for capturing required measurement data. Thus, the Instrumentation component is used to inject antipattern specific measurement probes into the SUT (cf. Section 5). As different SPAs require different workload variations to be detected by analyzing measurement data, each Detection Component contains a *Load Variation* sub-component. A Load Variation component creates an experiment series configuration and passes it to SoPeCo for experiment execution. In order to reduce disturbing influences

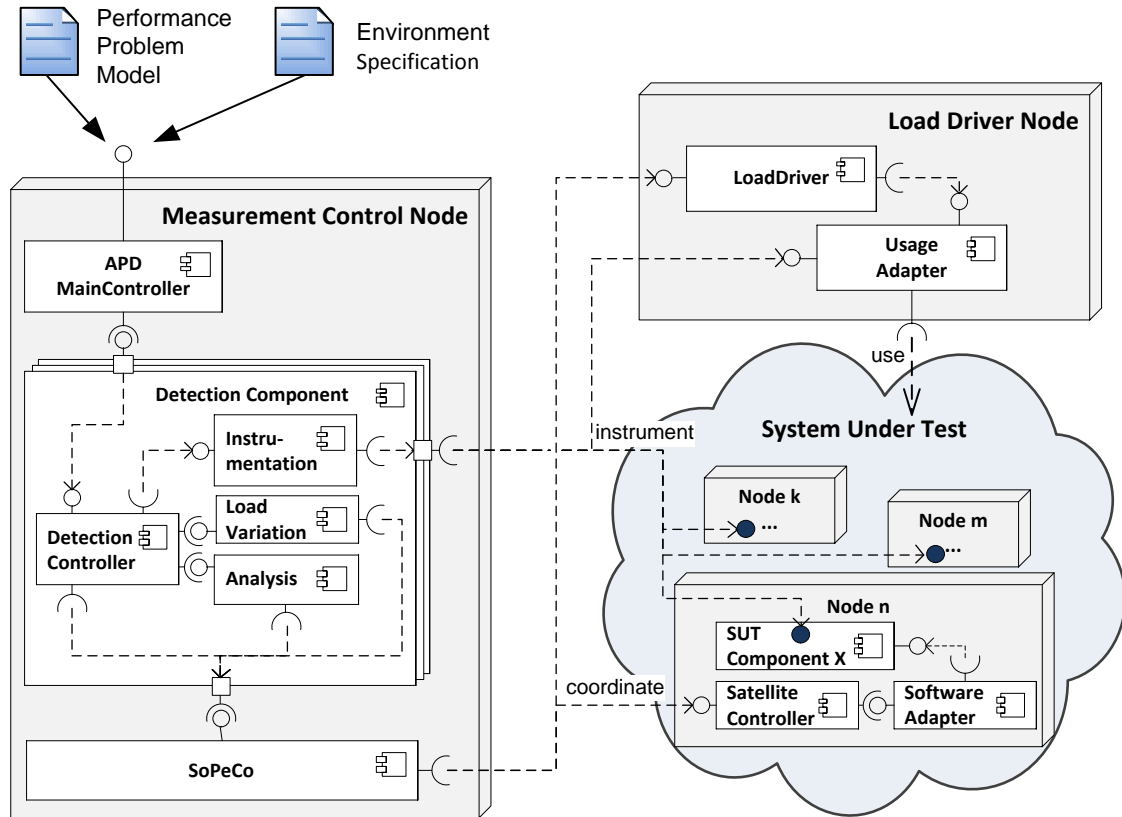


Figure 4.5.: General antipattern detection architecture

on the load driver, the LoadDriver component can be deployed on a separate node. The Detection Controller utilizes the SoPeCo for experiment execution and data gathering. In particular, SoPeCo triggers the LoadDriver to generate a workload varying it systematically according to the passed experiment series configuration. The workload generated by the LoadDriver accesses the SUT through an *UsageAdapter* component which simulates a typical usage profile for the SUT. During experiment execution, *Software Adapters* collect measurement data. When experiments are finished, the SoPeCo gathers measurements from all *SatelliteControllers* and persists it. For detection and root cause analysis, the Detection Component contains an *Analysis* component. The Analysis component retrieves measurement data from SoPeCo, evaluates the data and decides whether an antipattern has been detected, a root cause found respectively.

This architecture abstracts from antipattern specific aspects and, thus, provides extension points for antipatterns defined in the future. In order to enrich antipattern detection by a new performance antipattern, one has to provide two artifacts. Firstly, the Performance Problem Model has to be extended. Secondly, an additional Detection Component for that new antipattern has to be provided according to the architectural template. In order to illustrate the interaction between these components, we refined the process from previous section (cf. Figure 4.4) and integrated it with the described architecture. The result is depicted in Figure 4.6 and Figure 4.7. Based on the performance problem model, the APD MainController guides the overall antipattern detection process. In particular, the APD MainController selects the next performance problem in the hierarchy to be examined. Then, the APD MainController delegates the detection task to the antipattern specific Detection Component. Firstly, the Detection Component uses the Instrumentation sub-component to inject probes into the SUT. The second step of the detection task is the creation of an experiment series configuration which is used by the experiment components

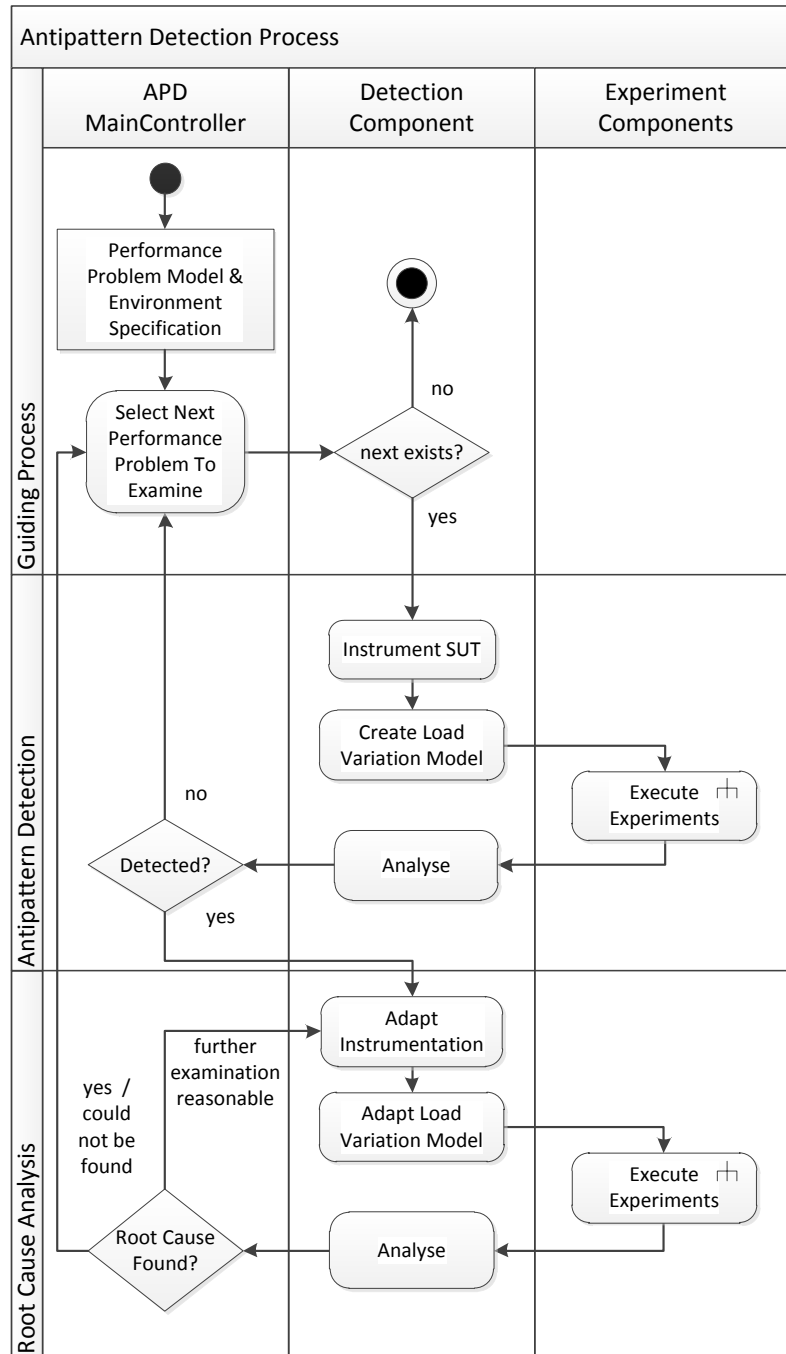


Figure 4.6.: Antipattern Detection Process

for experiment execution. The experiment execution sub-process is depicted in Figure 4.7. The SoPeCo starts an experiment series according to the passed experiment series con-

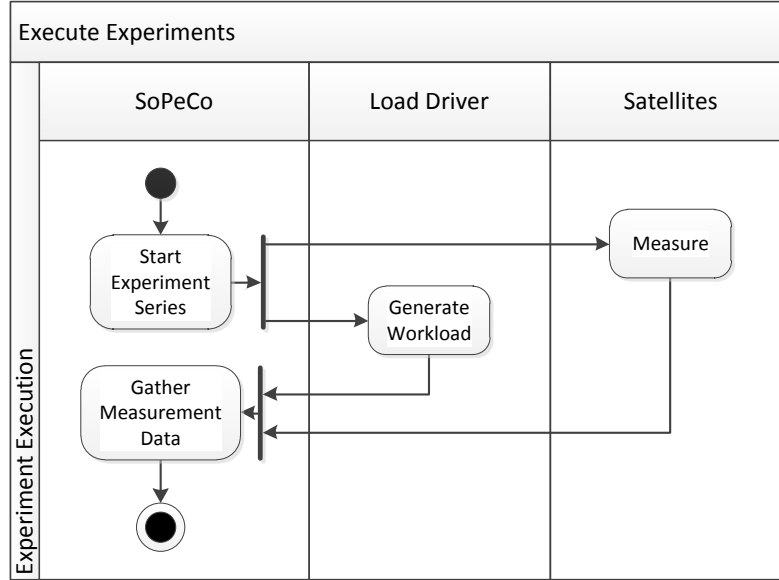


Figure 4.7.: Experiment Execution Sub-Process

figuration. During experiment execution, the LoadDriver generates a workload while the Satellites take measurements. When the experiment series is finished, the SoPeCo gathers measurement data from all Satellites and persists the data. After experiment execution the Detection Component triggers the analysis of measurement data. If the considered antipattern or performance problem could be detected by the analysis task, then the process starts the root cause analysis phase, otherwise, the process returns to the selection of the next problem to be examined. The sub-process of the root cause analysis phase is similar to the one of antipattern detection. However, instrumentation and load variation have to be adapted. The root cause analysis sub-process is repeated until a root cause could be found or another limiting factor, like analysis time restrictions, has been reached. When root cause analysis is finished, the process proceeds with problem selection. This process is repeated until the APD MainController cannot find further performance problems to be examined.

To sum up, we have two aspects, which contribute to a systematic and efficient search for antipatterns. Firstly, the overall process is guided by the APD MainController which decides for each antipattern whether it is reasonable to investigate the SUT with regards to presence of the considered antipattern. As this decision is based on previous measurements, the performance problem hierarchy can be utilized to avoid unnecessary investigations. Secondly, experiments and analyses to find the root cause of an antipattern are executed only when the according antipattern could be detected. Thus, with this two aspects, we increase the efficiency of the antipattern detection approach by avoiding unnecessary experiments.

In this section, we introduced the general architecture of our antipattern detection approach and described the antipattern detection process. In order to be able to detect specific antipatterns, we have to provide antipattern specific Detection Components. In Chapter 6, we investigate each antipattern individually and develop for each antipattern a detection and root cause analysis approach which suits best for the considered antipattern.

4.4. General Assumptions

In this section, we introduce some important assumptions on which we base our work and which apply for all antipattern considerations. We meet these assumptions in order to reduce the scope of this thesis. However, these assumptions should not limit the applicability of the described detection concept. Moreover, abandoning these assumptions is an issue for future work.

4.4.1. Monitoring Overhead

In Section 4.2, we introduced a measurement approach which minimizes the monitoring overhead during experiment execution. Nevertheless, measurements are always paired with a monitoring overhead. However, the overhead is quite low when applying the adaptive measurement approach. Thus, we assume that the overhead does not affect measurement data analysis decisively. In the following, we will neglect the monitoring overhead.

4.4.2. Knowledge About Usage

As our antipattern detection is based on systematic measurement experiments, we need information about the usage profile of the target system. In our work, we assume that this information is available to the developer or a domain expert. In particular, we assume that the developer is able to specify a typical usage profile in form of a script or a usage model. We use that usage profile for generating load on the target system.

4.4.3. No Disturbing Sources

For experiment execution we presume, that the SUT is executed in an isolated environment. In particular, we expect that no disturbing sources exist, like other running applications or services which might influence the performance of the SUT. With this assumption we assure that effects observed in measurement data are caused by the SUT, not by the environment.

4.4.4. Fixed Environment

Recent technologies like Cloud Computing [MG11] allow to keep software performance on a high level by adopting the amount of physical or virtual resources to the workload and system state. The ability to adopt the environment to the current workload situation is called *elasticity* [MG11]. In order to analyze the performance of such systems one has to examine not only the software state but also the state of the environment and underlying resources. However, in this work we consider only software and abstract from the underlying resources. Therefore, we have to assume that physical and logical environment is fixed for the time of observation. The following example shows the importance of this assumption. Imagine an application whose performance decreases over time in an environment with fixed resources. In a fixed environment, we are able to detect this decreasing behaviour solely by measuring the software performance. However, if we put this application into an elastic environment, the software performance can be kept constant through adding resources. Thus, we are not able to detect the decreasing performance behaviour anymore, without analyzing the state of the environment.

4.4.5. Standard Technologies

In order to detect performance antipattern through measurements, data from different sources is required. These sources might be certain interfaces or services, like database interfaces, messaging services, etc. For simplicity, in our work we assume that certain

standards are used. Firstly, we analyze only programs written in Java. Secondly, we assume, that the Java Database Connectivity (JDBC) interface [RO00] is used for accessing databases and the Java Messaging Service (JMS) is used for messaging. Furthermore, we assume that multi-threading systems use a thread pool for managing threads. As implementing antipattern detection for all programming languages and possible standards is beyond the scope of this thesis, this assumption serves as a small simplification. However, this assumption does not limit the applicability of the approaches introduced in this thesis. Moreover, we presume that with some adoptions the approaches described in this thesis are applicable to other languages and standards in a similar way.

4.4.6. Byte Code Analysis

Some antipatterns can be detected purely by analysing measurement data which has been captured at system boundaries considering the system under test as a black-box. However, for the most antipatterns such measurement data is insufficient. In particular, root cause analysis depends on the ability to capture measurement data from the internal of a system under test. For this purpose, we use techniques like byte code analysis in order to perform detailed instrumentation of the system under test. Therefore, we assume that such techniques are applicable on the system under test.

4.5. Summary

In this chapter, we introduced our idea and approach for detecting software performance antipatterns. Firstly, we described the performance feedback mechanism which is the main motivation for detecting SPAs during development. Secondly, we considered two different measurement approaches. Because of low measurement overhead and increased efficiency, the adaptive measurement approach emerged to be the best alternative for SPA detection. The adaptive measurement approach is based on a hierarchical measurement and decision process stepwise approaching the specific antipattern responsible for a performance problem. Based on the measurement approach, we introduced the overall process for detecting SPAs. Extending the architecture of the SoPeCo framework, we designed the architecture for SPA detection. On top of SoPeCo, we created an antipattern detection layer comprising a main controller and a set of detection components. While the main controller is responsible for coordinating the overall process the detection components are antipattern specific detection entities utilizing SoPeCo for experiment execution. We integrated the overall detection process with the described architecture. Finally, we introduced some generic assumptions we use in this thesis.

5. Instrumentation

In Chapter 4.2, we described two different approaches (*general monitoring* and *adaptive measurements*) for executing measurements. A measurement approach is tightly coupled with a proper instrumentation technique. An instrumentation technique describes the process and rules for placing *monitoring probes* into the code of the system under test. Monitoring probes are code injections responsible for tapping performance data. In this chapter, we consider for both measurement approaches possible instrumentation techniques and implementations. Firstly, we introduce the *full instrumentation* technique which is tightly coupled with the *general monitoring* approach. In particular, as mentioned in Chapter 4.2, we demonstrate the impact on the measurement overhead caused by this approach. Then, we consider two possible implementations for *dynamic instrumentation* which is the instrumentation technique required by the *adaptive measurement* approach.

5.1. Full Instrumentation

As mentioned in Section 4.2 the general monitoring approach is based on excessive data gathering as all required data has to be collected during one experiment. A proper instrumentation technique for this approach has to instrument the SUT completely. Thus, every operation call has to be intercepted for measurement purpose. However, these interceptions result in a high measurement overhead as every interception increases the response time of the executed operation. Thus, the measurement overhead affects the accuracy of measurements. We illustrate this problem on a simple example depicted in Figure 5.1. In both figures the same aggregated call tree is depicted. Tree nodes represent methods, the edges denote calls from one method to another and their frequency. While in Figure 5.1(a) only the top level method is instrumented, in Figure 5.1(b) each method is instrumented. Leaf nodes are annotated with their response times, while the response time for an inner node is the sum of response times the called methods exhibit. Under the assumption that instrumenting a method call entails an overhead of 15 ms, the response time of method **a** in the left case is 975 ms comprising an overhead of 15 ms. However, if we instrument every method call the overhead adds up to 1035 ms, resulting in a response time of 1995 ms. Note, the actual response time of method **a** is 960 ms. This simple example illustrates the impact of full instrumentation on the accuracy of measured data. Thus, we have reasons to assume that full instrumentation causes an unacceptably high monitoring overhead. In order to prove our presumption we examine this overhead effect on an actual Java

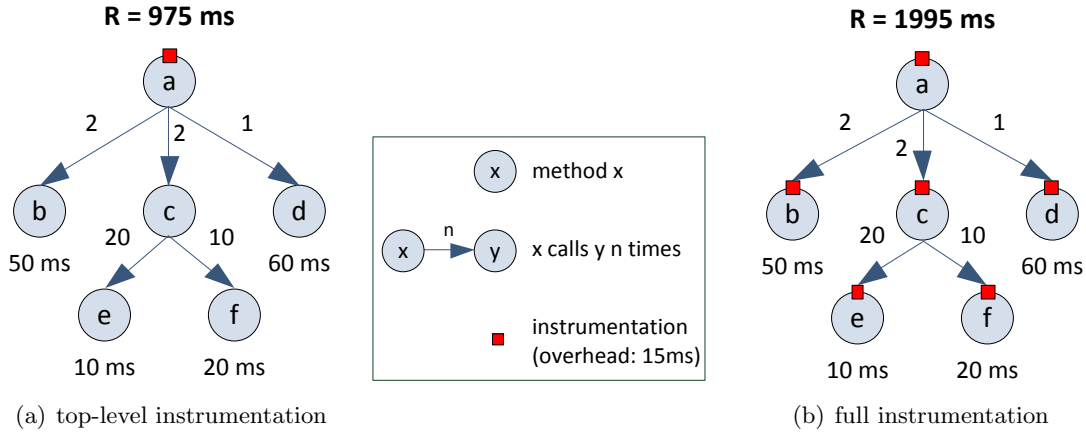


Figure 5.1.: Example: comparing partly instrumentation with full instrumentation

program. Therefore, we created a simple Java application providing some services which cause some internal method invocations. The Java application is described in more detail in Chapter 6.1. At this point, it is only important to know that the number of internal method invocations per service request increases over time. We perform two experiments, whereby we are interested in the response times of the top-level method call observing it for one minute. The two experiments differ from each other only in the instrumentation of the target application. One available implementation for the full instrumentation technique is the *OperationExecutionAspectFull* of the Kieker framework (cf. Section 2.2.4). With the first case the application is fully instrumented using the *OperationExecutionAspectFull* aspect while the second experiment is executed with top-level instrumentation (cf. Figure 5.1(a)). The experiment results are depicted in Figure 5.2. The response times do not

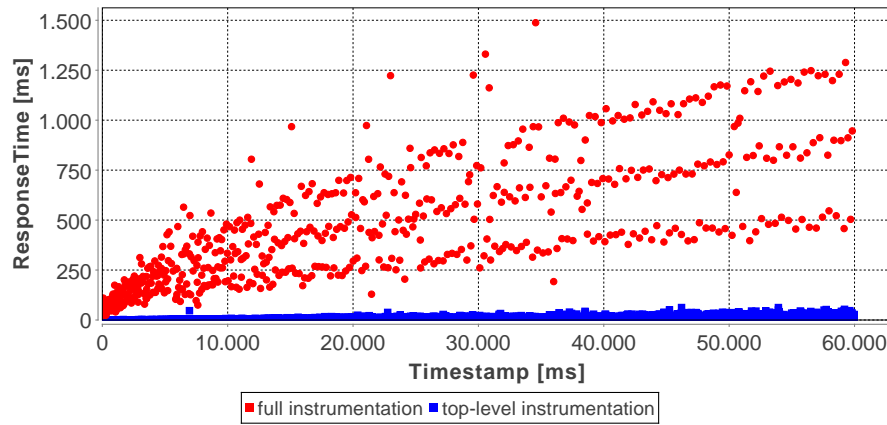


Figure 5.2.: Full instrumentation compared to top-level instrumentation

exceed 70 ms considering the case with top-level instrumentation. However, the response times grow to 1250 ms when applying full instrumentation. Note, in both cases the response times increase over time, as over time the number of internal method invocations increases. However, in the case of full instrumentation a growing number of (internal) method invocations has a progressive effect on the monitoring overhead. With this simple application, we have shown that the overhead of full instrumentation is not acceptable. Consequently, full instrumentation is not applicable in real systems, as the monitoring overhead distorts the measurements. Therefore, for SPA detection we have chosen the adaptive measurement approach which applies the dynamic instrumentation technique.

5.2. Dynamic Instrumentation

In Chapter 4.2, we introduced the adaptive measurement approach. In order to apply this approach, we need a dynamic instrumentation approach as we have to adapt the instrumentation for each experiment to be executed. Our goal is to fully automate antipattern detection which means that there has to be a central unit controlling the overall detection process (cf. *APDMainController* in Section 4.3). Therefore, it is not desirable to restart the software system each time the instrumentation of the SUT has to be adapted for a new experiment. Thus, an important requirement on an implementation of the dynamic instrumentation technique is the ability to adapt instrumentation dynamically, without having to restart the software system.

As the Kieker framework provides advanced means for simply collecting and persist measurement data we use Kieker for gathering and organizing measurement data. There are some alternatives for implementing dynamic instrumentation using Kieker. These alternatives are evaluated in the following.

5.2.1. Dynamic Instrumentation with AOP and Kieker

As mentioned in Chapter 2.2.4, Kieker provides two types of AOP/AspectJ ([HH04]) probes which allow either to monitor all methods or only annotated ones. As we have shown that full instrumentation is not applicable to antipattern detection, we discard that type of probes. However, annotation probes are not suited as well, as in general annotations are a static construct and cannot be modified dynamically without reloading classes (cf. Section 5.2.2). Thus, annotation probes cannot be used for realizing dynamic instrumentation. We need to develop an own AOP probe which allows us to dynamically select which methods should be monitored. For this purpose, we extend the *OperationExecutionAspectFull* probe of the Kieker framework by some additional features.

Firstly, we need a possibility to define a set of methods to be monitored. Thus, we extend the *OperationExecutionAspectFull* probe by a list containing method names which can be modified at runtime. Secondly, for each called method the probe must decide whether that method should be monitored or not. Usually, AspectJ utilizes pointcuts for culling join points and, thus, to decide which advice should be weaved into which method (method matching). However, this mechanism is performed during class loading. Thus, pointcuts are a semi-dynamic concept rather than a dynamic one, as pointcuts can only be changed by reloading classes [CST03]. In order to realize dynamic instrumentation we have to use generic pointcuts. These generic pointcuts shift the decision of method matching to the weaved in code, respectively advice. For this purpose, we create an aspect as depicted in Listing 5.1. The *methodsToBeMonitored* list can be used to dynamically define methods to be monitored. The pointcut *genericPointcut* matches with all method calls and is used by the around advice. Thus, during class loading the code defined within the around advice is weaved around every method execution. Every time a method is executed, the defined advice retrieves the name of the executed method and checks whether it is contained in the *methodsToBeMonitored* list. If this is the case, the executed method is monitored otherwise the advice just proceeds with executing the method.

As weaved in aspects can not be changed dynamically without reloading classes [CST03], we can not avoid to intercept each method call at least for matching it against the *methodsToBeMonitored* list. Thus similar to full instrumentation, we have reason to presume that dynamic instrumentation using AOP still causes a high measurement overhead. In order to examine our presumption we repeated the experiments from Section 5.1 for dynamic instrumentation with AOP. The results are depicted in Figure 5.3. Compared to full instrumentation, dynamic instrumentation using AOP causes a significantly lower overhead. However, compared to static top-level instrumentation the overhead is still very high. With dynamic AOP instrumentation, the response times grow to values about 400 ms, which is

Listing 5.1: Dynamic instrumentation aspect

```

aspect DynmaicInstrumentationAspect {
    public List<String> methodsToBeMonitored;
    ...
    pointcut genericPointcut() : execution(* *.*(..));

    around() : genericPointcut(){
        String methodName = thisJoinPoint.getMethodName();
        if(methodsToBeMonitored.contains(methodName))
            monitor();
        else
            proceed();
    }

    private void monitor(){
        ... // measure, use Kieker framework
        proceed();
        ... // measure, use Kieker framework
    }
}

```

more than five times higher than the greatest measured response times with top-level instrumentation (values about 70 ms). Thus, realizing dynamic instrumentation with AOP is not a satisfactory alternative.

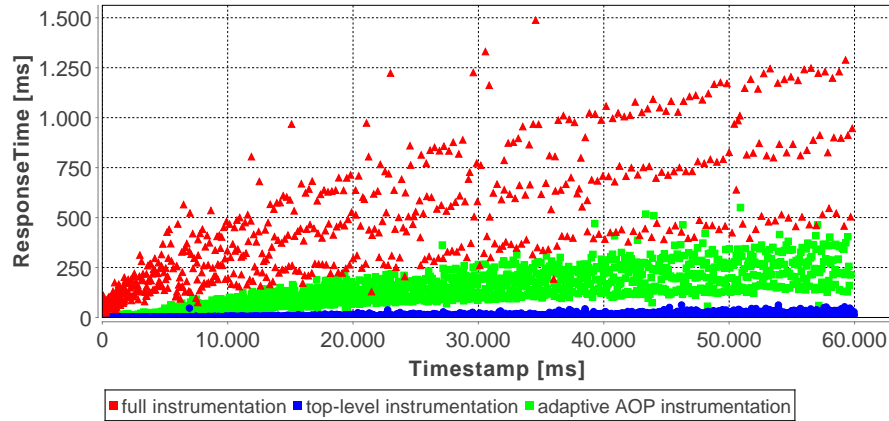


Figure 5.3.: Dynamic AOP instrumentation compared to top-level instrumentation and full instrumentation

5.2.2. Dynamic Instrumentation using Javassist, HotSwap and Kieker

As an alternative to dynamic instrumentation with AOP, we implemented an instrumentation framework based on Javassist, HotSwap and Kieker. Our instrumentation API allows to consciously instrument and remove instrumentation from certain methods. For the insertion of probes into methods we use the Javassist framework (cf. Chapter 2.2.2) which allows us to directly modify method bodies. The monitoring code of this instrumentation is equal to the advice implementation of the Kieker *OperationExecutionAspectFull* probe (cf. Chapter 2.2.4). In particular, we surround the target method with the same monitoring code as it was the case with all Kieker AOP probes (cf. Listing 5.2). In order to modify

Listing 5.2: Dynamic instrumentation with Javassist

```

public void instrumentMethod(String methodName){
    ...
    CtMethod metaMethod = ...
    metaMethod.insertBefore("
        // measure, use Kieker framework");
    metaMethod.insertAfter("
        // measure, use Kieker framework");
}

```

methods, we first retrieve the Javassist meta object for the target method. Then, we surround the method body with Kieker monitoring code using the Javassist *insertBefore* and *insertAfter* methods. Compared to the AOP implementation of the dynamic instrumentation technique there is no difference in the instrumentation code. However, the method *instrumentMethod()* of this instrumentation framework is executed at runtime, rather than during class loading. Using the HotSwap mechanism (cf. Chapter 2.2.2) we are able to reload the target-class which contains the instrumented method. In this way, we decouple the instrumentation phase from the experiment phase. In contrast, with dynamic AOP instrumentation these phases were mixed up as during execution all methods had to be checked against the list *methodsToBeMonitored*. Thus, during execution of the target application only instrumented methods are intercepted for measurement. As we are able to consciously instrument specific methods, not instrumented methods are not intercepted during execution and do not have to be checked against a list.

In order to compare this instrumentation approach to the previous alternatives, we

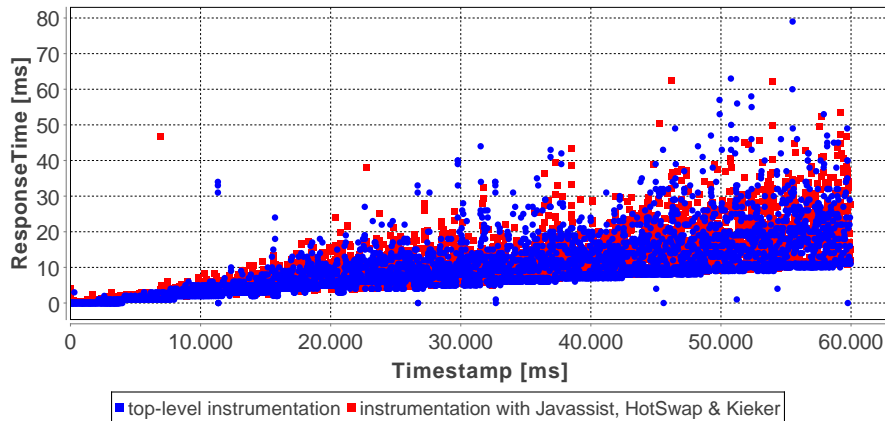


Figure 5.4.: Dynamic instrumentation with Javassist, HotSwap & Kieker compared to top-level instrumentation

repeated the experiments from Section 5.1 applying the considered instrumentation implementation with Javassist, HotSwap and Kieker. The measurement results are depicted in Figure 5.4. The response times taken with the considered implementation are hardly distinguishable from the response times captured using top-level instrumentation. Thus, we can assume that instrumenting the SUT using Javassist, HotSwap and Kieker causes only a negligible monitoring overhead. However, the overhead is low only if the instrumentation framework is used in a target-oriented and selective manner according to the adaptive measurement approach.

6. Antipattern Detection in Detail

In this chapter, we examine the detection approach for individual antipatterns in more detail. To place this chapter into the overall context, individual investigations of each SPA in this chapter correspond to realizations of single *Detection Components* which are described in Chapter 4.3 (cf. Figure 4.5). As the scope of investigating all introduced antipatterns (cf. Chapter 2.1.4) is too large, in this chapter, we examine only a part of these antipatterns. Figure 6.1 depicts a part of the performance problem hierarchy from Chapter 2.1.4 (cf. Figure 2.2) comprising the SPAs we investigate in this chapter. In particular this part of the hierarchy contains antipatterns derived from the Varying Response Times indicator.

A closer look at Figure 6.1 reveals that we have removed the Traffic Jam antipattern from this hierarchy. As explained in [SW02b], the Traffic Jam antipattern can have two possible reasons: the One Lane Bridge antipattern or a temporarily high service demand. As analyzing the real workload for the system under test is beyond the scope of this thesis, we do not include the high demand cause into our considerations. Thus, for our considerations the One Lane Bridge antipattern is the only one possible cause for the Traffic Jam. Under these circumstances we are not able to distinguish an occurrence of a Traffic Jam from an One Lane Bridge. Therefore, we abstain from investigating the Traffic Jam antipattern.

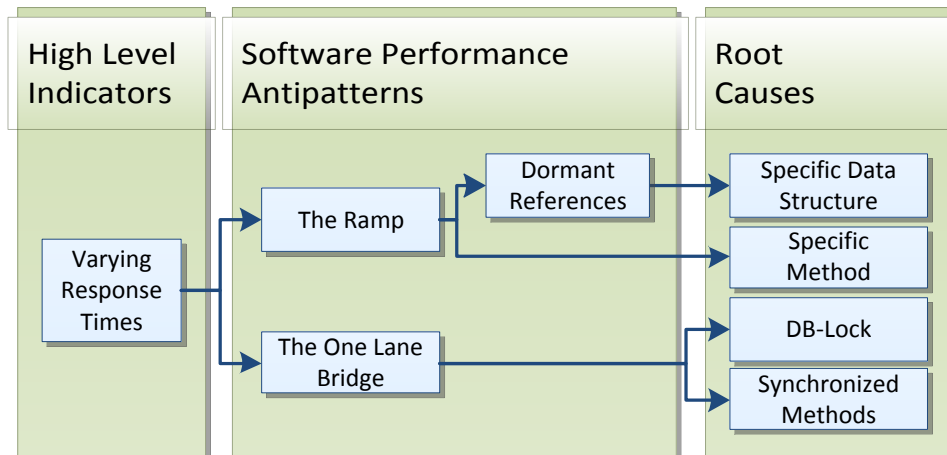


Figure 6.1.: Part of the performance problem hierarchy

For the high level indicator and each SPA depicted in Figure 6.1, we examine different alternatives for detecting these performance problems and finding the root causes. In order to select the most accurate detection technique we need a measure allowing us to compare different alternatives. For this purpose, in Section 6.1, we design a simple software system which we use for validation of individual detection techniques. Furthermore, we provide different implementations for the validation system which define different scenarios. Each scenario contains one or more antipatterns exhibiting behaviours to be investigated. From the observation point of view, the scenarios differ from each other in the antipatterns they contain, and thus, in their performance behaviour. For each investigated SPA, we determine our expectation for which scenarios the considered antipattern must be detected. Based on these expectations, we define a measure for comparing different detection techniques. In the following we describe this measure:

Let $\vec{s} = (s_1, \dots, s_n)$ be the vector of all defined scenarios. For an antipattern a , we define the *expectation vector* \vec{v}_a as:

$$\vec{v}_a = (v_{a,1}, \dots, v_{a,n}), \quad v_{a,i} = \begin{cases} 1 & , s_i \text{ contains } a \\ 0 & , \text{otherwise} \end{cases} \quad (6.1)$$

Let $t_a = \{t_{a,1}, \dots, t_{a,m}\}$ be the set of considered detection techniques for antipattern a . Applying a detection technique $t_{a,k}$ on all scenarios s_i yields a *detection vector* $\vec{d}_{a,k}$:

$$\vec{d}_{a,k} = (d_{a,k,1}, \dots, d_{a,k,n}), \quad d_{a,k,i} = \begin{cases} 1 & , t_{a,k} \text{ detected } a \text{ in } s_i \\ 0 & , \text{otherwise} \end{cases} \quad (6.2)$$

Based on the expectation vector and the detection vector, for an antipattern a and a detection technique $t_{a,k}$ we define the *error vector* $\vec{e}_{a,k}$:

$$\vec{e}_{a,k} = \vec{d}_{a,k} \oplus \vec{v}_a \quad (\oplus : \text{binary addition}) \quad (6.3)$$

The error vector $\vec{e}_{a,k}$ describes for which scenarios the detection technique $t_{a,k}$ made wrong decisions. A wrong decision occurs, if a detection technique detects an antipattern a in a scenario s although s does not contain a , or vice versa.

$$\vec{e}_{a,k} = (e_{a,k,1}, \dots, e_{a,k,n}), \quad e_{a,k,i} = \begin{cases} 1 & , t_{a,k} \text{ made a wrong decision for } s_i \\ 0 & , \text{otherwise} \end{cases} \quad (6.4)$$

Summing up all wrong decisions in $\vec{e}_{a,k}$ and normalizing this sum yields the *error rate* $r_{a,k}$ for detection technique $t_{a,k}$:

$$r_{a,k} = \frac{\vec{e}_{a,k} \cdot \vec{1}}{n} \quad (6.5)$$

We use the error rate as a measure for comparing different detection techniques regarding individual SPAs. Considering an antipattern a , a detection technique $t_{a,1}$ with error rate $r_{a,1}$ is more accurate than a detection technique $t_{a,2}$ with error rate $r_{a,2}$, if $r_{a,1} < r_{a,2}$.

In the following section, we introduce the validation system and define ten scenarios used for validation. In Section 6.2, we investigate techniques for detecting the Varying Response Time indicator. Section 6.3 deals with detection of the Ramp antipattern and search for its root causes. In Section 6.4, we examine the Dormant References antipattern. Finally, we investigate the One Lane Bridge and its possible root causes in Section 6.5. For each investigated antipatterns, our goal is to find a detection technique with a detection error rate of zero for the considered scenarios.

6.1. Validation System and Scenarios

In this section, we introduce a software system used for validation of single SPA detection techniques. For this purpose, we implemented an artificial Online Banking System which provides some services to the system user. However, instead of performing real work, these services simulate certain behaviours executing different operations and generating load on resources. We provide different variants of the Online Banking System consciously injecting SPAs into the implementation. Thus, we consider different scenarios which we use for validation of individual SPA detection techniques. For each considered SPA manifestation, we define at minimum one *positive scenario* and one *negative scenario*. While *positive scenarios* contain certain performance antipatterns, *negative scenarios* do not exhibit the considered antipattern behaviour. Thus, a detection technique for a considered SPA is valid if it detects the antipattern in all *positive scenarios* but does not react on *negative scenarios*. In the following, we provide a short description of the validation system and describe ten scenarios we use to validate detection techniques for individual SPA manifestations and performance problems.

6.1.1. Online Banking System

In order to avoid disturbing sources, we exclude as much external effects as possible during validation of the SPA detection techniques. However in Chapter 7, we conduct an advanced evaluation on a more complex and realistic software system. Here, we keep the validation system as simple as possible implementing only those parts required for simulating certain SPA behaviours. Accordingly, the system depicted in Figure 6.2 is quite simple comprising only two components. While the *Transaction Manager* is responsible for the processing of financial transactions, the *Account Manager* provides services for administering account data. Finally, all data is stored in an external database which we consider as a black-box. For validation, we distribute this system among two system nodes. For this purpose,

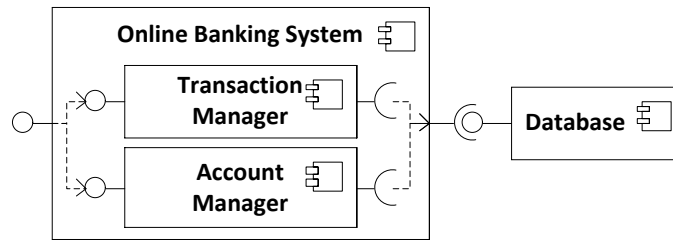


Figure 6.2.: Online Banking System

we deploy the database on a dedicated database node. Assuming that the execution of the SoPeCo has only a low overhead, we run the application part of the Online Banking System on the same node as SoPeCo's Measurement Control components (cf. Figure 2.10, Figure 4.5). The user interface of the Online Banking System comprises three services. Each service is designed for the validation of one certain SPA manifestation.

- **commitTransaction:** This service allows the user to commit financial transactions. For this purpose, the Online Banking System utilizes the *Transaction Manager* component. In order to avoid unintentional duplicate transactions, the Transaction Manager compares each committed transaction against a set of previously committed transactions. For this purpose, the Transaction Manager accesses a set of the most current transactions before irrevocably committing transactions. Different implementations of the Transaction Manager entail different performance behaviours which is described in Section 6.1.2. We use this service to validate detection techniques for performance problems related to the Ramp antipattern.

- **viewAccountState:** This service allows the user to display the financial state and statistics of his account. `viewAccountState` comprises three sub-tasks performed by the *Account Manager*:
 - *retrieveInformation:* Reads data from the database.
 - *generatesStatistics:* Generates an average CPU demand of 20 ms.
 - *renderGraphs:* Generates an average CPU demand of 80 ms.

While the `retrieveInformation` sub-task requests the database to retrieve information, the other two tasks generate CPU demands which simulate real work to be done. In Section 6.1.2, we describe different implementations of the *Account Manager* leading to different synchronization behaviours. Thus, we use this service for validation of detection techniques concerning the *Synchronized Methods* manifestation of the One Lane Bridge antipattern and all its hierarchical predecessors.

- **changePersonalData:** This service is used to change personal user data. For this purpose, the Account Manager accesses a database containing this data. We examine different approaches to use a database in order to validate detection techniques for the *DB-Lock* manifestation of the One Lane Bridge and all its hierarchical predecessors.

In the following, we describe different scenarios based on different implementations of these simple services.

6.1.2. Scenarios

In this section, we describe for each manifestation of an SPA some *positive* and *negative scenarios*. These scenarios are used not only for the validation of detection techniques for specific SPA manifestation, but for the detection steps along the performance problem hierarchy (cf. Figure 6.1) as well. For instance, we use scenarios related to the One Lane Bridge antipattern in order to validate detection techniques for the Varying Response Time problem, too. In the first step, we examine the detection techniques for each antipattern individually and isolated. Thus, the first nine scenarios differ in the implementations of single services provided by the Online Banking System. The last scenario combines some performance problems in order to examine interaction effects.

- **Scenario 1 - classic ramp behaviour:**

As described in the previous section, the `commitTransaction` service retrieves a set of previously committed transactions in order to avoid duplicates. In this scenario, the Transaction Manager stores all current transactions in a file on the hard disk in order to avoid high memory consumption. For each transaction committed by a user, the Transaction Manager reads the file from disk, parses it and compares its entries with the newly committed transaction. Once a day, the Transaction Manager commits irrevocably all transactions from this file. Depending on the intensity of the workload, the size of the file can grow significantly during this time window. However, for duplicate avoidance each commit of a transaction has to be checked against all transactions committed at the same day. This circumstances lead to increasing response times as the size of the file increases. The response time progression of this scenario for a request arrival rate of $10s^{-1}$ is depicted in Figure A.1(a). As this scenario represents a classic ramp behaviour, a valid detection technique for the Ramp antipattern has to detect the presence of this antipattern in Scenario 1. Thus, Scenario 1 is a *positive scenario* for the Ramp antipattern. Furthermore, this scenario is a *positive scenario* for the Varying Response Time indicator as each ramp behaviour results in highly varying response times. Regarding the One Lane Bridge and the Dormant References antipatterns, this scenario is a *negative scenario*.

- **Scenario 2 - growing list:**

The second scenario refers to the `commitTransaction` service as well. However, instead of storing transactions in a file, in this scenario, the Transaction Manager holds current transactions as a list in the memory to avoid overhead for accessing the hard disk. For duplicate avoidance the Transaction Manager iterates the whole list to find duplicates. With a growing list the time for duplicate search increases. Thus, in this case, using a list as a repository is a simple implementation mistake leading to a ramp behaviour of response times (cf. Figure A.1(b)). Additionally, the memory consumption increases over time as the list is hold in memory. Consequently, this scenario is a *positive scenario* for the Varying Response Times problem, the Ramp and the Dormant References antipattern. For the One Lane Bridge antipattern this scenario is a *negative scenario*.

- **Scenario 3 - periodical clean-up:**

This scenario is similar to Scenario 2. Again, the Transaction Manager uses a list to keep current transactions of the `commitTransaction` service in memory. In contrast to Scenario 2, the Transaction Manager in this scenario performs a periodical clean-up of the whole repository. Semantically, this might mean that transactions are kept in the memory not for the whole day but only for some hours. This implementation of the Transaction Manager results in a response time behaviour as depicted in Figure A.1(c). Under a stable workload, the response times grow between two clean-up phases. As this increase in response times does not exceed a certain limit, on the whole, response times do not increase over time. Thus, in contrast to the first two scenarios, this is a *negative scenario* for the Ramp and the Dormant References antipattern. Equivalently to the first two scenarios this scenario is a *negative scenario* for the One Lane Bridge antipattern, too. Although this scenario does not contain a Ramp, response times vary significantly between two clean-up phases, resulting in a high overall variance. Therefore, this scenario is a *positive scenario* for the Varying Response Time indicator.

- **Scenario 4 - fixed-sized queue:**

Scenario 4 refers to the `commitTransaction` service, too. In this scenario we use a fixed-sized first-in-first-out queue as repository for the transactions. Considered over time, the queue size increases until the maximum size is reached. Then, the queue starts to drop the oldest transaction when a new transaction is committed. As response times behave proportionally to the queue size, with this scenario we observe a saturating behaviour as depicted in Figure A.1(d). As response times are limited by the maximum queue size, the ramp behaviour occurs only during the initial phase, but not on the whole. Thus, this is a *negative scenario* for the Ramp and the Dormant References antipattern. Assuming that the initial phase is rather small compared to the observation time the variation of response times caused by the initial phase can be neglected. Thus, this scenario causes only little variation in response times resulting in a *negative scenario* for the Varying Response Time indicator. Scenario 4 is a negative scenario for the One Lane Bridge antipattern, as well.

- **Scenario 5 - hashing transactions:**

This implementation of the Transaction Manager uses a hash-based repository for holding transactions of the `commitTransaction` service in memory. As searching for duplicates in a hash-based data structure can be done in $O(1)$, duplicate avoidance is independent of the data structure size. Using a hash-based data structure results in a stable response time behaviour (cf. Figure A.1(e)). As response times do not increase over time and their variance is quite low, Scenario 5 is a *negative scenario* for the Ramp, the Varying Response Times problem and the One Lane Bridge an-

tipattern. Although this is a *negative scenario* for the Ramp antipattern Scenario 5 contains a Dormant References antipattern as the memory consumption increases over time. Note, this circumstances do not form a contradiction to the *Adaptive Measurement Approach* (cf. Section 4.2). As the Dormant References antipattern has two hierarchical predecessors (cf. Figure 2.2), a software system can contain a Dormant References antipattern although it does not exhibit a Ramp behaviour.

- **Scenario 6 - synchronized method:**

In this scenario, we consider the `viewAccountState` service. As mentioned before, this service is divided into three sub-tasks. Here, we assume that the developer of the Account Manager implements the `viewAccountState` service as a synchronized method in order to avoid concurrent access to the account storage. However, as the whole method is synchronized, this scenario quickly leads to congestion, and thus, highly varying response times. The response time behaviour for this scenario is depicted in Figure A.1(f). Actually, this scenario is a typical case of the Synchronized Methods manifestation of the One Lane Bridge antipattern. Thereby, Scenario 6 is a *positive scenario* for the One Lane Bridge and the Varying Response Times problems. Considering the Ramp and the Dormant References antipattern this scenario is a *negative scenario*.

- **Scenario 7 - resolved synchronization:**

Scenario 7 is a solution to the problem of the `viewAccountState` service in Scenario 6. Considered more closely, the critical task to be synchronized is the `retrieveInformation` sub-task of the `viewAccountState` service, as this is the point where the account storage is accessed. As this sub-task consumes only a small part of the `viewAccountState` service's response time, much synchronization time can be avoided by synchronizing solely the `retrieveInformation` sub-task. This solution leads to a more stable behaviour, as can be seen in Figure A.1(g). As the response time behaviour of this scenario is quite stable, this scenario is a *negative scenario* for the Varying Response Time indicator and all its hierarchical successors.

- **Scenario 8 - storing data as big byte arrays:**

We use Scenario 8 for examining the database manifestation of the One Lane Bridge antipattern. For this scenario we provide an implementation for the `changePersonalData` service of the Account Manager which stores the entire account repository as one large object in the database. In order to modify a single account entry, the Account Manager has to store the whole repository. This circumstances lead to high locking times while all other threads wanting to access the same database table have to wait. The response time behaviour of this scenario is depicted in Figure A.1(h) showing greatly varying response times caused by the database lock. With Scenario 8, we observe a typical database manifestation of the One Lane Bridge antipattern. Equivalently to Scenario 6, this scenario is a *positive scenario* for the One Lane Bridge and the Varying Response Times performance problems. For the Ramp and the Dormant References antipattern this scenario is a *negative scenario*.

- **Scenario 9 - using proper database structures:**

In Section 9, we solve the problem of the `changePersonalData` service in Scenario 8. Instead of storing the entire account repository object as a whole, we create a proper database schema for storing single account entries directly in a database table. Thus, accessing one account entry locks the database table for a negligibly small period of time. The response time behaviour of this scenario (cf. Figure A.1(i)) is similar to Scenario 7 as synchronization is reduced significantly. Thus equivalently to Scenario 7, this is a negative scenario for the Varying Response Time indicator and all its hierarchical successors.

- **Scenario 10 - SPA interaction:**

In the previous nine scenarios, we isolated single performance problems. In order to examine interaction effects among individual antipatterns, in Scenario 10, we combine the behaviour of Scenario 1, Scenario 7 and Scenario 8. Thus, Scenario 10 comprises all three services of the Online Banking system containing the Ramp antipattern and the database manifestation of the One Lane Bridge. The response time behaviour for this scenario is depicted in Figure A.1(j). The detection techniques for both antipatterns should be able to detect the corresponding antipatterns despite additional performance effects. Scenario 10 is a *positive scenario* for the Varying Response Times problem, the Ramp and the One Lane Bridge antipattern. For the Dormant References antipattern this scenario is a *negative scenario*.

In this section, we introduced a simple validation system which we designed for validation of SPA detection techniques. Based on this system, we defined ten scenarios by providing different implementations for single components of the validation system. For each scenario, we described our expectations regarding the detection results of the detection techniques. In Figure 6.3, we summarized our expectations for each scenario. Here,











		Varying Response Times	Ramp	Dormant References	One Lane Bridge	Sync. Methods	DB-Lock
Sc. 1: classic ramp behaviour		detect	detect	---	not	---	---
Sc. 2: growing list		detect	detect	detect	not	---	---
Sc. 3: periodical clean-up		detect	not	---	not	---	---
Sc. 4: fixed-sized queue		not	---	---	---	---	---
Sc. 5: hashing transactions		not	---	(detect)	---	---	---
Sc. 6: sync. method		detect	not	---	detect	detect	not
Sc. 7: resolved synchronization		not	---	---	---	---	---
Sc. 8: storing big byte arrays		detect	not	---	detect	not	detect
Sc. 9: using db structures		not	---	---	---	---	---
Sc. 10: SPA interaction		detect	detect	not	detect	not	detect

Figure 6.3.: Detection result expectations for each considered scenario

“detect” denotes the presence of the corresponding performance problem, whereby, a “not” symbolizes that the considered scenario does not contain the corresponding performance problem. According to the *Adaptive Measurement Approach* (cf. Chapter 4.2), there is no need to dig deeper into the performance problem hierarchy if a scenario does not contain the considered performance problem. For instance, Scenario 4 does not exhibit a Varying Response Times behaviour. Thus, there is no need to investigate any performance problems which are hierarchical successors of the Varying Response Times problem. In Figure 6.3, these cases are denoted by a hyphen. The Dormant References antipattern for Scenario 5 is an exception of this rule as the Dormant References antipattern has two hierarchical predecessors (cf. Figure 2.2). For detailed validation of single detection techniques we use *all ten* scenarios. Therefore, during validation we consider the cases which

are denoted by a hyphen in Figure 6.3 as “not” detectable. In these cases we expect from a specific detection technique that it does not detect the considered antipattern for the corresponding scenario.

In the following, we examine each detection step in detail using these scenarios to compare different alternatives for detecting performance problems. Whereby, a valid detection technique for a performance problem should detect the problem in all scenarios which are denoted by a “detect” in Figure 6.3, but must not react on scenarios denoted by a “not” or a hyphen.

6.2. The Varying Response Times Detection

In this section, we examine detection techniques for the top element of our performance problem hierarchy: the Varying Response Times (VRT) indicator (cf. Figure 6.1). Firstly, we describe the experiment configuration used for detecting the VRT problem. Then, we investigate different data analysis techniques for detecting highly varying response times.

6.2.1. Experiment Setup

6.2.1.1. Problem Specific Instrumentation

As the Varying Response Times problem is a rather general problem, we instrument the system under test (SUT) at the top abstraction level. For this purpose, we measure the response time for each service request at the user interface. In parallel, we sample the CPU utilizations of each system node. Based on this data, we perform the analysis for detecting highly varying response times.

6.2.1.2. Proper Workload for Experiments

As the name of the considered performance problem suggests, we are interested in the variance of response times. However, before we are able to examine this issue we have to determine under which workload the response times should be examined. In the case of synchronization problems leading to highly varying response times, we are especially interested in situations when the SUT becomes unstable (cf. Section 2.4) under a overload situation. We use an open workload for the execution of experiments in order to examine unstable situations.

For isolated examination of this problem, we use only one isolated workload class per examined scenario. Thus, the usage behaviour is fixed for one scenario.

The workload intensity is the third property to be specified. However, we cannot choose the workload intensity arbitrarily. Synchronization problems leading to highly varying response times, for instance, occur only if the workload intensity is high enough to result in concurrent processing. Thus, in order to find such performance problems, for each scenario and SUT we have to determine sufficient workload intensity values. However, these values differ from case to case. The software system approaches its load limit, if at minimum one passive or active resource is highly utilized. In order to find this point, we start the measurements with a very low workload intensity. From one experiment to the next, we increase the workload intensity until the average CPU (, disk, memory or network) utilization of a system node exceeds 90% or the response times increase dramatically. Both circumstances indicate a high load situation which is the point of interest. In order to reduce the amount of required experiments, we increase the intensity exponentially rather than linearly by doubling the load for each new experiment. For the detection of the Varying Response Times indicator, we evaluate the response time variance for each workload intensity below the point of interest. In the following, we consider different analysis techniques for detecting high variance in response times based on data retrieved from these experiments.

6.2.2. Detection Techniques

The most intuitive way to evaluate the variance of response times is to use the statistical variance $\sigma^2(R)$ or the standard deviation $sd(R)$ (cf. Chapter 2.3). For instance, we can examine whether $sd(R)$ exceeds a certain threshold. However, as $\sigma^2(R)$ and $sd(R)$ are absolute values it is difficult to find a generic threshold. Therefore, we use the coefficient of variance (COV) (cf. Chapter 2.3) for the evaluation of the response time variance.

6.2.2.1. Fixed COV Threshold

As mentioned before, we repeat experiments increasing the workload intensity until a high system load is reached. For each workload intensity value, we check whether the coefficient of variance $COV(R)$ exceeds a threshold T_{COV} . For the scenarios described in Section 6.1.2, we examined this detection technique varying the value for T_{COV} between 0.1 and 0.9. The detection results are depicted in Table 6.1. If $COV(R)$ is greater or equal

Scenarios	v_{vrt}	T_{COV}								
		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Scenario 1	D	D	D	D	D	D	D	D	D	D
Scenario 2	D	D	D	D	D	D	D	D	D	<i>N</i>
Scenario 3	D	D	D	D	D	D	D	D	D	D
Scenario 4	N	<i>D</i>	<i>D</i>	<i>D</i>	<i>D</i>	<i>D</i>	N	N	N	N
Scenario 5	N	<i>D</i>	<i>D</i>	<i>D</i>	<i>D</i>	N	N	N	N	N
Scenario 6	D	D	D	D	D	D	D	D	<i>N</i>	<i>N</i>
Scenario 7	N	<i>D</i>	<i>D</i>	N	N	N	N	N	N	N
Scenario 8	D	D	D	D	D	D	<i>N</i>	<i>N</i>	<i>N</i>	<i>N</i>
Scenario 9	N	<i>D</i>	<i>D</i>	<i>D</i>	<i>D</i>	<i>D</i>	N	N	N	N
Scenario 10	D	D	D	D	D	D	D	D	D	<i>N</i>
detection error rate	—	0.4	0.4	0.3	0.3	0.2	0.1	0.1	0.2	0.4

Table 6.1.: Detection experiments with different values for the threshold T_{COV} .

D: detected, N: not detected, red/italic: wrong decision

T_{COV} for any considered workload intensity, the Varying Response Times problem has been detected by the considered analysis technique for the corresponding scenario. These cases are denoted by a D in Table 6.1. Cases in which $COV(R)$ is smaller than T_{COV} for all considered workload intensities are denoted by a N meaning that high variance in response times could *not* be detected in the corresponding scenario. The first column contains the *expectation vector* v_{vrt} for the Varying Response Time antipattern. The other columns contain *detection vectors* for each examined COV threshold. Wrong detection decisions are illustrated by an italic (and red) letter.

The thresholds with the best detection error rates are 0.6 and 0.7. However, Table 6.1 demonstrates that we are not able to find a proper threshold T_{COV} for which the analysis technique would make right decisions for all considered scenarios. Thus, using a fixed threshold for the coefficient of variance leads to inaccurate detection results for the Varying Response Times problem.

6.2.2.2. Adapted COV Threshold

Because of measurement imprecision, $COV(R)$ is very unstable for small response times. Let us assume we have a measurement resolution of 15 ms. Little deviations (e.g. 15 ms) from an average response time of 50 ms affect the value of $COV(R)$ to a much higher degree than for higher response times (e.g. 1000 ms). Thus, depending on a central value

of the response times we have to adapt the threshold for $COV(R)$. In order to handle the instability of $COV(R)$ we “allow” higher threshold values for small response times. In contrast, T_{COV} should be small for large response time values. Realizing this idea, we define a function $f_{COV}(R)$ describing proper values of T_{COV} in dependence of the median value \tilde{x} of measured response times:

$$f_{COV}(\tilde{x}) = COV_{min} * (m - \frac{m-1}{\frac{t_{mid}}{\tilde{x}} + 1}) \quad m = \frac{COV_{max}}{COV_{min}} \quad (6.6)$$

The function $f_{COV}(\tilde{x})$ is defined by three configuration parameters: COV_{min} , COV_{max} and t_{mid} . COV_{min} is the minimal threshold which is approached when response times are quite large. COV_{max} is approached when response times are very small. t_{mid} determines for which response time the function $f_{COV}(\tilde{x})$ has the central value COV_{mid} :

$$f_{COV}(t_{mid}) = COV_{mid} = \frac{COV_{min} + COV_{max}}{2} \quad (6.7)$$

We use the value 0.1 for COV_{min} , 0.9 for COV_{max} and 500 ms for t_{mid} . The value in parenthesis is between 1 and 9 multiplying COV_{min} by 9 in the case of very small response times. Thus, the threshold is near 0.9 if \tilde{x} is very small. When response times increase, $f_{COV}(\tilde{x})$ decreases rapidly, reaching a value of 0.5 for a response time of 500 ms. In Figure 6.4, the graph of this function is depicted for response time values between 0 ms and 5000 ms. In order to demonstrate the effectiveness of the adapted COV threshold

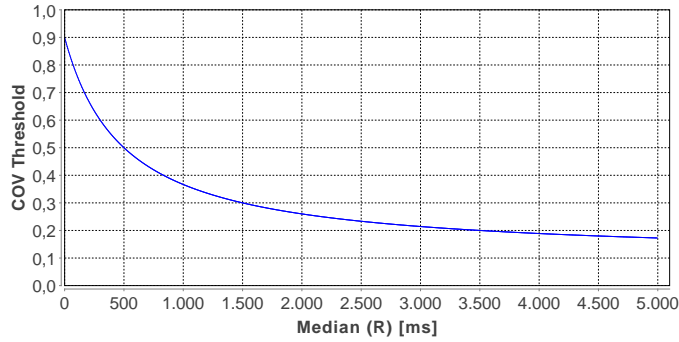


Figure 6.4.: $f_{COV}(\tilde{x})$ for configuration values $COV_{min} = 0.1$, $COV_{max} = 0.9$ and $t_{mid} = 500ms$

$f_{COV}(\tilde{x})$, we depicted the median \tilde{x} and the coefficient of variance $COV(R)$ for Scenario 8 and Scenario 9 in Figure 6.5. The top graphs show for each examined arrival rate the median response time value. The points in the bottom graphs show the calculated COV of measured response times. The curves show the adapted COV threshold $f_{COV}(\tilde{x})$. As described in Section 6.1.2, Scenario 8 is a typical case of the One Lane Bridge antipattern resulting in highly varying response times, while Scenario 9 does not contain an antipattern. However, in both cases the calculated $COV(R)$ values are in a similar range making it impossible to find an adequate fixed COV threshold. The adapted threshold overcomes this problem. In Figure 6.5(a), the threshold decreases to a value of 0.15 because of sharp increase of the median \tilde{x} . Thus, under an arrival rate of $16s^{-1}$ the analysis technique detects the Varying Response Times indicator. For Scenario 9, the threshold is quite high for all arrival rates as the values for the median \tilde{x} are very low. Consequently, in Scenario 9 varying response times could not be discovered. In both cases, the analysis provides a correct detection decision. We applied this analysis technique with an adapted COV threshold to all scenarios yielding an error detection rate of zero. Thus, using an adaptive threshold for the coefficient of variances is a valid approach for detecting the Varying Response Times problem.

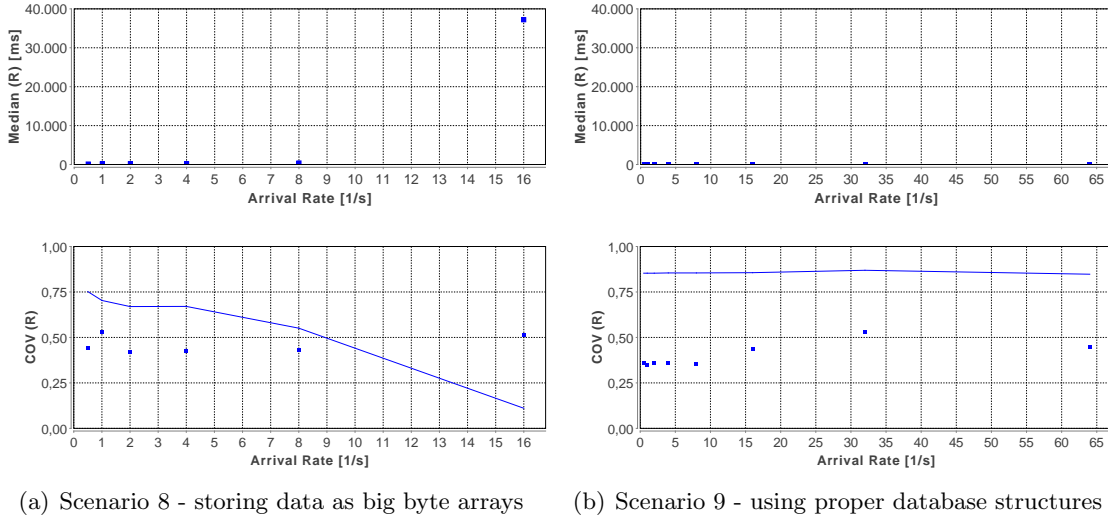


Figure 6.5.: $COV(R)$ and $f_{COV}(\tilde{x})$ depicted over the request arrival rate for Scenario 8 and Scenario 9

6.3. The Ramp Detection

In this section, we dig one step deeper into the performance problem hierarchy examining different detection techniques for the Ramp antipattern. Furthermore, we explain the root cause analysis approach for the Ramp antipattern.

6.3.1. Detection

The detection of the Ramp antipattern is based on the assumption that the observation interval necessary to detect it is known. The Ramp is a time dependent antipattern which can be detected only under a long enough observation phase. Here, we assume that the developer of the system under test has the knowledge to specify an adequate approximation for the time interval required for observing the Ramp antipattern. For instance, this value can be derived from typical system run times and workload intensities. The detection techniques described in this section are based on this assumption.

Similar to the Varying Response Times problem, the Ramp is a quite abstract performance problem. For the detection of the Ramp antipattern we instrument the system under test (SUT) at the user interface level capturing the response time for each service request. However, the instrumentation for the root cause analysis is more complex which is described in Section 6.3.2.

In general there are different experiment execution and data analysis approaches which can be applied to detect the Ramp antipattern. We introduce and evaluate three possible approaches. The first two approaches, the *Linear Regression Analysis* and the *T-Test Analysis*, are based on the observation of response times over operation time utilizing a constant workload. For the *Separated Time Windows Analysis* approach, we utilize another experiment execution approach which is described in Section 6.3.1.2.

6.3.1.1. Analyses on Chronologically Continuous Measurement Data

In this subsection, we describe two different analysis techniques which are based on measurement data taken from continuous observation time intervals. First, we describe the experiment execution approach. Then, we introduce the *Linear Regression Analysis* and the *T-Test Analysis* techniques.

Experiment Configuration

For the *Linear Regression Analysis* and the *T-Test Analysis*, we use the same experiment configuration as for the Varying Response Times problem (cf. Section 6.2.1.2). For this purpose, we observe response times of the system service over a fixed operation time interval. We use an open workload examining the response times for a scenario under exponentially increasing workload intensities. Basically, we can reuse the measurement data from the Varying Response Times problem for these analysis techniques avoiding additional experiments. This measurement data contains for each observed workload intensity w_j a set of response time samples. Each sample is a value pair of the operation timestamp T_i and the corresponding response time R_i for a workload intensity w_j : $(T_i, R_i)_j$.

Linear Regression Analysis

A simple method to find an increasing tendency in response times is to use linear regression (cf. Chapter 2.3.4). Applying linear regression on the samples $(T_i, R_i)_j$ yields for each observed workload intensity w_j a regression function $R(t, j) = m_j * t + R_{0,j}$ describing response times in dependence of the operation time. As the factor m_j is the gradient of the curve, positive values for m_j indicate an increase, and thus, the Ramp behaviour. However, linear regression is quite sensitive. In particular, single outliers and small irregularities in the samples $(T_i, R_i)_j$ may lead to positive gradient values although, actually, no Ramp behaviour is present. Thus, we have to determine a threshold m^* for the gradients m_j . Based on this threshold, the analysis technique detects the Ramp antipattern if for all observed workload intensities w_j the gradient exceeds the threshold:

$$\forall j : m_j > m^* \quad (6.8)$$

In order to derive a general threshold m^* , we analyzed the regression functions for all scenarios. For each scenario and observed workload intensity w_j , we calculate the gradient m_j . For *positive scenarios* regarding the Ramp antipattern (Scenario 1, 2 and 10), we are interested in the smallest gradient:

$$m_{min} = \min_j(m_j) \quad (6.9)$$

In order to achieve correct detection decisions in these cases for each observed workload intensity the gradient must exceed the threshold. Thus, the smallest gradient m_{min} determines the upper limit m_u^* for m^* . The opposite applies for the remaining scenarios. Here, m_{min} determines the lower limit m_l^* . As these scenarios do not contain the Ramp antipattern at minimum one workload intensity w_j must exist where the gradient m_j is smaller than the threshold m^* .

In Table 6.2, we depicted the smallest gradients m_{min} for each scenario. For Scenario 3 - 9, these values determine the lower bound for the threshold as these scenarios are negative scenarios for the Ramp antipattern. Generalizing for all scenarios, $\max(m_l^*)$ is the general lower bound for the threshold m^* . For Scenario 1, 2 and 10, $\min(m_u^*)$ is the general upper bound for m^* . In Table 6.2, these values are depicted in bold letters. Thus, we have a general lower bound with the value $11.4 * 10^{-4}$ and a general upper bound with the value $0.91 * 10^{-4}$. However, these values form a contradiction demonstrating that it is not possible to find a general threshold m^* such that the detection error rate of this detection technique is zero. Therefore, the *Linear Regression Analysis* is not suitable for generically detecting the Ramp antipattern.

	\vec{v}_{ramp}	m_{min}	m_l^*	m_u^*
Scenario 1	D	$9.91 * 10^{-4}$	-	$9.91 * 10^{-4}$
Scenario 2	D	$0.91 * 10^{-4}$	-	$0.91 * 10^{-4}$
Scenario 3	N	$0.56 * 10^{-4}$	$0.56 * 10^{-4}$	-
Scenario 4	N	$11.4 * 10^{-4}$	$11.4 * 10^{-4}$	-
Scenario 5	N	$-0.11 * 10^{-4}$	$-0.11 * 10^{-4}$	-
Scenario 6	N	$0.051 * 10^{-4}$	$0.051 * 10^{-4}$	-
Scenario 7	N	$-124.0 * 10^{-4}$	$-124.0 * 10^{-4}$	-
Scenario 8	N	$-1.3 * 10^{-4}$	$-1.3 * 10^{-4}$	-
Scenario 9	N	$-1.1 * 10^{-4}$	$-1.1 * 10^{-4}$	-
Scenario 10	D	$4.8 * 10^{-4}$	-	$4.8 * 10^{-4}$

Table 6.2.: Minimal response time gradients for all scenarios determining the lower and upper bound for the gradient threshold

T-Test Analysis

As a second analysis alternative for the Ramp detection, we introduce the *T-Test Analysis* approach. For this analysis approach, we use the same measurement data as for the *Linear Regression Analysis*. However, instead of analysing the gradient of a regression function, we utilize the t-test (cf. Chapter 2.3) to find an increase tendency in response times. The idea behind this approach is to divide for each workload intensity w_j the samples $S_j = (T_i, R_i)_j$ chronologically into two sets S_j^1 and S_j^2 :

$$S_j^1 = (T_i^1, R_i^1)_j \quad S_j^2 = (T_i^2, R_i^2)_j \quad S_j^1 \cup S_j^2 = S_j \quad T_j^k = \{T_i^k\}_j \quad R_j^k = \{R_i^k\}_j \quad (6.10)$$

$$\forall t^1 \in T^1, t^2 \in T^2 : t^1 < t^2 \quad (6.11)$$

If the measurements under a workload intensity w_j exhibit the Ramp behaviour, the response time values in R_j^2 should be significantly larger than in R_j^1 . For this purpose, we perform a t-test for each scenario and each w_j . Let X_j^1 be the random variable for sample R_j^1 and X_j^2 the random variable for sample R_j^2 . Furthermore, \bar{x}_j^1 and \bar{x}_j^2 are the arithmetical means for both sets. For the t-test, we define the following null hypothesis H_0 and alternative hypothesis H_1 :

$$H_0 : \mu_j^1 = E[X_j^1] = E[X_j^2] = \mu_j^2 \quad (6.12)$$

$$H_1 : \mu_j^1 \neq \mu_j^2 \quad (6.13)$$

If the t-test rejects the null hypothesis the response times in R_j^1 differ statistically significant from the response times in R_j^2 . If additionally \bar{x}_j^1 is smaller than \bar{x}_j^2 , then the values in R_j^2 are significantly larger than in R_j^1 indicating an increasing response time behaviour for the considered scenario and workload intensity w_j . This detection technique detects a Ramp antipattern if for all examined workload intensities w_j can be shown that R_j^2 is statistically larger than R_j^1 .

In order to find a proper significance level for the t-test, we examined this analysis technique for all scenarios with different confidence levels between 0.9 and 0.999. As depicted in Table 6.3, we could not observe any difference in the detection results when varying the confidence level. Cases where the Ramp could be detected are denoted by a D, otherwise the N stands for a negative detection result. The column \vec{v}_{ramp} contains the *expectation*

Scenarios	v_{ramp}	Simple T-Test (sig. level)					Second Half Analysis
		0.9	0.95	0.99	0.995	0.999	
Scenario 1	D	D	D	D	D	D	D
Scenario 2	D	<i>N</i>	<i>N</i>	<i>N</i>	<i>N</i>	<i>N</i>	<i>N</i>
Scenario 3	N	N	N	N	N	N	N
Scenario 4	N	<i>D</i>	<i>D</i>	<i>D</i>	<i>D</i>	<i>D</i>	N
Scenario 5	N	N	N	N	N	N	N
Scenario 6	N	N	N	N	N	N	N
Scenario 7	N	N	N	N	N	N	N
Scenario 8	N	N	N	N	N	N	N
Scenario 9	N	N	N	N	N	N	N
Scenario 10	D	D	D	D	D	D	<i>N</i>
detection error rate	—	0.2	0.2	0.2	0.2	0.2	0.2

Table 6.3.: T-test analyses with different confidence level values, second half analysis.

D: detected, N: not detected, red/italic: wrong decision

vector for the Ramp antipattern. Italic letters stand for wrong detection decisions in *detection vectors*. As can be seen in Table 6.3, using a simple t-test analysis results in wrong decisions for Scenario 2 and Scenario 4. In Scenario 2 no Ramp behaviour could be found, although it contains this antipattern, and vice versa for Scenario 4. In the case of Scenario 4, the initial phase where the fixed sized queue is not completely filled is the reason for wrong detection. Actually, in the second half of time the response times are significantly larger. However, response times do not grow anymore as soon as the saturation point is reached when the queue is filled. If the saturation point is reached late during experiment execution, we are not able to distinguish the classic Ramp behaviour of Scenario 1 from this case in Scenario 4. Thus, we assume that the saturation point is reached in the first half of time during experiment execution. Under this assumption we can overcome this detection problem by analyzing the second set R_j^2 analogously. For this purpose, we split R_j^2 again in two subsets on which we perform the t-test. The results for this extended approach is depicted in the last column of Table 6.3. Now the decision for Scenario 4 is correct, however, we get another wrong decision for Scenario 10. Furthermore, the wrong decision for Scenario 2 remains. In the cases of Scenario 2 and 10, the Ramp cannot be detected because under low load the time interval of observation is too short to observe increasing response times. However, if the Ramp antipattern is present, we assumed an increasing response time behaviour for all considered workload intensities. On the other hand, under heavy load synchronization performance problems (like in Scenario 6 or Scenario 8) exhibit an increasing response time behaviour, too. However, synchronization problems are not the same as the Ramp antipattern.

All detection techniques depicted in Table 6.3 have a detection error rate of 0.2 which is not satisfactory. This error rate is caused by an improper experiment execution approach for the Ramp detection. It is difficult to determine a workload intensity and an observation time window which are suitable for observing the Ramp antipattern. In particular, these values vary greatly from system to system. At the same time, the Ramp is an antipattern whose occurrence is independent from the current workload intensity. Thus, in order to provide an accurate detection technique for the Ramp we have to develop an experiment execution approach which is independent of the workload intensity and allows for finding a proper observation time window. In the following, we introduce such an experiment execution and analysis approach.

6.3.1.2. Separated Time Windows Analysis

For *Linear Regression Analysis* and *T-Test Analysis* we used the same measurement data as for the Varying Response Time problem (cf. Section 6.2) describing for each workload intensity the response time progression over time. For the *Separated Time Windows Analysis* we use another experiment configuration. This experiment configuration is explained in the next paragraph.

Experiment Execution

In contrast to synchronization problems, for the Ramp, the current workload intensity during observation does not have to be high in order to observe the performance problem. However, the workload intensity determines *how fast* response times increase. Higher load pushes the Ramp behaviour faster than a low workload intensity. On the other hand, in order to accurately investigate the Ramp antipattern synchronization effects have to be excluded. Synchronization effects can impair the typical behaviour exhibited by a Ramp. Based on this requirements, we divide an experiment into two phases: a *warm-up phase* and an *observation phase*.

During the warm-up phase, we do not take any measurements. However, we use this phase to push a potential Ramp antipattern. Therefore, we commit a quite high workload intensity to the SUT during the warm-up phase. In particular, we use a workload intensity which brings the SUT to its limit. In Section 6.2.1.2, we described how to find this workload intensity.

During the observation phase we capture a fixed number of response times for the service requests. In this phase, synchronization effects should be avoided. For this purpose, we use a closed workload with only one user and a short think time. This workload guarantees that requests are not processed concurrently, allowing us to exclude synchronization problems.

In order to capture the response time progression over operation time, we repeat this experiment increasing the duration of the warm-up phase. In this way, we get n chronologically separated sets S_i (“Time Windows”) each containing a fixed number of measured response times. As an example we depicted the measurement data for a scenario containing the Ramp antipattern (Scenario 1) in Figure 6.6(a) and for a scenario without the Ramp antipattern (Scenario 3) in Figure 6.6(b). These figures illustrate the dependency between response times and the corresponding warm-up phase durations.

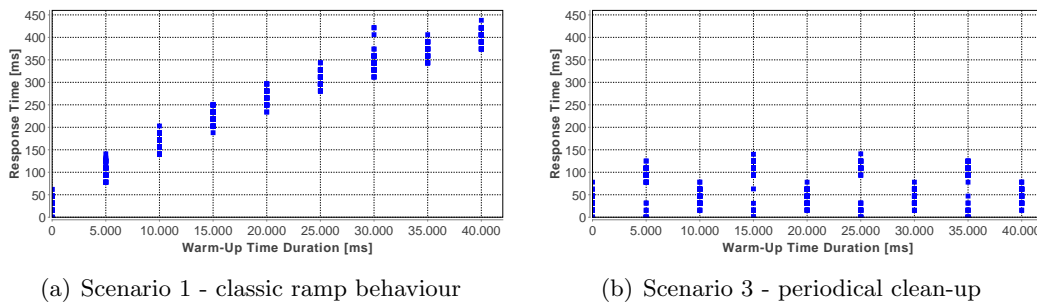


Figure 6.6.: Measurement data for the Separated Time Windows Analysis technique

The described experiment configuration has two advantages. Firstly, we can exclude synchronization problems because of the closed workload containing only one user. Secondly, because of the heavy load during the warm-up phase, we increase the probability that a present Ramp behaviour is captured through measurements.

Analysis Technique

In order to detect an increasing response time behaviour, we perform pairwise t-tests on neighbouring sets. Two sets S_i and S_{i+1} are neighbouring if S_{i+1} is the response time series with the next greater warm-up time in regard to S_i . Let X_i be the random variable for sample S_i and \bar{x}_i its arithmetical mean. For S_1, \dots, S_n , we perform $n - 1$ t-tests T_i using the following samples as input.

$$1 \leq i < n : \begin{cases} \text{sample 1: } S_i \\ \text{sample 2: } S_{i+1} \end{cases} \quad (6.14)$$

The null hypothesis H_0 and the alternative hypothesis H_1 are defined as follows:

$$H_0^i : \mu_i = E[X_i] = E[X_{i+1}] = \mu_{i+1} \quad (6.15)$$

$$H_1^i : \mu_i \neq \mu_{i+1} \quad (6.16)$$

If a t-test T_i rejects H_0^i and $\bar{x}_i < \bar{x}_{i+1}$, then the values in S_{i+1} are significantly larger than in S_i . This means that a measurement series with the next higher warm-up time contains significantly larger response times. If this is the case for all performed t-tests T_i , we consider that response times grow significantly with the operation time.

We applied this technique on all scenarios yielding correct detection decisions in all cases. Thus, with the *Separated Time Windows* approach we found the desired detection technique for the Ramp antipattern with an error rate of zero.

In this section, we investigated different experiment execution approaches and analyses for the detection of the Ramp antipattern. While *Linear Regression Analysis* and *T-Test Analysis* showed some weaknesses resulting in inaccurate detection results, the *Separated Time Windows Analysis* made correct detection decisions for all considered scenarios. In the following, we explain a root cause analysis approach which reuses the *Separated Time Windows Analysis* approach for identifying operations causing the problem.

6.3.2. Root Cause Analysis

As depicted in Figure 6.1, we consider two possible causes for the Ramp antipattern: the Dormant References antipattern or specific methods. As Dormant References is an antipattern, we examine it in Section 6.4. Here, we focus on the search for guilty methods causing the increasing response time behaviour. In the following, we explain the approach for finding root causes for the Ramp antipattern and evaluate this approach. As the Ramp antipattern is present only in Scenario 1, 2 and 10, we use these scenarios for validation of the root cause analysis approach.

6.3.2.1. Realization

In order to find guilty methods, we use a *call tree*. A call tree is a dynamic construct describing the calling hierarchy of single methods. In Listing 1, we fragmentary depicted the call tree for Scenario 10. Scenario 10 combines three other scenarios (Scenario 1, 7 and 8). Therefore, in Scenario 10 all three services of the Online Banking System are invoked. Each service request results in further internal method invocations. For instance, the `commitTransaction(...)` service invokes the `createTransactionObject(...)` method.

A *guilty method* for the Ramp antipattern, is a method within the call tree which exhibits the Ramp behaviour itself. Generally, there are several guilty methods within a call tree. In particular, a guilty method may call other guilty methods which are the actual cause for the

Listing 1 Partly call tree for Scenario 10

```

User.use()*
  +-- OBSystem.viewAccountState(...)
  |   +-- AccountManager.retrieveInformation(...)
  |       +-- ...
  |   +-- ...
  +-- OBSystem.commitTransaction(...)*
  |   +-- TransactionManager.createTransactionObject(...)
  |   +-- TransactionManager.validateTransactionData(...)
  |   +-- TransactionManager.addTransactionToRepository(...)*
  |       |   +-- Repository.checkForDuplicate(...)**
  |       |   +-- Repository.add(...)
  |       +-- TransactionManager.createNotification(...)
  +-- OBSystem.changePersonalData(...)
      +-- ...

```

Ramp behaviour. In our example (cf. Listing 1), guilty methods are denoted by a “*” symbol. As the top level usage profile method `use()` exhibits a Ramp behaviour it is a guilty method. However, this behaviour is caused by a call to the `commitTransaction(...)` method, etc. A *root cause method* is a guilty method which does not invoke any further guilty methods within defined system boundaries. For instance, if the Ramp is actually caused by an external 3rd party operation E, then an operation M is a root cause method if M calls E. Furthermore, we do not dig into the Java API. In Listing 1, we have only one root cause method which is denoted by a “**” symbol: `checkForDuplicate(...)`.

Our root cause analysis approach for the Ramp antipattern is based on a breadth-first search for guilty methods on the corresponding call tree. For this purpose, we recursively perform two main tasks: experiment execution and measurement data analysis. The complete root cause searching approach is depicted in Listing 2. If the Ramp antipattern

Listing 2 Recursive root cause analysis approach

```

1: function FINDROOTCAUSEMETHODS( $m_p$ ,  $M$ )
2:    $C_p \leftarrow \text{DetermineCallChildren}(m_p)$ 
3:    $\text{AdaptInstrumentation}(C_p)$ 
4:    $D \leftarrow \text{ExecuteExperiments}$ 
5:    $G_p \leftarrow \text{SeparatedTimeWindowsAnalysis}(D)$ 
6:   if  $G_p \neq \emptyset$  then
7:     for all  $m_j^g \in G_p$  do
8:        $\text{FindRootCauseMethods}(m_j^g, M)$ 
9:     end for
10:  else
11:     $M \leftarrow M \cup \{m_p\}$ 
12:  end if
13: end function

```

has been detected, we now that the top level method m_{top} of the usage profile exhibits an increasing response time behaviour. In order to find a guilty method we dig into the parent method m_p analyzing the behaviour of all directly invoked *child methods* m_i . We use byte code analysis to find out which methods m_i are invoked by a parent method m_p :

$$m_i \in C_p = \{m | m \text{ invoked by } m_p\} \quad (6.17)$$

Let R be the set of all methods exhibiting a Ramp behaviour. Using this information we are able to dynamically adapt the instrumentation (cf. Chapter 5) by removing monitoring probes from m_p and injecting probes into the child methods m_i . Having instrumented the child methods, we execute the same experiments as for the detection of the Ramp applying the *Separated Time Windows* approach. As measurement data we get for each observed child method m_i chronologically separated sets of response times. On these sets we perform for each m_i the *Separated Time Windows Analysis* selecting guilty child methods m_j^g :

$$m_j^g \in G_p = \{m | m \in C_p \wedge m \in R\} \quad (6.18)$$

For all m_j^g , we recursively repeat all steps until we find a root cause method m_{rc} for which the following applies:

$$C_{rc} = \{m | m \text{ invoked by } m_{rc}\} \quad (6.19)$$

$$G_{rc} = \{m | m \in C_{rc} \wedge m \in R \wedge m \text{ is within system boundaries}\} = \emptyset \quad (6.20)$$

In this way, we get a set M of methods which we suppose to be responsible for the Ramp behaviour observed at the root of the call tree.

Note, if the Ramp antipattern is caused by some external sources, random effects can occur. For instance, methods which do not cause the Ramp behaviour would be identified as the “guilty” ones. However, as we repeat measurement experiments several times the probability to observe such random distortion effects is very low.

6.3.2.2. Evaluation

We applied the described root cause analysis approach on all scenarios containing the Ramp antipattern (Scenario 1, 2 and 10, cf. Figure 6.3). In all cases we were able to find the actual method causing the Ramp behaviour. For instance, in the case of Scenario 10 this approach recognized the `checkForDuplicate(...)` method (cf. Listing 1) as the root cause. In Scenario 2, this approach discovered the `checkForDuplicate(...)` method as a root cause, too. Actually in Scenario 2, the Ramp is caused by a Dormant References antipattern. Nevertheless, the `checkForDuplicate(...)` method is a root cause, too, as this method accesses the data structure which causes the Ramp. Thus, the detection of method `checkForDuplicate(...)` as root cause for the Ramp in Scenario 2 is not a wrong decision.

Listing 3 Iterative root cause analysis approach

```

1: function FINDROOTCAUSEMETHOD( $m_p$ )
2:    $G_p \leftarrow \{m_p\}$ 
3:   while  $G_p \neq \emptyset$  do
4:      $m_p \leftarrow \text{SelectMostPromisingMethod}(G_p)$ 
5:      $C_p \leftarrow \text{DetermineCallChildren}(m_p)$ 
6:      $\text{AdaptInstrumentation}(C_p)$ 
7:      $D \leftarrow \text{ExecuteExperiments}$ 
8:      $G_p \leftarrow \text{SeparatedTimeWindowsAnalysis}(D)$ 
9:   end while
10:  return  $m_p$ 
11: end function

```

In regard to functionality this root cause searching approach works well. However, in order to apply this approach one assumption has to be made. For this approach we assume that repeating experiments with an equal configuration results in equal or at least similar call trees and response time behaviours. Most applications exhibit such stable behaviour.

However, this approach is not suited for applications containing randomized code as in these cases the call tree structure is not stable among individual experiments. Furthermore, there is potential for improvement regarding extra functional attributes. In our scenarios we use a quite simple software system yielding simple call trees during experiment execution. In realistic systems these call trees may become much larger. As the number of required experiments depends on the call tree depth and the degree of branching a breadth-first search is inapplicable in realistic systems. Therefore, we suggest to reject the recursive search and the expectation to find *all* possible root cause methods. A more applicable approach is to search only for the most promising root cause. This would transform the recursive approach from Listing 2 into an iterative approach as depicted in Listing 3. In this case, from the set of guilty child methods G_p only the most promising method is selected for further analysis. In order to realize this selection we use the *p-values* of the t-tests applied during the *Separated Time Windows Analysis*. A small p-value indicates a high probability that the considered samples of a t-test originate from different populations. Thus, the most promising method for the root cause is the method with the smallest p-value as this indicates the greatest increase in response times.

6.3.3. Summary

In this section, we investigated different detection techniques for the Ramp antipattern. Analysis techniques based on chronologically continuous measurement data proved to be not suited for detecting the Ramp. However, abstracting from the workload intensity under observation we introduced the *Separated Time Windows* approach which provided correct detection results for all considered scenarios. Based on this detection technique, we developed an approach for root cause analysis. While the recursive approach finds all possible root causes, the iterative approach searches only for the most promising root cause. However, because of the need for a huge amount of experiments the recursive approach is not applicable in realistic environments.

6.4. The Dormant References Detection

In the previous section, we considered root cause analysis techniques for the Ramp antipattern which are suitable for finding specific methods causing the Ramp behaviour. In this section, we investigate another possible cause for the Ramp which is an antipattern for itself: the Dormant References antipattern.

6.4.1. Experiment Configuration

For the detection of the Dormant References antipattern it is necessary to examine the memory consumption of the target software system. If a software system contains a Dormant References antipattern the memory consumption increases over time. Therefore, for this antipattern we use a similar experiment configuration as for the *Separated Time Windows Analysis* in Section 6.3.1.2. Each experiment is divided into a warm-up phase and a measurement phase. During the warm-up phase we do not take any measurements, however, we use a high workload intensity to push a potential Dormant References antipattern. Equivalently to the Ramp antipattern the Dormant References is an antipattern whose occurrence does not depend on the concurrency level. Moreover, a high concurrency level may impair the typical Dormant References behaviour. Thus, in order to avoid synchronization effects we use a closed workload with only one user for the measurement phase. During the measurement phase we sample the memory consumption of the corresponding process.

6.4.2. Analyzing Memory Consumption

Accurately retrieving the memory consumption of an application is a challenge if the application is written in a programming language based on managed memory (like Java or C#). Automatic, non-deterministic garbage collection makes it difficult to determine the actual memory consumptions. If we retrieve the memory consumption M of the corresponding application process, M does not necessarily correspond to the actually used memory size. M is the sum of actually used memory U plus the size G of garbage objects which have not been collected by the garbage collector yet: $M = U + G$. However, it is not possible to determine U or G , as garbage collection is not deterministic. In particular, it is not possible to invoke the garbage collection explicitly. In java, for instance, we can only suggest the system to use the next opportunity to run garbage collection. We use this service to increase the opportunity for accurate measurements of the memory consumption. Each time we measure the memory consumption, we suggest the system to perform garbage collection. In general, garbage collection influences measured response times as it is a rather time consuming task. During the measurements for the Dormant References antipattern we are not interested in response times. Thus in this case, garbage collection does not influence our measurement results negatively.

Executing the measurement experiments, we get for each warm-up phase duration w_i a sample $M_i = (M_i^1, \dots, M_i^m)$ of memory usages. Let X_i be the random variable for the sample M_i and \bar{M}_i its mean value. In order to detect an increase in memory consumption we execute $n - 1$ t-tests on the samples M_i and M_{i+1} ($1 \leq i < n$) with the following null hypotheses:

$$H_0^i : \mu_i = E[X_i] = E[X_{i+1}] = \mu_{i+1} \quad (6.21)$$

If each t-test rejects the corresponding null hypothesis and $\bar{M}_i < \bar{M}_{i+1}$, we consider that memory consumption increases with the duration w_i of the warm-up phases. Thus, the Dormant References antipattern is detected if the memory consumption increases over time.

We applied this detection approach on all scenarios which yielded the results depicted in Table 6.4. The first column contains the *expectation vector* \vec{v}_{dr} for the Dormant References antipattern. The column “original” contains the detection vector for the ten original sce-

Scenarios	\vec{v}_{dr}	original	KB	MB
Scenario 1	N	N	N	N
Scenario 2	D	<i>N</i>	<i>N</i>	D
Scenario 3	N	N	N	N
Scenario 4	N	N	N	N
Scenario 5	D	<i>N</i>	<i>N</i>	D
Scenario 6	N	N	N	N
Scenario 7	N	N	N	N
Scenario 8	N	N	N	N
Scenario 9	N	N	N	N
Scenario 10	N	N	N	N
detection error rate	—	0.2	0.2	0.0

Table 6.4.: Memory consumption analysis for different magnitudes of memory sizes.

D: detected, N: not detected, red/italic: wrong decision

narios as they were described in Section 6.1.2. For all scenario, the Dormant References antipattern could not be detected. This leads to wrong decisions for Scenario 2 and 5 which both contain the Dormant References antipattern. Although, in Scenario 2 and 5

the corresponding repositories grow over time this circumstances could not be observed in the measurements. Figure 6.7 shows that the measured memory consumption does not increase with the duration of warm-up phases.

The transaction objects stored by the Transaction Manager of the Online Banking system

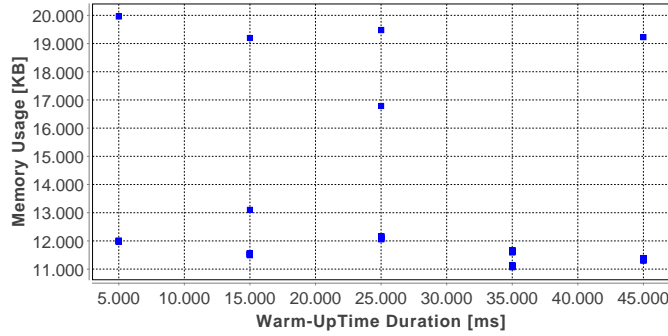
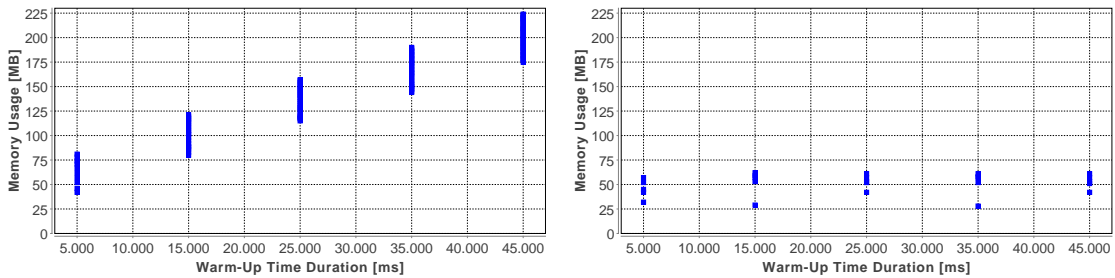


Figure 6.7.: Memory consumption measurements for Scenario 2 (using small transaction objects)

(cf. Section 6.1.2) have a memory size of some bytes. As mentioned before, measuring the memory consumption has only a low accuracy. Even if some thousands of transactions are stored in the repositories of Scenario 2 or Scenario 5, the memory consumption increases only by some kilo bytes. This small increase cannot be captured accurately by measurements. We repeated the experiments with bigger transaction objects. Even using transaction objects of one kilo byte size yields the same wrong detection decisions which is depicted in column “KB” of Table 6.4. Only when using transaction objects with a size within the range of mega bytes the increase of memory consumption could be detected (cf. column “MB” of Table 6.4). In this case the detection error rate is zero. Figure 6.8(a)



(a) Increasing memory consumption in Scenario 2 (b) Constant memory consumption in Scenario 3

Figure 6.8.: Memory consumptions for Scenario 2 and 3 using big transaction objects

shows the increase of memory consumption in Scenario 2 when using big transaction objects. Using big transaction objects in Scenario 3, for instance, does not lead to increasing memory consumption (cf. Figure 6.8(b)).

6.4.3. Evaluation

The analysis technique for the Dormant References antipattern is similar to the one of the Ramp antipattern. The only difference is the data on which the analysis is performed. For the Dormant References antipattern we need accurate measurements of the memory consumption. However as mentioned before, in the scope of programming languages like Java or C# it is difficult to measure memory consumption accurately. Therefore, we were not able to provide a detection technique allowing us to detect small memory increases. However, we developed a detection technique which allows for discovering larger differences

in memory consumption.

Furthermore, we were not able to determine root causes of the Dormant References antipattern. Root causes of the Dormant References antipattern are either specific data structures or single references, which are not used anymore, but which are held in the memory. For the detection of these root causes two steps are required. Firstly, we have to retrieve a list of objects ranked by their memory consumption. Secondly, objects have to be selected which are not used any more. While the first step is theoretically possible, the second step is quite difficult, as semantics are needed to determine which objects will not be used anymore. In practice even the first step is not realizable such that the ranking can be provided within a satisfactory time frame. For these reasons, we were not able to find detection techniques for the root causes of the Dormant References antipattern.

To sum up, accurate detection and root cause analysis for the Dormant References antipattern remain challenges for future work.

6.5. The One Lane Bridge Detection

In this section, we evaluate different detection alternatives for the One Lane Bridge (OLB) antipattern and examine approaches to find the root causes for this antipattern.

6.5.1. Detection

The OLB antipattern is based on the assumption that the application contains a synchronization point which blocks several threads under concurrent execution. Thus, in order to detect an OLB we must examine the portion of time each single thread is blocked during processing of a request. In the following, we investigate different approaches to derive this portion of time and introduce an analysis technique for detecting the OLB antipattern based on this information.

6.5.1.1. Experiment Configuration

For the detection of the Ramp antipattern (cf. Section 6.3) we avoided concurrency during experiment execution in order to eliminate synchronization effects. In contrast, for the examination of the OLB antipattern we are especially interested in blocking and concurrency behaviour. If an application contains an OLB, we expect that the amount of time a user request is stuck at a synchronization point increases with the concurrency level. In order to investigate the dependency between concurrency level and blocking behaviour, we have to control the concurrency level systematically. For this purpose, we use a closed workload for the following experiments. We set the think time for the workload to zero. By this means, we can guarantee a constant concurrency level during a single experiment. Starting with a workload intensity ($w_1 = 1$) of one user, we increase the workload intensity from one experiment to the next by a constant c : $w_{i+1} = w_i + c$. Using this workload we observe different measurement values which depend on the actual detection technique.

6.5.1.2. Direct Blocking Times Analysis

One of the most intuitive ways to examine the OLB is to capture the blocking times of single service requests. The blocking time of a request is the amount of time the corresponding thread is blocked by another thread. In general, a request is blocked if it waits to acquire a passive resource which is utilized by another thread. For the *Direct Blocking Times Analysis*, we capture the blocking time of each user request. For this purpose, we instrument the top level interface with measurements probes which are able to extract the blocking time. While varying the workload intensity w_i , we analyze the dependency between the concurrency level and the blocking times. The synchronized

methods scenario (Scenario 6) is a typical example for the One Lane Bridge antipattern. The mean response times and blocking times for this scenario are depicted in Figure 6.9(a). As one can see, the response times are dominated by the blocking times. In particular,

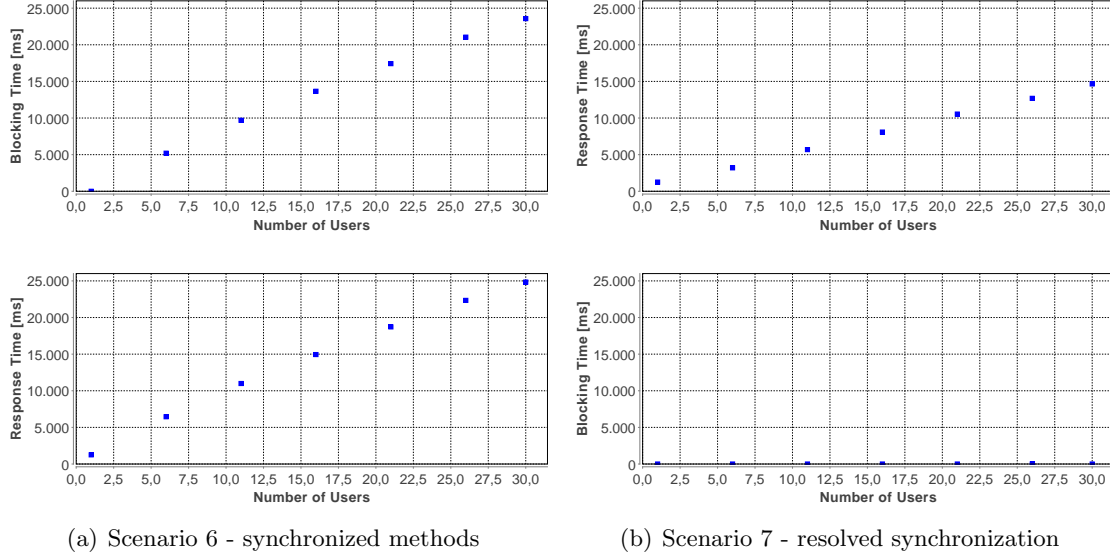


Figure 6.9.: Mean response time and blocking time behaviour for Scenario 6 (positive OLB scenario) and Scenario 7 (negative OLB scenario)

the blocking times increase with the number of concurrent users. In contrast, the blocking times for Scenario 7 (cf. Figure 6.9(b)) do not increase significantly with the number of concurrent users as this scenario does not contain an OLB. These two examples suggest that blocking times increasing with the concurrency level indicate an OLB. This idea can be used to detect this antipattern. Similar to the detection of the Ramp antipattern, we examine whether blocking times increase significantly with the number of concurrent users. For this purpose, we utilize the t-test:

Let w_1, \dots, w_n ($w_{i+1} > w_i$) be the examined workload intensities (in the following called: concurrency levels). For each concurrency level w_i we observe a sample $S_i = B_1, \dots, B_{k_i}$ of blocking times B_j . For each sample S_i let \bar{S}_i be the mean value and X_i the corresponding random variable with the expected value μ_i . On the samples S_1, \dots, S_n , we execute $n - 1$ t-tests with the following samples as input:

$$1 \leq i < n : \begin{cases} \text{sample 1: } S_i \\ \text{sample 2: } S_{i+1} \end{cases} \quad (6.22)$$

For each t-test we define the following hypotheses:

$$H_0^i : \mu_i = E[X_i] = E[X_{i+1}] = \mu_{i+1} \quad (6.23)$$

$$H_1^i : \mu_i \neq \mu_{i+1} \quad (6.24)$$

As significance level for the t-tests we use $\alpha = 0.05$. If all t-tests reject the null hypothesis and $\bar{S}_i < \bar{S}_{i+1}$, we consider that for all observed concurrency levels the blocking times for concurrency level w_{i+1} are significantly greater than for a lower concurrency level w_i . In this case we assume the presence of the OLB antipattern in the examined application. If at minimum one t-test has a non-significant result or $\mu_i \geq \mu_{i+1}$, the OLB is not detected. Applying this analysis technique on all scenarios, we yielded the results depicted in column “Blocking Times Analysis” of Table 6.5. The column “ v_{ob}^{\rightarrow} ” contains the *expectation*

vector for the OLB antipattern. According to Table 6.5, the *Blocking Times Analysis* has a detection error rate of 0.2. Wrong decisions were made for Scenario 8 and 10 which both contain the database manifestation of the One Lane Bridge antipattern. In Figure 6.10, we

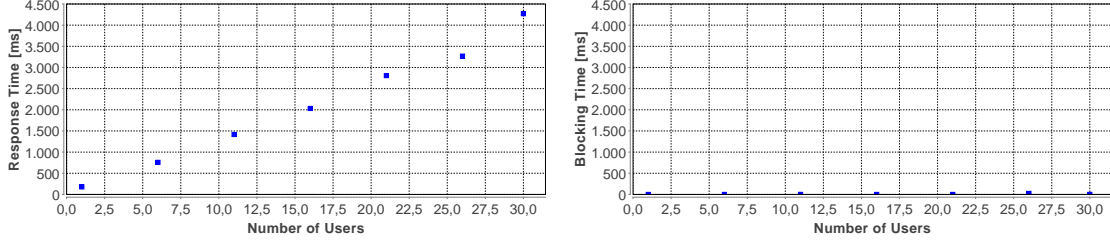


Figure 6.10.: Mean response time and blocking time behaviour of Scenario 8 which contains the database manifestation of the OLB antipattern

depicted mean response times and mean blocking times over the concurrency level for Scenario 8. Although the response times increase significantly, there is no significant growth regarding the blocking times. Moreover, the blocking times are close to zero. Apparently, the blocking times caused by the database lock in Scenario 8 were not captured.

The instrumentation we used for this analysis technique captures the blocking times of all threads at the applications layer. As we deployed the database on a dedicated node, blocking behaviour which occurs on the database node is not captured by the instrumentation probes. Thus, the considered detection technique is suitable for detecting One Lane Bridges of the application layer, however, this technique is not sufficient to detect the database manifestation of this antipattern.

Scenarios	v_{olb}	Blocking Times Analysis	Approx. Blocking Times Analysis	
			naive	advanced
Scenario 1	N	N	<i>D</i>	N
Scenario 2	N	N	<i>D</i>	N
Scenario 3	N	N	<i>D</i>	N
Scenario 4	N	N	<i>D</i>	N
Scenario 5	N	N	N	N
Scenario 6	D	D	D	D
Scenario 7	N	N	<i>D</i>	N
Scenario 8	D	<i>N</i>	D	D
Scenario 9	N	N	<i>D</i>	N
Scenario 10	D	<i>N</i>	D	D
detection error rate	—	0.2	0.6	0.0

Table 6.5.: Comparison of detection techniques for the OLB antipattern.

D: detected, N: not detected, red/italic: wrong decision

6.5.1.3. Approximated Blocking Times Analysis

In order to overcome the problem of the *Direct Blocking Times Analysis*, in this section we use another metric for detecting an OLB. We use the same experiment configuration as before, however, instead of capturing the blocking times, we capture the CPU time and the response time for each service request. The CPU time of a request is the accumulated time the corresponding thread is served by the CPU. Thus, considering an isolated request Q (without concurrency effects) there is the following dependency between the actual

response time R^Q and the CPU time T_{CPU}^Q of the request:

$$R^Q \approx T_{CPU}^Q + W^Q \quad (6.25)$$

Excluding concurrent execution, the waiting time W^Q comprises the amount of time when the request Q is waiting for an event or other circumstances occur preventing Q from being serviced by the CPU. In this trivial case W^Q depends only on Q which allows us to approximate the blocking times B_j (cf. Section 6.5.1.2) with W^Q . Using this approximation, we can calculate for each considered concurrency level w_i the set of waiting times W_i . Under the naive assumption that Equation 6.25 approximates the behaviour under concurrent execution, we apply the *Direct Blocking Times Analysis* (cf. Section 6.5.1.2) on the sets W_i . The detection results are depicted in the column “naive Approximated Blocking Times Analysis” of Table 6.5. The detection error rate for this analysis technique is 0.6. This high error rate is caused by the generous approximation of the blocking times. Equation 6.25 applies only under very low CPU utilization. The situation is more complex under concurrent execution with a higher CPU utilization.

Let (Q_1, \dots, Q_m) be the set of concurrent requests. For the first consideration let us assume that we have only one CPU. As all the requests Q_i share the same CPU, high CPU utilization indicates CPU contention. Thus, we have the following dependency:

$$R^{Q_i} \approx T^* + W^{Q_i} \quad , \quad T^* = \sum_{j=1}^m \left(T_{CPU}^{Q_j}(t_{out}^i) - T_{CPU}^{Q_j}(t_{in}^i) \right) \quad (6.26)$$

Here, R^{Q_i} is the response time of request Q_i and W^{Q_i} the waiting time, respectively. Instead of using the CPU time $T_{CPU}^{Q_i}$ of the considered request Q_i , in Equation 6.26, we consider resource sharing effects by defining T^* . T^* accumulates the CPU demands of all requests which compete for the CPU during execution of Q_i . $T_{CPU}^{Q_j}(t)$ is the CPU time of request Q_j at the timestamp t , while t_{in}^i and t_{out}^i enclose the execution time interval of Q_i . In particular, the following applies:

$$R^{Q_i} = t_{out}^i - t_{in}^i \quad (6.27)$$

In general, T^* is the CPU time consumption within the interval $[t_{in}^i, t_{out}^i]$ of the application process P containing all request threads Q_j .

If there is a number of n CPUs, T^* contains the sum for all CPUs. Therefore, we have to adopt the Equation 6.26 as follows:

$$R^{Q_i} \approx \frac{T^*}{n} + W^{Q_i} \quad (6.28)$$

For the realization of the *Advanced Approximated Blocking Times Analysis* technique, we capture for each request Q its response time and the CPU consumption of the parent process during the execution of Q . According to Equation 6.28, for each concurrency level w_i we calculate the sets of waiting times W_i . Again, we approximate the blocking times with the waiting times W_i on which we apply the *Blocking Times Analysis*. The results for this analysis technique are depicted in the last column of Table 6.5. As one can see, the *detection vector* of the *Advanced Approximated Blocking Times Analysis* technique is equal to the *expectation vector* for the OLB antipattern resulting in a detection error rate of zero.

In the following, we examine different techniques for finding root causes of the One Lane Bridge.

6.5.2. Root Cause Analysis

According to Figure 6.1, we investigate two possible root causes for the OLB antipattern. In the following we introduce a root cause analysis technique for each root cause.

6.5.2.1. Synchronized Methods Root Cause

In this subsection we introduce the *Synchronized Methods Root Cause Analysis (SMRCA)* which is responsible to identified synchronized methods manifestations of the OLB antipattern. This manifestation of the One Lane Bridge occurs if one or more synchronized methods at the application layer cause long queues and great blocking times. More precisely, in this case the actual root cause is not the synchronized method itself but the object for which the method acquires a lock. In order to detect these objects, we use the same experiment configuration as for the detection of the OLB antipattern. The idea is to perform a similar analysis technique as in Section 6.5.1 on all synchronized methods. Thus, instead of capturing (or approximating) the blocking times for the whole service request we have to capture the blocking times for each object which is locked by a synchronized method during service execution. Because of technical reasons it is difficult to capture directly the blocking times for a synchronized method. For this purpose we would have to find all calls to a synchronized method by performing static code analysis. However, this task is too expensive. In contrast, capturing the service times, the queue length and the throughput of a synchronized method entails only a low overhead. Therefore, we instrument each synchronized method with a service time, throughput and a queue length probe. Executing the experiments according to the experiment configuration described in Section 6.5.1.1 yields for each concurrency level (or workload intensity) w_i a mean throughput X , a sample of queue lengths (Q_1, \dots, Q_k) and a sample of service times (S_1, \dots, S_l) . Based on this data we use Little's Law (cf. Chapter 2.4) to derive for each concurrency level w_i the mean blocking time B of the considered object. For the application of Little's Law we consider the locked object and all synchronized methods locking this object as one system. Let us assume we have an object which is locked only by one synchronized method m . In this case the situation is quite simple as schematically depicted in Figure 6.11. We have

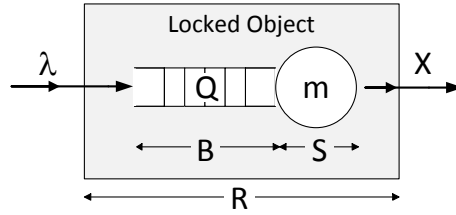


Figure 6.11.: Object locked by one method.

one server representing method m and one queue with a mean queue length Q . As we use a closed workload for measurements, considered over a longer period of time we have a steady system state such that the mean arrival rate λ is equal to the mean throughput X . The mean number of requests within the considered system is the queue length plus one request which is serviced by m : $N = Q + 1$. The response time of one request is the blocking time plus the service time: $R = B + S$. From the measurements we get the mean throughput X , the mean queue length Q and the mean service time S . Applying Little's Law we calculate the mean blocking time B :

$$R = \frac{N}{X} \quad || R = B + S, N = Q + 1 \quad (6.29)$$

$$B = \frac{Q + 1}{X} - S \quad (6.30)$$

If an object is locked by multiple synchronized methods, the situation is more complex. This general case is depicted in Figure 6.12. As n synchronized methods lock the object,

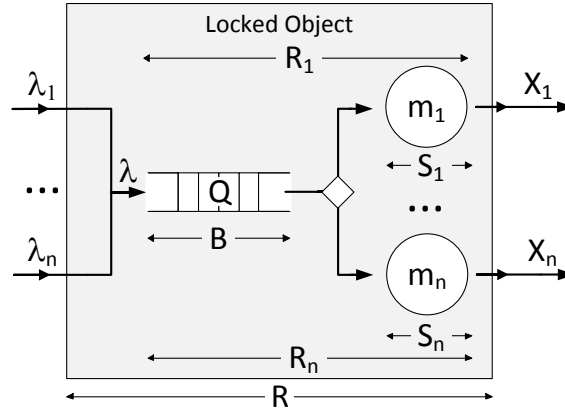


Figure 6.12.: Object locked by multiple methods.

there are n servers (m_1, \dots, m_n) representing these methods. At an arbitrary point of time *only one* server is allowed to serve a request which is illustrated by the rhombus in Figure 6.12. As all methods m_i lock the same object only one queue exists for the considered object. Although each method m_i has an individual arrival rate λ_i , all requests have to pass the same queue with a mean queue length Q and a mean blocking time B . Consequently, the mean response time of a method m_i is the mean blocking time plus the mean service time of m_i : $R_i = B + S_i$. In order to apply Little's Law on this system, we have to abstract from individual methods m_i to get a model as depicted in Figure 6.11. For this purpose, we aggregate values which depend on single methods m_i :

$$\lambda = \sum_{i=1}^n \lambda_i \approx \sum_{i=1}^n X_i = X \quad (6.31)$$

$$S = \frac{1}{n} \sum_{i=1}^n S_i \quad (6.32)$$

$$R = \frac{1}{n} \sum_{i=1}^n R_i \quad (6.33)$$

Using these aggregated values we can apply Little's Law to derive the mean blocking time B :

$$R = \frac{N}{X} \quad (6.34)$$

$$\frac{1}{n} \sum_{i=1}^n R_i = \frac{Q + 1}{X} \quad (6.35)$$

$$\frac{1}{n} \sum_{i=1}^n B + S_i = \frac{Q + 1}{\sum_{i=1}^n X_i} \quad (6.36)$$

$$B = \frac{Q + 1}{\sum_{i=1}^n X_i} - \frac{1}{n} \sum_{i=1}^n S_i \quad (6.37)$$

Applying Equation 6.37 on the measurement data for each concurrency level w_j yields a set of mean blocking times B_j . Similar to the *Blocking Time Analysis* in Section 6.5.1, we compare the mean blocking times pairwise. As B_j are absolute values and not samples of

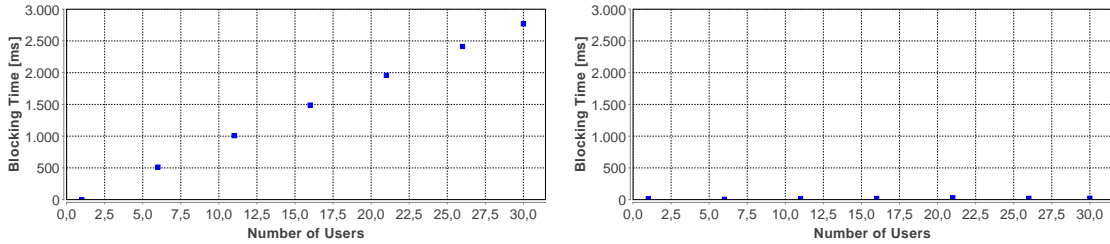
blocking times, we cannot apply the t-test to compare B_j with B_{j+1} . Therefore, we perform an absolute value comparison taking into account a small offset ϵ . More precisely, we assume the considered locked object to be the root cause for the detected OLB antipattern, if:

$$\forall 1 \leq j < n : B_{j+1} > B_j + \epsilon \quad (6.38)$$

Thus, if the blocking times at a locked object O increase with the concurrency level, O is a possible root cause for the detected OLB.

We applied this root cause analysis step on all scenarios which contain an One lane Bridge (Scenario 6, 8 and 10). Scenario 6 contains a synchronized methods manifestation of the One Lane Bridge. Thus we expect the *SMRCA* to find a locked object which is responsible for the OLB. Both, Scenario 8 and Scenario 10 contain an OLB, however, the root cause in this case is a locked database table. Therefore, we expect that the *SMRCA* does not find any locked objects which cause the OLB.

Actually, during the execution of Scenario 8 no synchronized methods are invoked. Thus, *SMRCA* did not detect any root causes. Scenario 10 contains some synchronized methods, however, the blocking times of these methods are so small that the corresponding locked object does not cause an OLB under the considered workloads. Figure 6.13(b) shows the mean blocking times for Scenario 10 which do not monotonously increase with the concurrency level. Consequently, *SMRCA* does not identify the corresponding blocked object as the root cause for the OLB in Scenario 10. In Scenario 6, the *viewAccountState*



(a) Scenario 6:

synchronized methods manifestation of the OLB

(b) Scenario 10:

database lock manifestation of the OLB

Figure 6.13.: Mean blocking times for the corresponding locked objects of Scenario 6 and Scenario 10

service is a synchronized method (cf. Section 6.1.2) locking the corresponding *Account Manager* object. Thus, the implementation of the *Account Manager* in Scenario 6 is the root cause for the occurring OLB. Consequently, the blocking times caused by the *Account Manager* increase strictly with increasing concurrency level as depicted in Figure 6.13(a). The analysis technique *SMRCA* recognizes these circumstances and identifies the *Account Manager* instance as a possible root cause for the OLB antipattern in Scenario 6.

6.5.2.2. Database Lock Root Cause

In this subsection we introduce the *Database Lock Root Cause Analysis (DBRCA)* intended to find database locks which cause an One Lane Bridge. In general, we again can apply the *Blocking Times Analysis* technique to detect the database root cause of the OLB. However, instead of observing or calculating the blocking times of single synchronized methods, we have to determine the blocking times of individual database accesses. As we consider the database as a black-box it is not possible to capture the blocking times of database requests. In particular, we can not distinguish requests which are blocked within the database from requests which are performed concurrently by the database application. However, it requires only little effort to measure the response times of database

accesses. For this purpose, we instrument all classes which implement certain database access interfaces with response time probes. For instance in the case of Java environments paired with usage of JDBC, we dynamically instrument all implementations of the JDBC interface. We execute the same experiment series as before varying the concurrency level from experiment to experiment. The experiments provide for each concurrency level w_i and each database access operation a response time sample R_i . Let X_i be the random variable for the sample R_i . As done with blocking times in Section 6.5.1.2, we perform $n - 1$ pairwise t-tests on the response time samples R_i and R_{i+1} with the following null hypothesis:

$$H_0^i : \mu_i = E[X_i] = E[X_{i+1}] = \mu_{i+1} \quad (6.39)$$

If for all $1 \leq i < n$ the t-tests reject the corresponding null hypothesis and $R_{i+1}^- > \bar{R}_i$, we consider that response times increase significantly with the concurrency level. The consequence of this discovery is that the database is some kind of bottleneck during the execution of the target software system. In general, there might be two possible reasons why a database becomes a bottleneck. Either the bottleneck is caused by a passive resource, for instance a database lock, or an active resource is the bottleneck. In the latter case, the corresponding active resource (for instance a CPU) is highly utilized. High CPU utilization on the database node could be the consequence of bad resource planning or other performance antipatterns like the Stifle (cf. Chapter 2.1.3.3). In contrast, a bottleneck caused by a passive resource is a classic OLB antipattern which does not compulsorily lead to high (hardware) resource utilizations. Thus, in order to distinguish these two cases we additionally monitor the CPU utilization of the database node. If for any considered concurrency level the mean utilization of at minimum one CPU on the database node exceeds a threshold $Th_{CPU} = 90\%$, we assume a CPU bottleneck. If the CPU utilization is low for all considered concurrency levels and the database access response times increase strictly, we assume a database lock manifestation of the OLB antipattern.

We applied the DBRCA technique on all scenarios containing an OLB antipattern. Although Scenario 6 contains database accesses, these database requests do not cause an OLB. Consequently, the response times of these requests do not increase with the concurrency level. Therefore, the DBRCA technique did not find any database lock which could be the root cause for the observed OLB. In the cases of Scenario 8 and 10, long SQL *update* statements lock the target database table for long time intervals causing a typical One Lane Bridge. Accordingly, the response times of database requests increase strictly with the concurrency level while the CPU utilization on the database node is quite low (less than 20%). The measurement results for these both scenarios are depicted in Figure 6.14. In both cases, the DBRCA technique recognized these circumstances as indicators for a database lock which causes the observed OLB.

In order to investigate the case when the CPU of the database node becomes a bottleneck, we modified Scenario 7. As described in Section 6.1.2 the `viewAccountState` service of the Online Banking system makes requests to the database to retrieve information. In Scenario 7, these requests were quite simple such that the database is utilized to a quite low degree. Modifying these database requests to quite complex SQL queries increases the database utilization significantly. In the following, we call the modified scenario: Scenario 7*. This modification makes Scenario 7* to a *positive scenario* for the Varying Response Times problem and the One Lane Bridge. Applying the DBRCA technique on Scenario 7* yields the results depicted in Figure 6.15. Similar to Scenario 8 and 10, the response times increase steadily with the concurrency level. However, the CPU utilization of the database node exceeds 90% for a concurrency level greater than 5 users. In this scenario, the DBRCA technique recognizes a database CPU bottleneck. Thus, the overload of the database CPU is the reason why response times increase with the concurrency level.

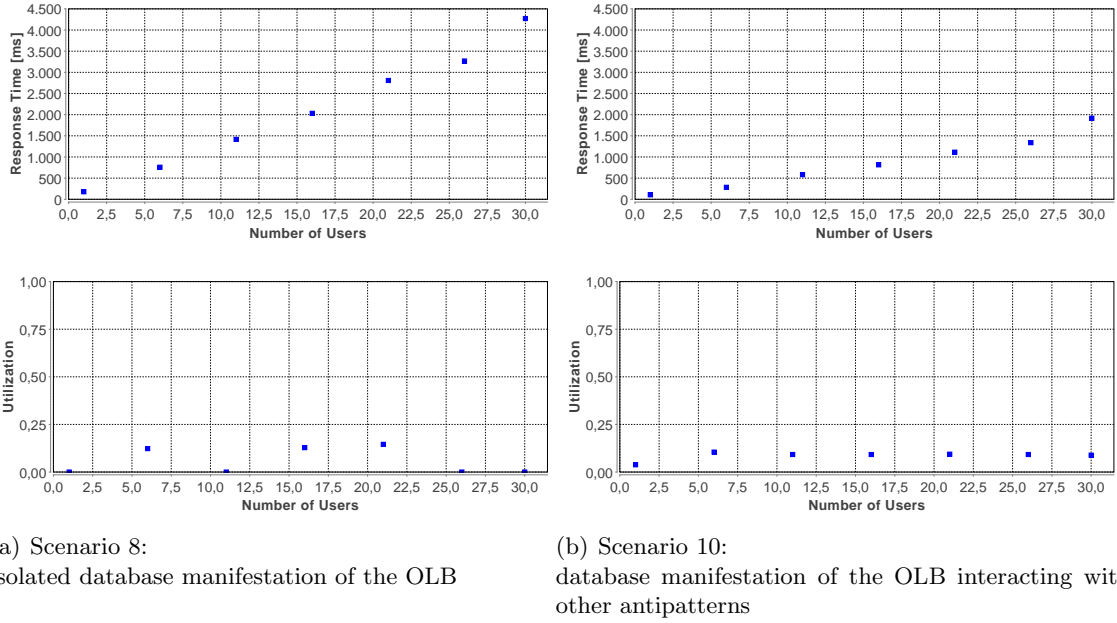


Figure 6.14.: Mean response times and database CPU utilization for scenarios containing the database manifestation of the OLB antipattern.

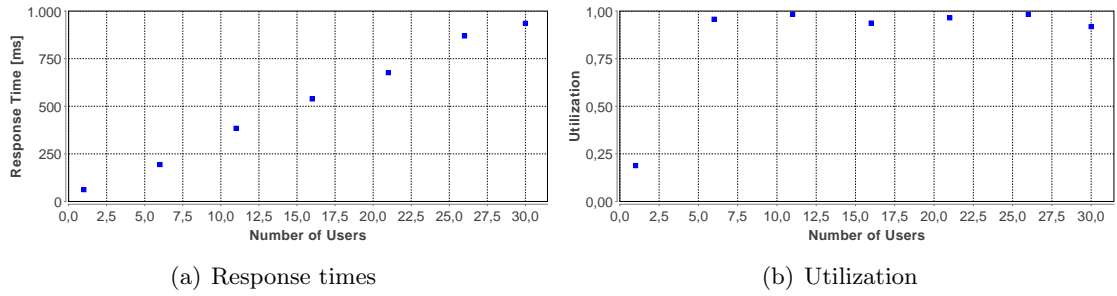


Figure 6.15.: Mean response times and database CPU utilization for Scenario 7*

6.5.3. Summary

In this section, we introduced two detection techniques for the One Lane Bridge antipattern. Both detection techniques are based on the analysis of blocking times in dependence of the concurrency level. However, the detection techniques differ in the way blocking times are measured or derived from other metrics. For both possible root causes of the One Lane Bridge we introduced analysis techniques which are based on the same experiment configurations as the detection part. In order to find the root causes we investigate blocking times and response times for methods and requests on a more detailed level. Using Little's Law we derived the blocking times of synchronized methods. Database manifestations of the One Lane Bridge are discovered by analysing the response times of database requests paired with an analysis of the CPU utilization of the database node. In this way, we are able to distinguish database locks from database CPU bottlenecks.

7. Evaluation

In this chapter, we evaluate the described antipattern detection approach on a more realistic software system. For this purpose, we examine a Java implementation of the TPC-W Benchmark [Tra02]. In Section 7.1, we briefly describe main aspects of the TPC-W Benchmark, including the system under test, the workload specification and a solution model. In Section 7.2, we describe the experiment setup for evaluation. Finally in Section 7.3, we present and discuss the results of the antipattern detection approach applied on the TPC-W scenario.

7.1. TPC-W Benchmark

In this section, we introduce the TPC-W Benchmark following the descriptions in [Tra02] and [Men02]. In general, a benchmark is a specification of a software system which is used to compare different hardware configurations, environment setups or implementations. The Transaction Performance Processing Council (TPC) [tpc] is one of the most important organisations providing benchmarks for databases and transaction processing solutions. TPC-W [Tra02] is a benchmark for investigating the scalability of web commerce solutions. This includes the evaluation of used database, application implementation, web infrastructure as well as underlying hardware. In general, a benchmark specifies three main aspects. Firstly, a benchmark describes the architecture and dynamic aspects of a system under test (SUT). The SUT specified in TPC-W is intended to emulate the performance behaviour of a web commerce application as realistic as possible. Secondly, a workload has to be specified which is submitted to the SUT. Finally, performance metrics are specified, which are used to compare different alternatives.

In this thesis, we do not use the TPC-W Benchmark for comparing alternatives of web commerce solutions. However, we use the SUT specification of the TPC-W Benchmark as a realistic scenario for evaluation of our antipattern detection approach. Therefore, we are not interested in the performance metrics specified by the TPC-W Benchmark. For this reason, we abstain from introducing the performance metrics specified within TPC-W. In the following, we introduce only the TPC-W specification of the SUT and the corresponding workloads.

7.1.1. System Under Test: Bookstore Emulator

The TPC-W Benchmark provides a specification for an emulator of a bookstore application. This bookstore application serves as a representative for realistic web commerce applications. Like most web commerce solutions, the bookstore application comprises three layers: a web layer, an application layer and a data layer. On the web layer, the bookstore application provides several services comprising the following user interactions:

- **Home:** Generates and shows the home web page of the bookstore which serves as the common navigation point. All user activities start with this interaction.
- **Shopping Cart:** Used to update the state of the cart by adding new items to the cart or updating the state of existing items.
- **Customer Registration:** Allows a user to register as a new customer or to authenticate as a known customer. This action is a precondition for a buy request.
- **Buy Request:** Generates a web page summarizing all selected items and providing input fields for entering billing information.
- **Buy Confirm:** Generates a new order from the current cart state and shows a purchase confirmation web page. During purchase processing the bookstore accesses an *external* system (the *Payment Gateway*) for payment authorization.
- **Order Inquiry:** Generates and shows a web page containing input fields for identifying a customer in order to inquire the last order of the corresponding customer.
- **Order Display:** Retrieves information about the last order of the identified customer and shows this information on a generated web page.
- **Search Request:** Generates and shows a web page containing input fields for submitting a search request.
- **Search Result:** Performs a search on items in the bookstore according to entered key words in the search input field of the search request web page. Shows the search result on a generated web page.
- **New Products:** Generates and shows a web page containing a list of new items in the bookstore.
- **Best Sellers:** Generates and shows a web page containing a list of best selling items in the bookstore.
- **Product Detail:** Retrieves and shows detailed information about a selected product.
- **Admin Request:** Allows a bookstore administrator to update items of the bookstore.
- **Admin Confirm:** Executes an item update request of an administrator and shows the result.

These interactions are described in more detail in [Tra02]. Each user interaction triggers the execution of application code which accesses a transactional database on the data layer in order to retrieve and update customer or ordering information. The TPC-W Benchmark specification prescribes the implementation of a minimum of eight database tables containing information about customers, addresses, countries, orders, order lines, items, authors and credit card transactions. The corresponding database schemas are described in [Tra02]. Furthermore, in [Tra02] TPC prescribes rules for implementing and realising the bookstore. Beyond the scope of these rules, it is up to the TPC-W solution provider how the bookstore is implemented.

7.1.2. Workload Specification

In the previous section we listed the services provided by the bookstore to the system users. The usage of these services is described by the workload specification which is an important part of the TPC-W Benchmark. A workload specification describes two things: the user behaviour and the workload intensity. In the TPC-W specification the user behaviour is described by a Customer Behaviour Model Graph (CBMG) [Men02]. A CBMG is a Markov Model describing for each two interactions A and B the probability $p_{A \rightarrow B}$ that an user performs interaction B directly after interaction A . Figure 7.1 shows

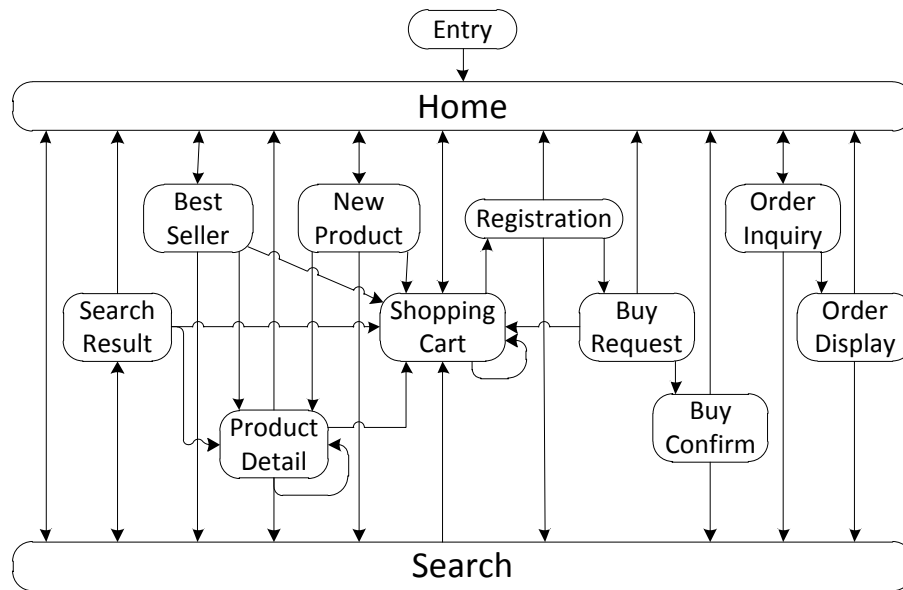


Figure 7.1.: Customer Behaviour Model Graph (following [Men02])

possible transitions between the services provided by the bookstore. Depending on the workload mix the arcs can be annotated with different probabilities. TPC divides the bookstore user interactions into two categories:

- **Browse:** Home, New Products, Best Sellers, Product Detail, Search Request and Search results
- **Order:** Shopping Cart, Customer Registration, Buy Request, Buy Confirm, Order Inquiry, Order Display, Admin Request and Admin Confirm

In [Tra02] three workload mixes are defined. The *Browsing Mix* is specified by 95% browsing actions and only 5% ordering actions. A *Shopping Mix* comprises 80% browsing and 20% ordering. Finally, the *Ordering Mix* consists of 50% browsing and 50% ordering. The TPC-W Benchmark prescribes a closed workload whereby the workload intensity is defined by a number of concurrent users and a think time (cf. Chapter 2.4). Each user is represented by an *Emulated Browser* which communicates with the SUT by sending and receiving HTTP requests. Emulated Browsers follow the Customer Behaviour Model performing interactions with the specified probabilities. Emulated Browsers are executed by Remote Browser Emulators (RBE), whereby one RBE can contain a set of Emulated Browsers. The workload submitted to the SUT is the set of all RBEs.

7.1.3. TPC-W Solution Model

As the TPC-W Benchmark specification provides many degrees of freedom for implementing and realizing the specified scenario, there is no common implementation of the TPC-W Benchmark. Here, we describe a typical model of a TPC-W solution following [Smi00]. However, individual implementations can deviate from this model. Figure 7.2 shows a model of a valid TPC-W solution. The overall setup consists of two parts, the SUT and

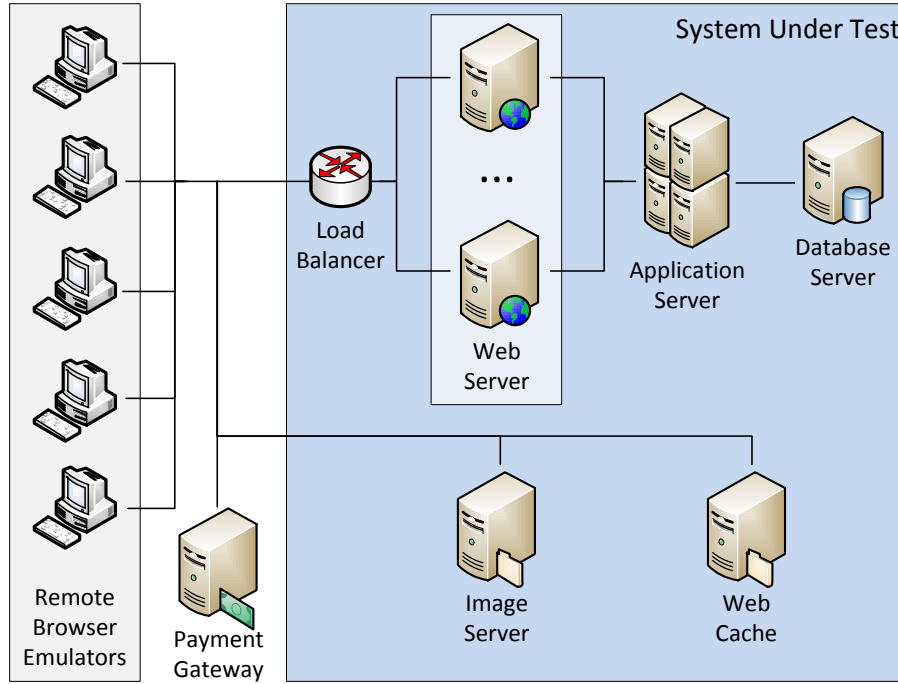


Figure 7.2.: Typical environment configuration for TPC-W (following [Smi00])

aspects which do not belong to the SUT. The layers of the web commerce application can be mapped to three tiers: a set of *Web Servers*, a cluster of *Application Servers* and a *Database Server*. As the TPC-W specification does not prescribe a specific server configuration, this distribution of the layers is not mandatory for realizing the TPC-W Benchmark. For instance, deploying the three layers on one server is a valid solution, too. Furthermore, the *Image Server* and the *Web Cache* are optional parts of the SUT. As mentioned before, the *Payment Gateway* is not a part of the SUT, but is an external system which is accessed by the Web Servers in order to perform payment authorization. The Remote Browser Emulators can be distributed among several devices. In the case of multiple Web Servers, the RBEs communicate with the Web Servers through a *Load Balancer*. The Web Servers interpret HTML requests from the Emulated Browsers and delegate the requests to the Application Server which communicates with the Database Server. In order to retrieve images the RBEs access the Image Server directly using references from the Web Servers.

In the following, we describe the setup we used for evaluation of the antipattern detection approach.

7.2. Experiment Setup

As mentioned before, we use the TPC-W Benchmark not for comparing alternatives but as a quasi-realistic scenario for evaluating our detection approach. For this purpose, we use a Java implementation (from [jav]) of the TPC-W Benchmark. In this implementation the bookstore is implemented using Java Servlets. Each user interaction is represented

by a Java Servlet. The Servlets interpret user requests, access directly the database to retrieve data and create HTML responses. For the management of database connections a connection pool is used which is accessed by the Servlets to acquire and release database connections. We use *MySQL Community Server 5.5.22* [mys] as database software for the bookstore application. As web server and application server we use *Apache Tomcat 6.0.35* [tom]. The Image Server, Web Cache and Payment Gateway are not implemented in the used realization. In Figure 7.3, we depicted the server configuration and distribution of the SUT as we use it for evaluation.

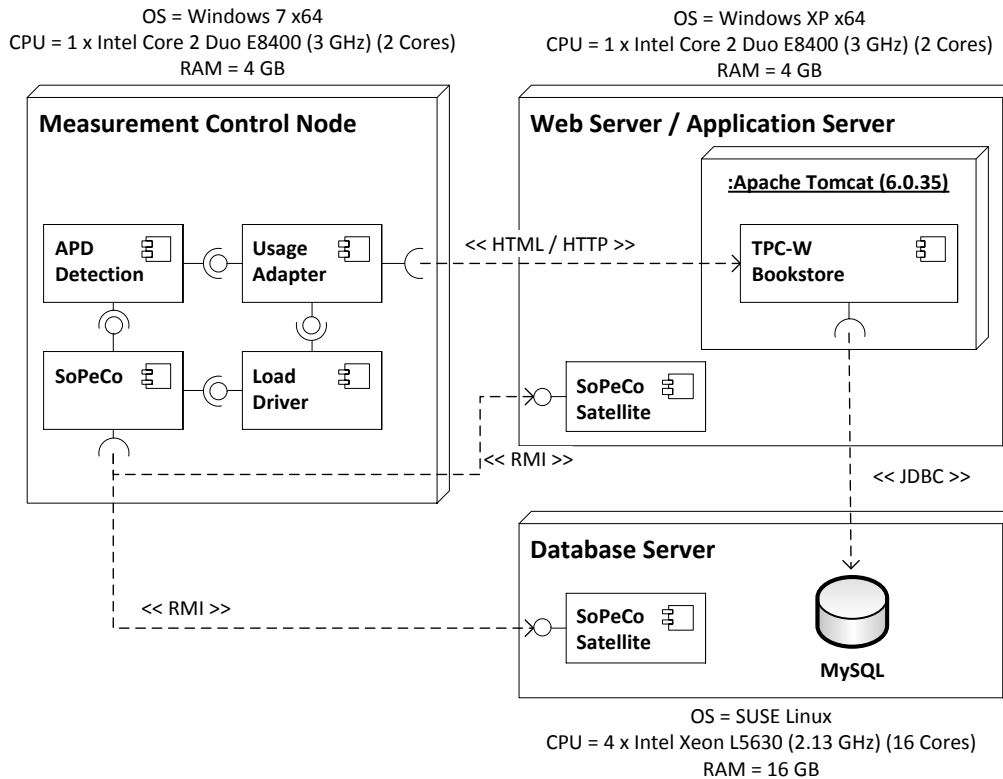


Figure 7.3.: Experiment setup for evaluation of the antipattern detection approach

The MySQL database is deployed on a SUSE Linux Server with 16 CPU cores and 16 GB RAM. Running on a Windows XP machine with 2 CPU cores and 4 GB RAM, the Apache Tomcat instance executes the TPC-W bookstore application. The application communicates with the database over the Java Database Connectivity (JDBC) interface. On both nodes SoPeCo Satellites are deployed which are responsible for instrumenting the SUT and gathering measurement data. The satellite on the database server purely monitors the utilization of hardware resources. The satellite on the Web Server additionally performs dynamic instrumentation of the bookstore application code. The Measurement Control Node is a Windows 7 machine with 2 CPU cores and 4 GB RAM running a SoPeCo instance (cf. Chapter 2.5), the APD Detection extension, the Load Driver and the Usage Adapter (cf. Chapter 4.3). SoPeCo communicates over RMI with the satellites in order trigger dynamic instrumentation and collecting measurement data after experiment execution. The Usage Adapter realizes a simplified form of the workload described in Section 7.1.2. The communication with the bookstore effects via HTTP by sending HTML requests and receiving responses.

As we perform systematic measurement experiments, we are less interested in a realistic workload mix. We are rather interested in the possibility to control the workload which is submitted to the SUT. Therefore, we do not utilize the Remote Browser Emulator

implementation of the used TPC-W solution. Within the Usage Adapter we define an own workload. In Section 7.1.2, the workload was defined by a Markov Model using probability values for individual interaction transitions. For antipattern detection, we define a workload in form of a sequence diagram yielding a fix workload mix which does not change over time. The user interaction sequence implemented by the Usage Adapter is depicted in Figure 7.4. Within an interaction scenario, a user visits the *Home* web

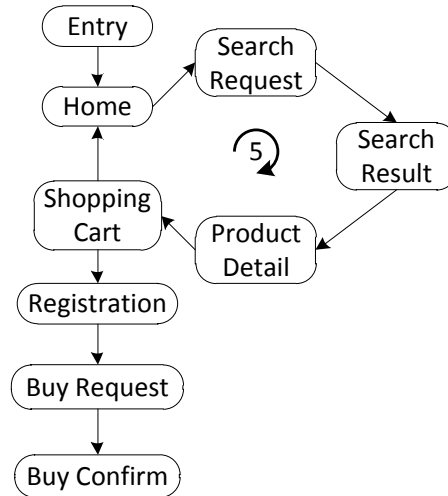


Figure 7.4.: Sequence diagram describing the workload for evaluation of the antipattern detection approach

page, searches for books, requests the product details for a found book and adds the corresponding product to the cart. This sequence is repeated five times before the user, finally, registers as a customer and places an order for the five books in the cart. According to the workload classification defined within the TPC-W specification (cf. Section 7.1.2) this workload sequence comprises 20 browsing interactions and 8 ordering interactions. With an ordering proportion of 28.6% our workload definition is between a Shopping Mix and an Ordering Mix [Tra02]. Although we define the interaction sequence, we do not determine workload type and workload intensity in advance. These values are defined for each detection step individually (cf. Chapter 6) and the corresponding workload is generated by the Load Driver (cf. Figure 7.3).

Using this experiment setup, we applied the antipattern detection approach described in this thesis on the TPC-W Benchmark. In the following, we present the results in detail.

7.3. Results

In this section, we present the antipattern detection results for the TPC-W solution described in the previous section. The detection approach discovered the Varying Response Times and the One Lane Bridge antipatterns in the used TPC-W solution. For the Ramp antipattern the detection result was negative. Consequently, the Dormant References antipattern has not been examined anymore as it is a hierarchical successor of the Ramp antipattern (cf. Figure 6.1). In the following, we consider the detection results in more detail.

7.3.1. The Varying Response Times Problem

For the analysis of the Varying Response Times (VRT) problem, response times have been measured for arrival rates between $1s^{-1}$ and $32s^{-1}$. In Figure 7.5 the response times are depicted in dependence on the arrival rate. Measurements have been stopped after an

arrival rate of $32s^{-1}$ as response times increased greatly under this workload intensity (cf. Chapter 6.2.1.2). As can be seen in Figure 7.5 response times are quite small and have low

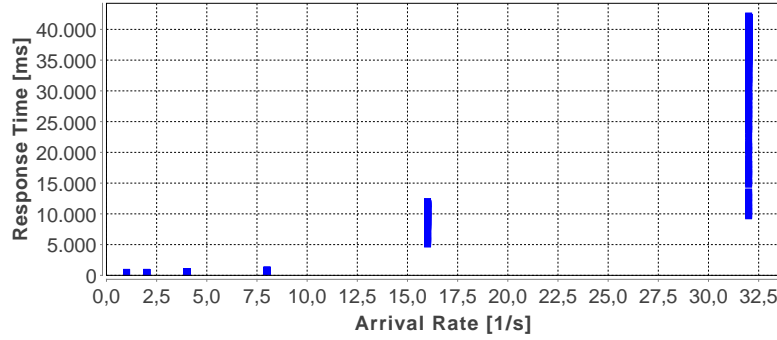
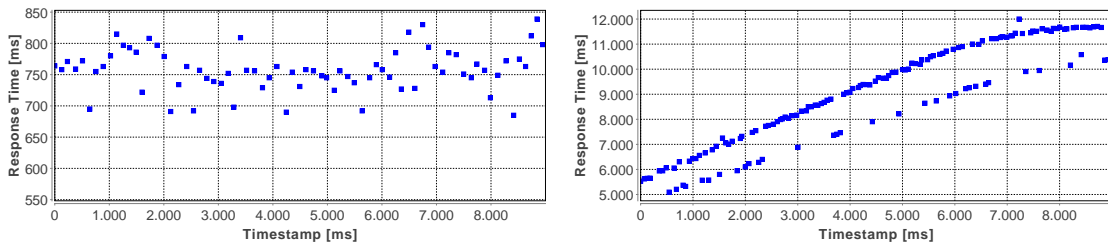


Figure 7.5.: Response times in dependence on the arrival rate



(a) Response time progression for arrival rate $8s^{-1}$ (b) Response time progression for arrival rate $16s^{-1}$

Figure 7.6.: Response times depicted over execution time for arrival rates $8s^{-1}$ and $16s^{-1}$

variance for arrival rates up to $8s^{-1}$. However, for arrival rates about $16s^{-1}$ or $32s^{-1}$ the response time values and their range increase greatly. Thus, we have reason to assume that the SUT runs into an unsteady state under arrival rates greater or equal $16s^{-1}$. In Figure 7.6, we depicted the response time progression over execution time for arrival rates $8s^{-1}$ and $16s^{-1}$. Under an arrival rate of $8s^{-1}$ the response times steadily vary between 650 ms and 850 ms indicating a steady state. In contrast, the response times increase permanently under an arrival rate of $16s^{-1}$. As response times steadily increase under high load but not under low load it is likely that the considered response time behaviour is caused by some kind of bottleneck. Actually, the detection approach discovers the VRT problem

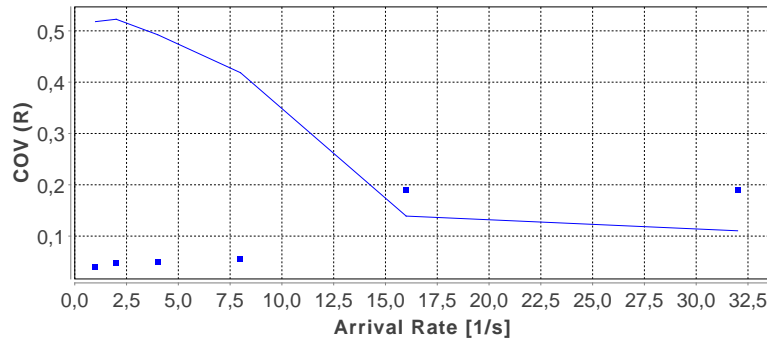


Figure 7.7.: COV for measured response times (points), calculated threshold for COV (curve)

because values for the coefficient of variance (COV) exceed the calculated threshold (cf. Chapter 6.2). Figure 7.7 shows the COV values for the measured response times. The

calculated threshold for COV is illustrated by the curve in Figure 7.7. As response times increase greatly for arrival rates $8s^{-1}$ and $16s^{-1}$ the adapted COV threshold (cf. Chapter 6.2.2.2) decreases rapidly, falling below the COV values of the corresponding arrival rates. Consequently, the detection technique for the VRT problem discovers the arrival rates for which the SUT presumably is in an unsteady state. As the VRT problem has been detected the detection process proceeded with investigating the Ramp and the One Lane Bridge antipattern according to the performance problem hierarchy (cf. Figure 6.1 in Chapter 6).

7.3.2. The Ramp

As mentioned before, the Ramp antipattern was not discovered by the detection approach. Figure 7.8 shows the measurement data from experiments according to the Separated Time Windows Analysis (cf. Chapter 6.3.1.2). For single measurement time windows the

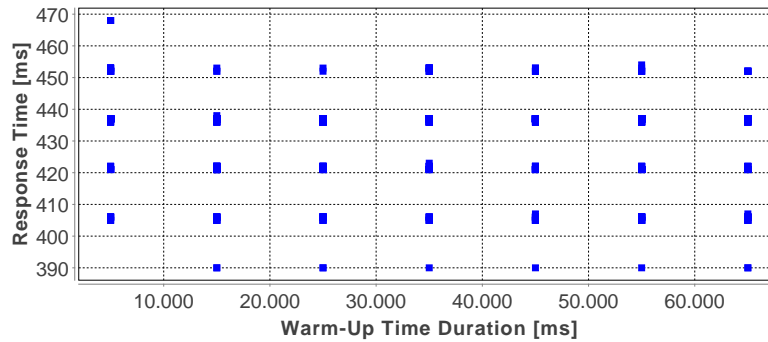


Figure 7.8.: Response times over increasing warm-up time durations

response times vary slightly within a range of 70 ms. However, considered over increasing warm-up time durations the response times are quite constant. Therefore, the Ramp antipattern could not be detected.

7.3.3. The One Lane Bridge

For the detection of the One Lane Bridge (OLB), each service on the application layer has been instrumented automatically in order to retrieve response times and CPU times of each service request. Based on this data the Advanced Approximated Blocking Times Analysis (cf. Chapter 6.5.1.3) has been conducted. In six of the eight provided services an One Lane Bridge has been detected. For the Search Request and the Buy Confirm interactions an OLB could not be found. In Figure 7.9, we exemplarily depicted the approximated blocking times over the concurrency level (Number of Users) for the Buy Request interaction. The blocking times increase monotonously over the concurrency level indicating the presence of an One Lane Bridge antipattern which forces some execution threads to wait. The OLB detection technique discovered an OLB antipattern by applying the Advanced Approximated Blocking Times Analysis (cf. Chapter 6.5.1.2) on this data. Therefore, a search for OLB root causes has been conducted during the investigation of the OLB antipattern.

Firstly, the detection approach searched for synchronized methods in the bookstore implementation. However, no synchronized methods has been found in the implementation of the bookstore. Therefore, the search for a synchronized methods root cause has been aborted. As a second step, a search for database related root causes has been conducted automatically. Through dynamic instrumentation monitoring probes for capturing response times have been injected in all operations accessing the database or the database connection pool. For some of these operations the measured response times are depicted in

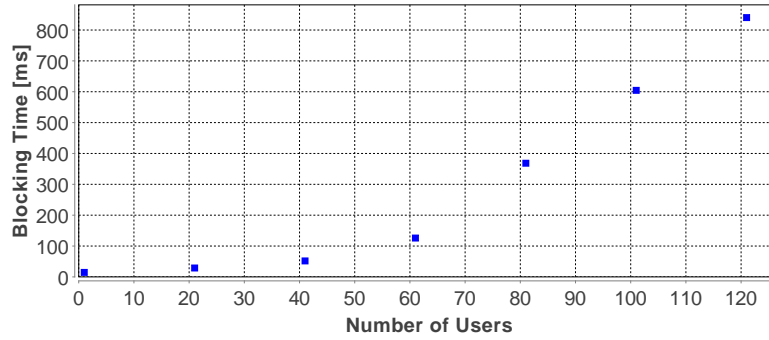


Figure 7.9.: Approximated blocking times for the Buy Request interaction

Figure 7.10. The `get connection` operation is the only one which yields strictly growing response times under increasing workload intensity. Therefore, the OLB root cause analysis technique discovers this operation as the root cause for the observed One Lane Bridge. Now, we know that the used TPC-W solution contains a bottleneck which is caused by the `get connection` operation. Further interpretations of this detection result have to be conducted manually. As these steps require much semantics they are not realized in the scope of the detection approach, yet. Using the detection and measurement results from the detection approach, in the following, we conduct some further steps to find the actual reason for the discovered performance problem.

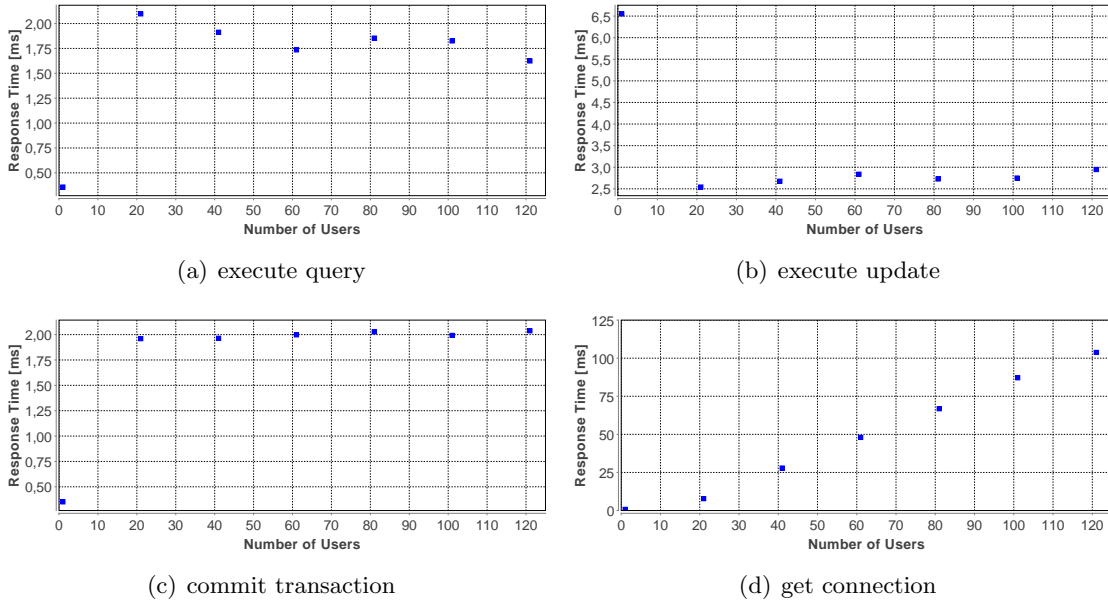


Figure 7.10.: Response times of database related operations

7.3.4. Interpreting the Result

The response times for executing queries and updates or committing transactions do not increase with the concurrency level (cf. Figure 7.10(a), (b), (c)). At the same time, the mean utilization of the CPUs on the database server is very low as can be seen in Figure 7.11(a). Thus, the database itself or the access to the database is not the root cause for the discovered One Lane Bridge. However, according to Figure 7.10(d) the response times for getting a database connection increase with the number of concurrent users which causes a bottleneck. According to the description of the experiment setup (cf. Section

7.2), database connections are retrieved from a database connection pool (DBCP). As the DBCP runs on the application server it is possible that the bottleneck is caused by the machine hosting the application server. However, as depicted in Figure 7.11(b) the mean utilization of the CPUs on the application server machine does not exceed 70%. Moreover, the application server quickly reaches an utilization of about 60% which does not grow anymore with further increase of the concurrency level. Therefore, we have reason to

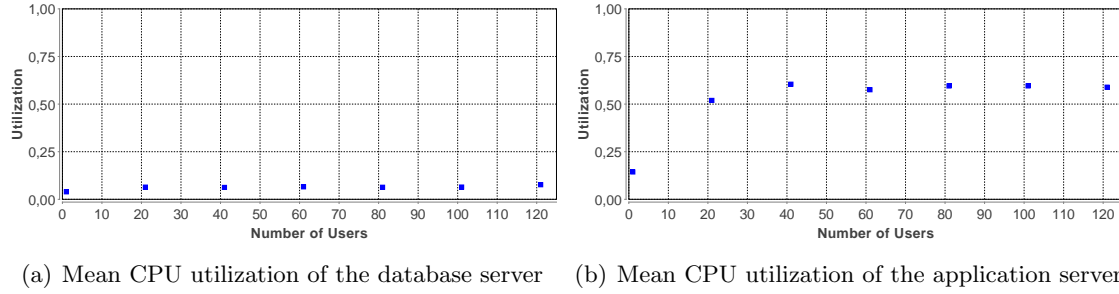


Figure 7.11.: CPU utilizations

presume that the considered bottleneck is caused by the DBCP itself.

Analyzing the configurations for connection pooling, we discovered that the pool size was set to 15 connections. As the system has been examined under concurrency levels which are much higher than 15, it seems that the observed bottleneck is caused by a lack of available database connections. We reran the antipattern detection approach using DBCP sizes of 100 and 500 connections. However, the same antipatterns have been discovered as before. Considering response times from the Varying Response Times measurements we observe a response time behaviour which is even worse when using pool sizes of 100 or 500 (cf. Figure 7.12). Using a pool size of 15 yields a mean response time about 35s for an

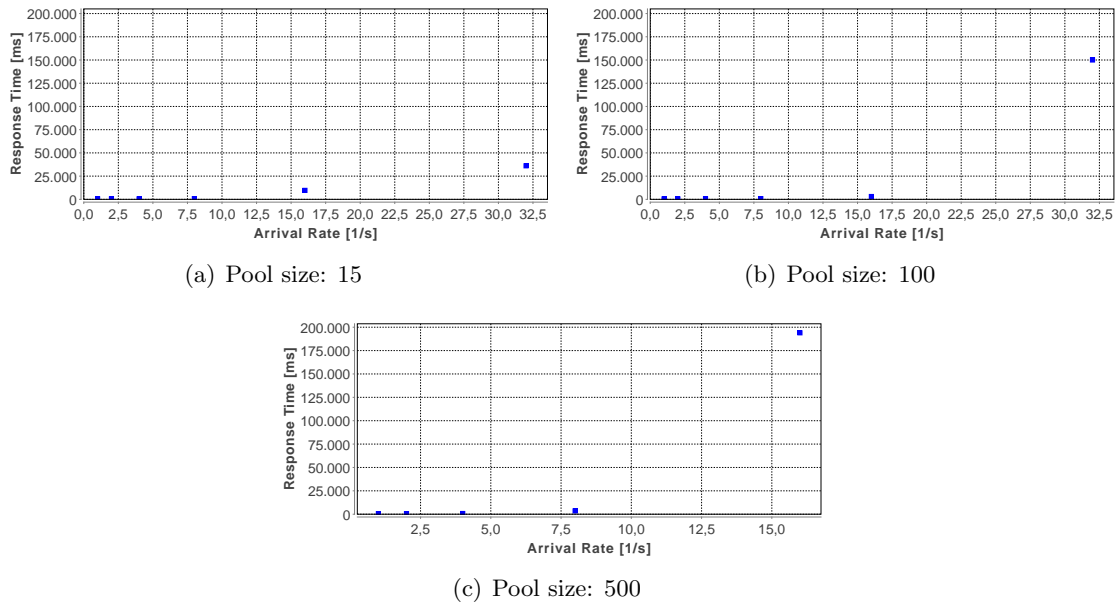


Figure 7.12.: Mean response times measured using different DBCP sizes

arrival rate of $32s^{-1}$. For the same arrival rate and a pool size of 100 connections the mean response time value is 150s. For a pool size of 500 the mean response time is 190s even for an arrival rate of $16s^{-1}$. Thus, increasing the pool size did not solve the bottleneck. Moreover, the bottleneck behaviour has been intensified. Consequently, more connections increase the management overhead of the DBCP. Figure 7.13 confirms this presumption.

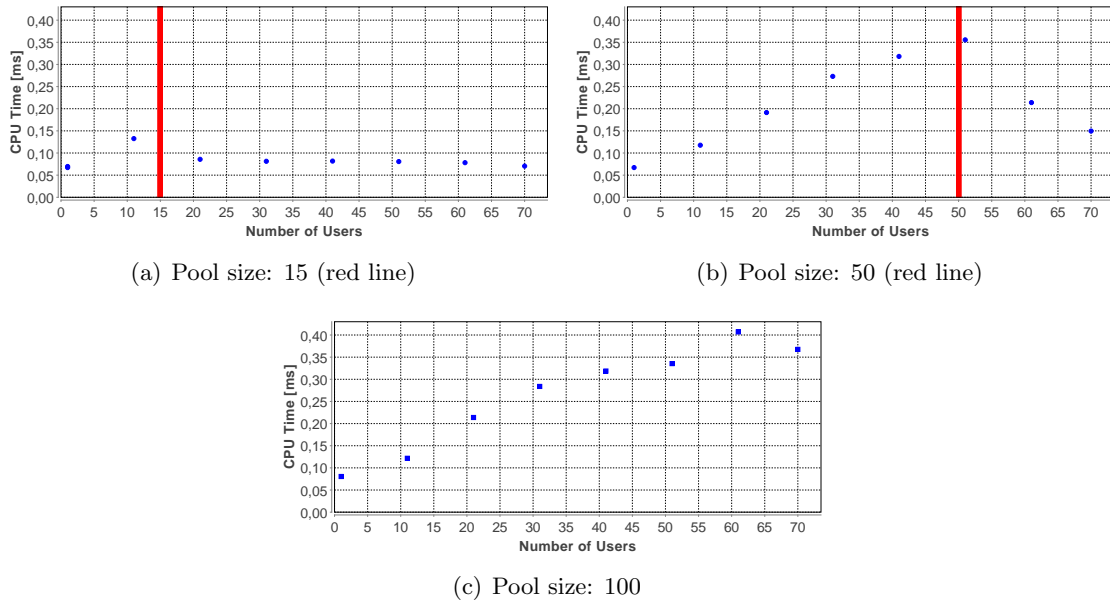


Figure 7.13.: CPU times of the `get connection` operation in dependence on the concurrency level for different DBCP sizes

As long as the concurrency level is smaller than the DBCP size the CPU times for retrieving a database connection increase with the concurrency level. Furthermore, retrieving and releasing connections from the DBCP are synchronized operations. Thus, more connections lead to higher CPU times of the `get connection` operation. Higher CPU times plus synchronization lead to queueing. Finally, progressive queueing leads to a bottleneck. Thus, a big DBCP size leads to a bottleneck. On the other hand, a small DBCP size leads to a bottleneck, too. Although the CPU times do not increase anymore if the concurrency level exceeds the DBCP size (cf. Figure 7.13), a bottleneck is caused by a lack of available database connections. In order to solve this scalability problem, the server configuration of the used TPC-W solution has to be adapted. In particular, more application instances are needed in order to reduce the bottleneck effect of each used DBCP. However, we do not investigate other server configurations as this is beyond the scope of this thesis.

7.4. Summary

In this chapter, we conducted a comprehensive evaluation of the antipattern detection approach introduced in this thesis. For this purpose, we set up a solution of the TPC-W Benchmark using a Java implementation of the bookstore, one application server machine hosting Apache Tomcat and one database server machine running MySQL. Instead of using the workload generation as specified by TPC-W we defined an own interaction sequence. Using this sequence, different workload types and workload intensities have been applied to the system under test according to the antipattern detection approach described in this thesis. The detection approach automatically discovered a Varying Response Times problem caused by an One Lane Bridge. Furthermore, the `get connection` operation of the database connection pool has been automatically identified as root cause for the observed One Lane Bridge. Using this information and measurement data from the detection approach we easily found the actual reason for the discovered bottleneck by performing some manual steps. The evaluation in this chapter demonstrates the applicability of the described detection approach.

8. Conclusion

In this thesis, we introduced a measurement-based approach for automatically detecting performance problems by executing systematic experiments. In this Chapter, we conclude our work by summarizing the results and defining issues for future work.

8.1. Summary

In order to overcome the problems of existing monitoring-based detection approaches we introduced and combined some concepts. We introduced an adaptive measurement approach which is based on systematic measurement experiments. Through systematic measurement experiments we are able to conduct target-oriented search for performance problems. In particular, the detection results do not depend on a random workload but are yielded from controlled experiments. For the realization of the adaptive measurement approach we developed an antipattern detection process and introduced an architecture which is based on the Software Performance Cockpit.

As our approach is based on analyses of measurement data, we investigated different alternatives for capturing this data. In particular, we have shown that the monitoring overhead of dynamic instrumentation is negligible in contrast to full instrumentation.

For a set of antipatterns to be detected we investigated different detection techniques. To compare these alternatives we introduced and implemented a validation software system used for testing each detection technique. Based on the validation system, we defined some scenarios by injecting specific performance antipatterns into the application. For each investigated antipattern and each scenario we defined our expectations for the detection result. These expectations and the compliance degree of the actual detection results form a quality metric which has been used for comparing different detection techniques.

We examined different alternatives for investigating the variety of response times. As a common metric we used the coefficient of variance (COV) as an indicator for the Varying Response Times problem. However, using a fixed threshold for the COV yielded wrong decisions for some scenarios. Adapting the threshold to the median of measured response times solved this problem yielding correct decision results for all scenarios.

The Ramp was the second examined antipattern. Here, we were interested in the response time progression over operation time. As concurrency effects impair the typical behaviour of a Ramp antipattern we developed an experiment configuration which allows us to exclude concurrency effects when observing the Ramp antipattern. For root cause analysis of the Ramp antipattern we analyzed the call tree of invoked service requests. Using a

breadth-first-search on the call tree we provided two algorithms allowing to identify operations which cause an observed Ramp antipattern.

The detection of the Dormant References antipattern is quite similar to the detection technique of the Ramp antipattern. However, we experienced that accurately measuring memory consumption is a difficult task when using programming languages with a managed heap (like Java or C#). As garbage collection is a non-deterministic task differences in memory consumption could be captured only when big sized objects have been used causing large memory consumption differences.

The One Lane Bridge was the last examined antipattern, whose detection is based on the analysis of blocking times. We investigated different alternatives for determining and analyzing blocking times. Retrieving blocking times directly from the virtual machine did not capture external blocking times. Therefore, wrong detection decisions for scenarios containing a database related One Lane Bridge have been made. We applied Little's Law to approximate the blocking times from response times and CPU times of the corresponding request. Analyzing the approximated blocking times yielded correct decisions for all scenarios. Finally, we examined root causes for the One Lane Bridge. For the synchronized methods root cause we analyzed the blocking times at each synchronized method. In order to discover database related root causes we captured and analyzed response times of all operations accessing the database.

Having developed proper detection techniques for the considered antipatterns we evaluated the overall detection approach. Therefore, we applied the detection approach to a Java Servlets solution of the TPC-W Benchmark which is a quasi-realistic software system for scalability testing. In the used TPC-W solution two antipatterns have been detected automatically: the Varying Response Time problem and the One Lane Bridge. Moreover, the detection approach discovered an operation causing the observed problems. With little effort, we were able to determine the actual problem root cause by performing some manual considerations. Finding the problem and its root cause without the introduced detection approach would be a much more difficult and time-consuming task. The evaluation shows the applicability and the benefits of the introduced detection approach.

8.2. Future Work

In this section, we discuss some open problems to be investigated in future work.

Although we developed a flexible and extendable architecture for detecting performance antipatterns, we provided detection techniques only for some specific antipatterns. One goal for future work is to investigate further antipatterns and provide detection techniques for these antipatterns, as well.

When examining the Dormant References antipattern we experienced the difficulty to accurately conduct memory measurements. Thus, for future work it is important to find approaches which allow to overcome this problem in order to detect memory related performance antipatterns.

One of the biggest disadvantages of measurement-based performance evaluation approaches is the time required to conduct measurements. In this thesis, we increased the efficiency by introducing the adaptive measurement approach. However, there is further potential for optimizing the efficiency of experiment execution. For instance, analysis could be executed in parallel to experiment execution aborting the measurements when no further measurement data is required. Finding such optimization options is another task for future work. The evaluation with the TPC-W Benchmark has shown that the described antipattern detection approach is very useful when searching for performance problems. However at the end, we had to perform some consideration steps manually in order to find the actual root cause. Some steps require much semantics and knowledge about the specific system

under test to make decisions about possible root causes. Developing concepts which allow to automate these steps is another interesting point for future investigations.

Finally, we mentioned that the described detection approach is intended to be used by developers during the development phase of a software system. However, right now the approach comprises some steps which entail additional effort for the developer. In particular, the developer has to provide a measurement specification and a usage adapter. Furthermore, as measurement execution takes long (hours), many developers will not be willing to execute these measurements. Therefore, for future work we plan to apply the described detection approach on automatically extracted system models by executing simulations rather than real measurements. This idea has the potential to reduce the time and effort required to detect antipatterns.

Bibliography

- [AFC98] G. Antoniol, R. Fiutem, and L. Cristoforetti, “Design pattern recovery in object-oriented software,” in *Program Comprehension, 1998. IWPC’98. Proceedings., 6th International Workshop on.* IEEE, 1998, pp. 153–160.
- [AIS⁺77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A pattern language.* Oxford Univ. Pr., 1977.
- [AZ05] P. Avgeriou and U. Zdun, “Architectural patterns revisited—a pattern,” 2005.
- [BHS07] F. Buschmann, K. Henney, and D. Schmidt, *Pattern-oriented software architecture: On patterns and pattern languages.* John Wiley & Sons Inc, 2007, vol. 5.
- [BPSH05] S. Boroday, A. Petrenko, J. Singh, and H. Hallal, “Dynamic analysis of java applications for multithreaded antipatterns,” in *Proceedings of the third international workshop on Dynamic analysis*, no. May. ACM, 2005, pp. 1–7. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1083247>
- [Bro98] W. Brown, *AntiPatterns: refactoring software, architectures, and projects in crisis*, ser. ITPro collection. Wiley, 1998. [Online]. Available: <http://books.google.de/books?id=Dp9QAAAAMAAJ>
- [Chi00] S. Chiba, “Load-time structural reflection in java,” *ECOOP 2000 Object-Oriented Programming*, pp. 313–336, 2000.
- [CME10] V. Cortellessa, A. D. Marco, and R. Eramo, “Digging into UML models to remove performance antipatterns,” *Stochastic Models in*, p. 9, 2010. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1808877.1808880><http://portal.acm.org/citation.cfm?id=1808880>
- [CMR10] V. Cortellessa, A. Martens, and R. Reussner, “A Process to Effectively Identify ‘Guilty’ Performance Antipatterns,” *Approaches to Software*, pp. 368–382, 2010. [Online]. Available: <http://www.springerlink.com/index/WL11718486334174.pdf>
- [CST03] S. Chiba, Y. Sato, and M. Tatsubori, “Using hotswap for implementing dynamic aop systems,” in *1st Workshop on Advancing the State-of-the-Art in Run-time Inspection, july*, 2003.
- [CW00] M. Courtois and M. Woodside, “Using regression splines for software performance analysis,” in *WOSP ’00: Proceedings of the 2nd international workshop on Software and performance.* New York, NY, USA: ACM, 2000, pp. 105–114.
- [DAKW03] B. Dudley, S. Asbury, J. Krozak, and K. Wittkopf, *J2EE antipatterns.* Wiley, 2003. [Online]. Available: http://books.google.com/books?hl=en&lr=&id=8cV8awbqyi0C&oi=fnd&pg=PR5&dq=J2EE+Antipatterns&ots=HGJK2ghMI8&sig=tDmO_TMPzTC_74ToQnV4g8NHKOk

- [DGS02] R. F. Dugan, E. P. Glinert, and A. Shokoufandeh, “The Sisyphus database retrieval software performance antipattern,” in *Proceedings of the third international workshop on Software and performance - WOSP '02*. New York, New York, USA: ACM Press, 2002, p. 10. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=584369.584372>
- [EHJ11] N. Ehmke, A. Hoorn, and R. Jung, “Kieker 1 . 3 User Guide,” *Computer*, pp. 1–83, 2011.
- [FH12] M. Faber and J. Happe, “Systematic adoption of genetic programming for deriving software performance curves,” April 2012, to be published.
- [Gam95] E. Gamma, *Design patterns: elements of reusable object-oriented software*, ser. Addison-Wesley professional computing series. Addison-Wesley, 1995. [Online]. Available: <http://books.google.de/books?id=6oHuKQe3TjQC>
- [GHJ⁺02] E. Gamma, R. Helm, R. Johnson, J. Vlissides *et al.*, *Design patterns*. Addison-Wesley Reading, MA, 2002, vol. 1.
- [GHJV93] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design patterns: Abstraction and reuse of object-oriented design,” *ECOOP'93 Object-Oriented Programming*, pp. 406–431, 1993.
- [HAT⁺04] H. Hallal, E. Alikacem, W. Tunney, S. Boroday, and A. Petrenko, “Antipattern-based detection of deficiencies in java multithreaded software,” *Fourth International Conference on Quality Software, 2004. QSIC 2004. Proceedings.*, pp. 258–267, 2004. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1357968>
- [Heg12] C. Heger, “Automatische problemdiagnose in performance-unittests,” February 2012, diploma Thesis.
- [HEK05] J. Hartung, B. Elpelt, and K. Klösener, *Statistik*. Oldenbourg Wissenschaftsverlag, 2005.
- [HH04] E. Hilsdale and J. Hugunin, “Advice weaving in AspectJ,” in *Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, New York, USA: ACM, 2004, pp. 26–35. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=976270.976276>
<http://portal.acm.org/citation.cfm?id=976270.976276>
- [HHHL03] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, “Automatic design pattern detection,” in *Program Comprehension, 2003. 11th IEEE International Workshop on.* IEEE, 2003, pp. 94–103. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1199193
- [HWSK10] J. Happe, D. Westermann, K. Sachs, and L. Kapová, “Statistical inference of software performance models for parametric performance completions,” *Research into Practice-Reality and Gaps*, no. 216556, pp. 20–35, 2010.
- [hyp] “Hyperic homepage,” <http://www.hyperic.com/> - last visited: 15.11.2011.
- [jav] “Object web homepage: Java implementation of the tpc-w benchmark,” <http://jmob.ow2.org/tpcw.html> - last visited: 10.04.2012.
- [jpd] “Jpda homepage,” <http://docs.oracle.com/javase/1.4.2/docs/guide/jpda/enhancements.html> - last visited: 15.12.2011.

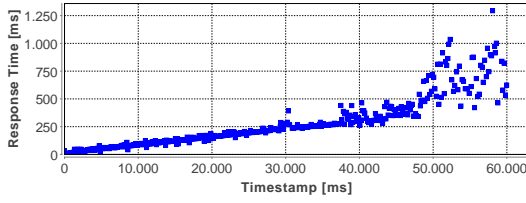
- [kie] “Kieker homepage,”
<https://se.informatik.uni-kiel.de/kieker/features/> - last visited: 15.11.2011.
- [Kis02] M. Kis, “Information security antipatterns in software requirements engineering,” in *9th Conference of Pattern Languages of Programs (PloP)*, vol. 11. Citeseer, 2002.
- [KLM⁺97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-oriented programming,” *ECOOP’97-Object-Oriented Programming*, pp. 220–242, 1997.
- [Koz10] H. Koziol, “Performance evaluation of component-based software systems: A survey,” *Performance Evaluation*, vol. 67, no. 8, pp. 634–658, 2010. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S016653160900100X>
- [Lil00] D. Lilja, *Measuring computer performance: a practitioner’s guide*. Cambridge Univ Pr, 2000.
- [MCC⁺95] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, “The paradyn parallel performance measurement tool,” *Computer*, vol. 28, no. 11, pp. 37–46, 1995.
- [Men02] D. Menascé, “Tpc-w: A benchmark for e-commerce,” *Internet Computing, IEEE*, vol. 6, no. 3, pp. 83–87, 2002.
- [Mes96] G. Meszaros, “A pattern language for improving the capacity of reactive systems,” in *Pattern languages of program design 2*. Addison-Wesley Longman Publishing Co., Inc., 1996, pp. 575–591.
- [MG11] P. Mell and T. Grance, “The nist definition of cloud computing (draft),” *NIST special publication*, vol. 800, p. 145, 2011.
- [mys] “Mysql homepage,”
<http://www.mysql.de> - last visited: 10.04.2012.
- [NR69] P. Naur and B. Randell, “Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7th to 11th october, 1968.” Nato, 1969.
- [OW90] M. Oliver and R. Webster, “Kriging: a method of interpolation for geographical information systems,” *International Journal of Geographical Information System*, vol. 4, no. 3, pp. 313–332, 1990.
- [PM08] T. Parsons and J. Murphy, “Detecting Performance Antipatterns in Component Based Enterprise Systems,” *Journal of Object Technology*, vol. 7, no. 3, pp. 55–90, 2008.
- [PS97] D. Petriu and G. Somadder, “A pattern language for improving the capacity of layered client/server systems with multi-threaded servers,” *Proceedings of EuroPLoP’97*, 1997.
- [PVR95] E. Pozzetti, V. Vetland, J. Rolia, and G. Serazzi, “Characterizing the resource demands of tcp/ip,” in *High-Performance Computing and Networking*. Springer, 1995, pp. 79–85.
- [Ray07] D. Rayside, “Object ownership profiling: a technique for finding and fixing memory leaks,” *Proceedings of the twenty-second IEEE/ACM*, 2007. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1321661>

- [RCS08] R. Rouvoy, D. Conan, and L. Seinturier, "Software architecture patterns for a context-processing middleware framework," *Distributed Systems Online, IEEE*, vol. 9, no. 6, pp. 1–1, 2008.
- [RO00] G. Reese and A. Oram, *Database Programming with JDBC and JAVA*. O'Reilly & Associates, Inc., 2000.
- [sap] "Sap internal document."
- [SKK⁺01] M. Sitaraman, G. Kulczycki, J. Krone, W. Ogden, and A. Reddy, "Performance specification of software components," in *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*. ACM, 2001, pp. 3–10.
- [Sma06] B. Smaalders, "Performance anti-patterns," *Queue*, vol. 4, no. 1, pp. 44–50, Jan. 2006. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/21800422http://portal.acm.org/citation.cfm?id=1117389.1117403>
- [Smi00] W. Smith, "TPC-W: Benchmarking an ecommerce solution," Intel Corporation, Tech. Rep., Feb 2000.
- [SW00] C. Smith and L. Williams, "Software performance antipatterns," in *Proceedings of the 2nd international workshop on Software and performance*. New York, New York, USA: Citeseer, 2000, pp. 127–136. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=350391.350420http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.31.6449&rep=rep1&type=pdf>
- [SW02a] —, *Performance Solutions: a practical guide to creating responsive, scalable software*. Addison-Wesley Boston, MA;, 2002, vol. 34.
- [SW02b] —, "Software Performance AntiPatterns; Common Performance Problems and their Solutions," *CMG-CONFERENCE-*, 2002. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.6968&rep=rep1&type=pdf>
- [SW03a] —, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in *Computer Measurement Group Conference*. Citeseer, 2003, pp. 717–725. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.4517&rep=rep1&type=pdf>
- [SW03b] —, "New software performance antipatterns: More ways to shoot yourself in the foot," *CMG-CONFERENCE-*, 2003. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.9256&rep=rep1&type=pdf>
- [TK11] C. Trubiani and A. Koziol, "Detection and solution of software performance antipatterns in palladio architectural models," in *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering*. ACM, 2011, pp. 19–30.
- [tom] "Apache tomcat homepage," <http://tomcat.apache.org/> - last visited: 10.04.2012.
- [tpc] "Tpc homepage," <http://www.tpc.org> - last visited: 06.04.2012.
- [Tra02] Transaction Processing Performance Council, "TPC Benchmark W (Web Commerce) Specification v.1.8," Feb 2002. [Online]. Available: <http://www.tpc.org>

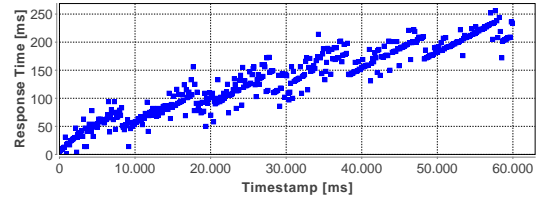
- [vmw] “vmware homepage,”
<http://www.vmware.com/> - last visited: 15.11.2011.
- [WHHH10] D. Westermann, J. Happe, M. Hauck, and C. Heupel, “The performance cockpit approach: A framework for systematic performance evaluations,” in *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2010, pp. 31–38.
- [WHW12] A. Wert, J. Happe, and D. Westermann, “Integrating software performance curves with the palladio component model,” April 2012, to be published.
- [WVCB01] M. Woodside, V. Vetland, M. Courtois, and S. Bayarov, “Resource function capture for performance aspects of software components and sub-systems,” *Performance Engineering*, no. 1, pp. 239–256, 2001. [Online]. Available: <http://www.springerlink.com/index/71k0ka2c1l7fn99q.pdf>
- [XH96] Z. Xu and K. Hwang, “Modeling communication overhead: Mpi and mpl performance on the ibm sp2,” *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 4, no. 1, pp. 9–24, 1996.
- [Xu09] J. Xu, “Rule-based automatic software performance diagnosis and improvement,” *Performance Evaluation*, pp. 1–12, 2009. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0166531609001631>
- [YB98] J. Yoder and J. Barcalow, “Architectural patterns for enabling application security,” *Urbana*, vol. 51, p. 61801, 1998.

Appendix

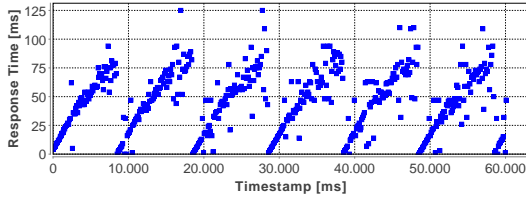
A. Validation Scenarios



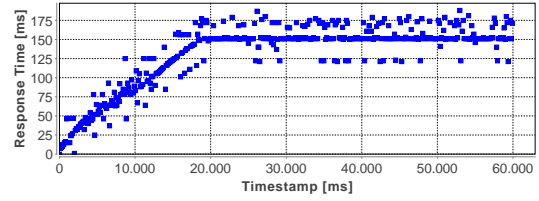
(a) Scenario 1 - classic ramp behaviour:



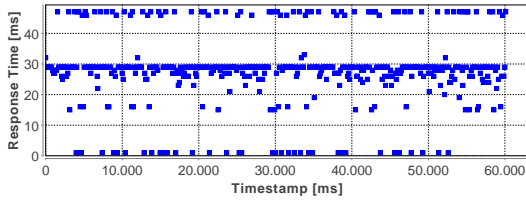
(b) Scenario 2 - growing list



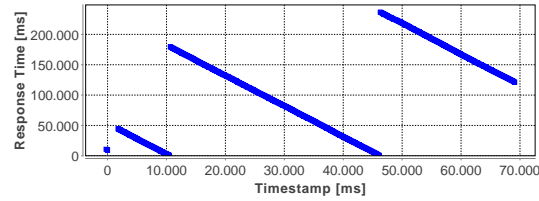
(c) Scenario 3 - periodical clean-up



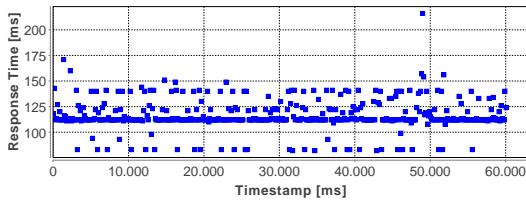
(d) Scenario 4 - fixed-sized queue



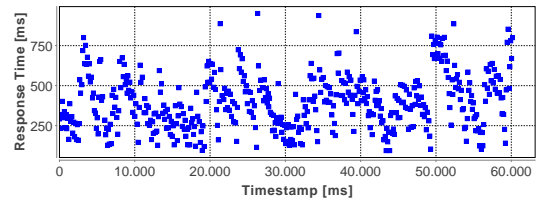
(e) Scenario 5 - hashing transactions



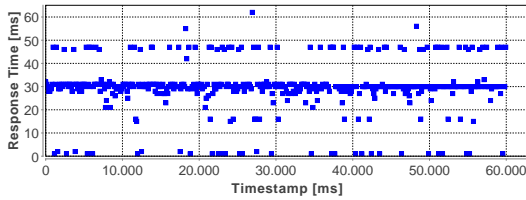
(f) Scenario 6 - synchronized method:



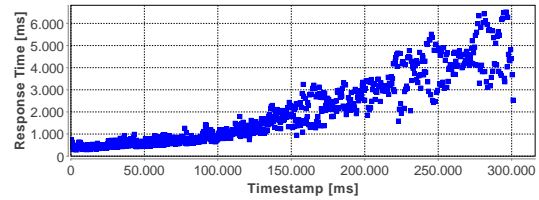
(g) Scenario 7 - resolved synchronization



(h) Scenario 8 - storing data as big byte arrays



(i) Scenario 9 - using proper database structures



(j) Scenario 10 - SPA interaction

Figure A.1.: Validation scenarios: response times for arrival rate of 8 s^{-1}

B. TPC-W Measurements

B.1. Varying Response Times

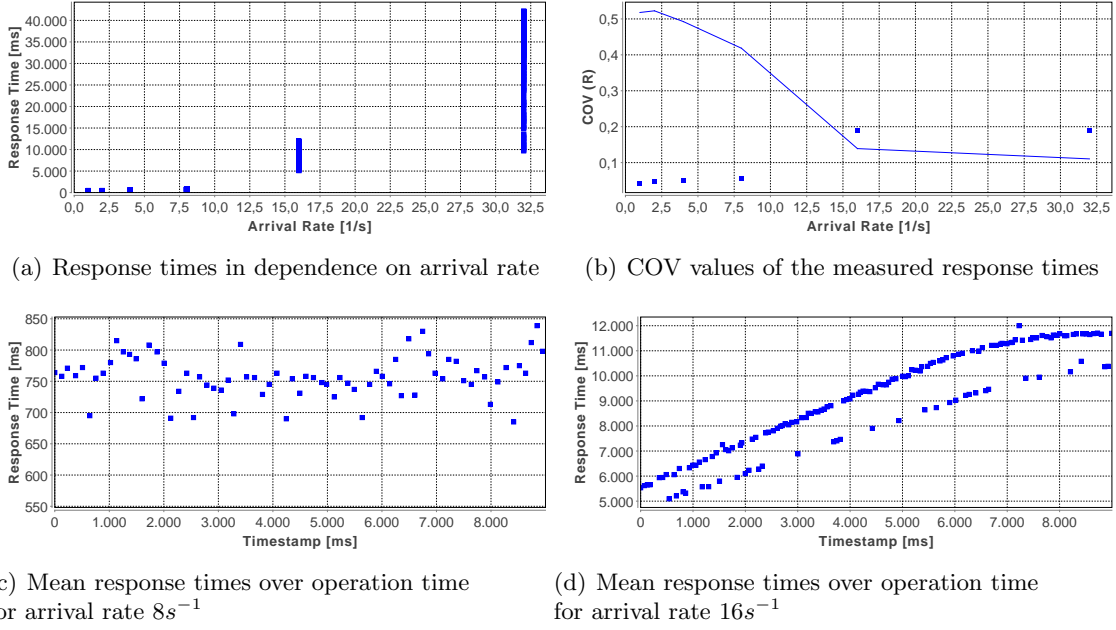


Figure B.2.: Response time measurements for the Varying Response Times problem

B.2. The Ramp

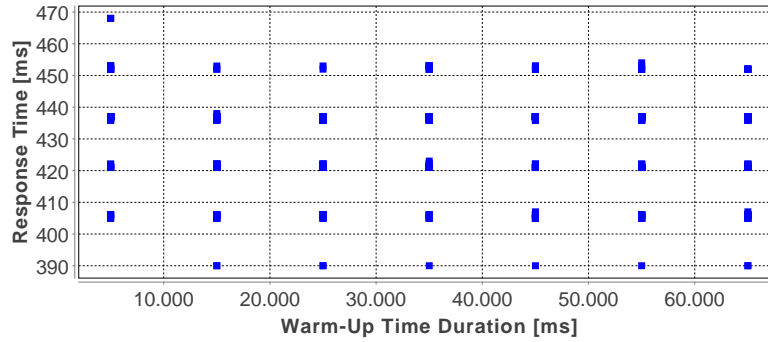


Figure B.3.: Response times over increasing warm-up time durations

B.3. One Lane Bridge Detection

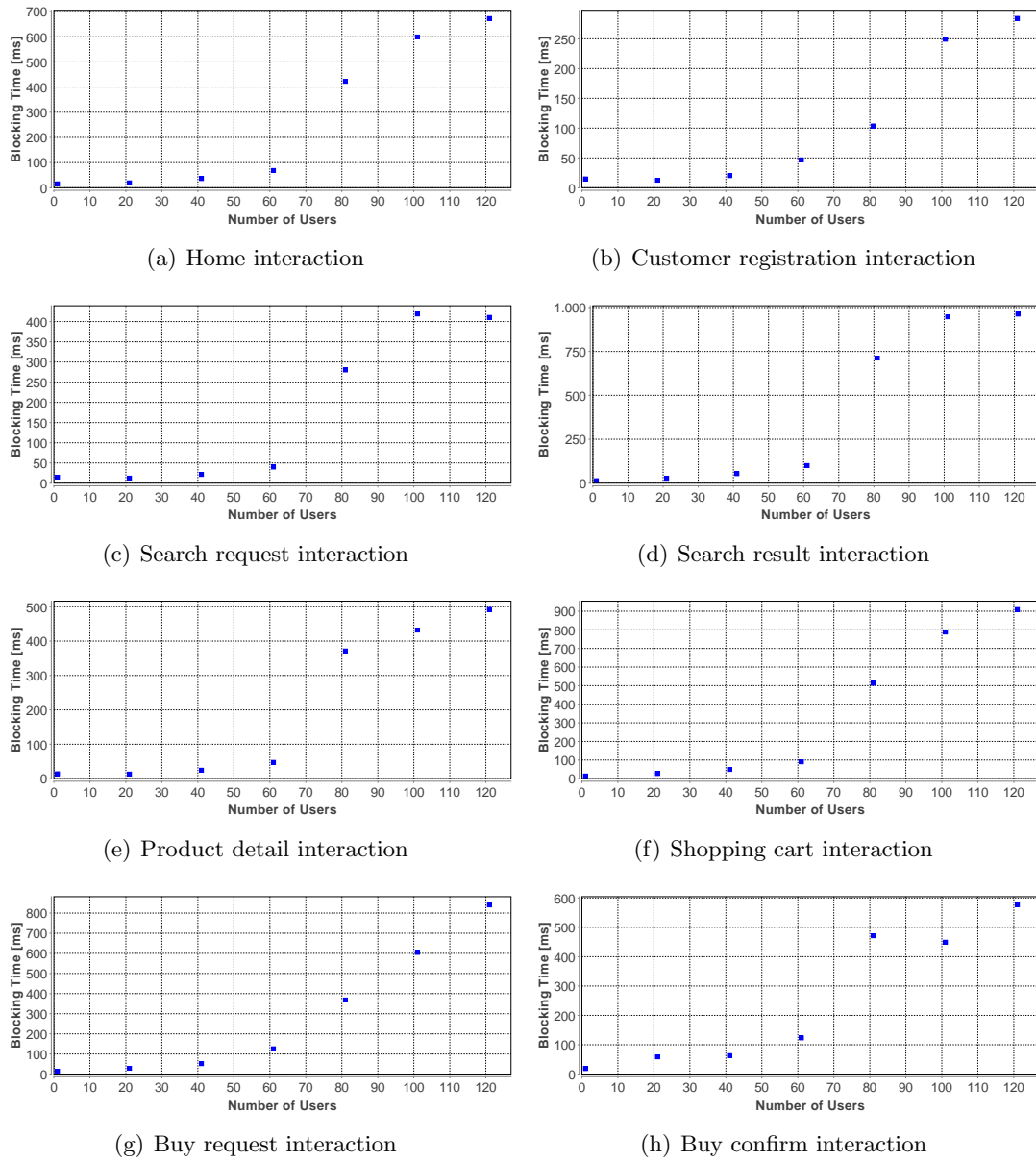


Figure B.4.: Mean approximated blocking times depicted over concurrency level for each TPC-W service

B.4. One Lane Bridge Root Cause Analysis

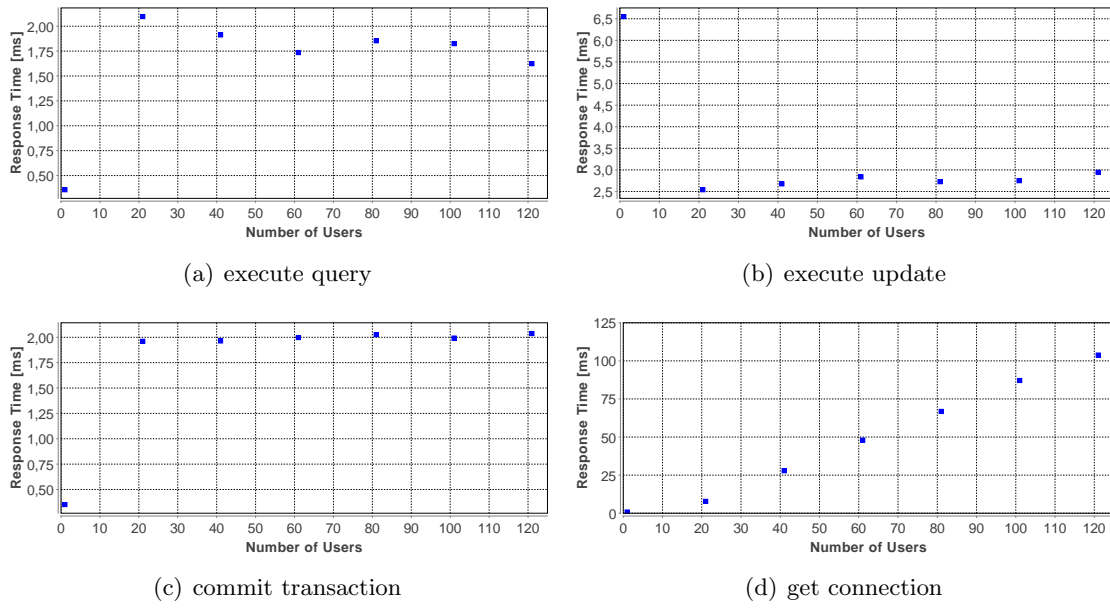


Figure B.5.: Response times of database related operations

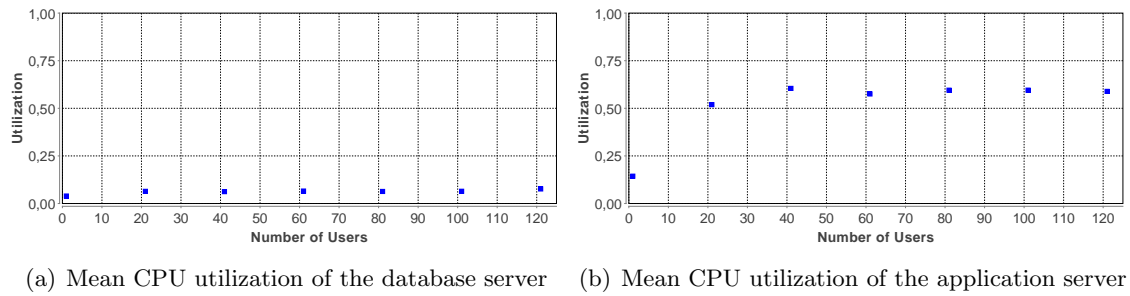


Figure B.6.: CPU utilizations

List of Figures

2.1. Decorator Pattern, [GHJ ⁺ 02]	6
2.2. Antipattern hierarchy (following and extending [PM08])	6
2.3. Empty Semi Trucks antipattern: example of an improper interface	10
2.4. Illustration of the Ramp antipattern, following [SW03b]	12
2.5. More is Less antipattern: example throughput behaviour	14
2.6. Overview on the categorization	19
2.7. Performance Problem Hierarchy	20
2.8. Kieker architecture, following [kie]	24
2.9. Example for a queueing network	29
2.10. Architecture of the Software Performance Cockpit	31
4.1. Antipattern detection integrated with development	38
4.2. Overall process applying general monitoring approach	39
4.3. Example for a decision tree	40
4.4. Overall process applying adaptive measurement approach	41
4.5. General antipattern detection architecture	42
4.6. Antipattern Detection Process	43
4.7. Experiment Execution Sub-Process	44
5.1. Example: comparing partly instrumentation with full instrumentation	48
5.2. Full instrumentation compared to top-level instrumentation	48
5.3. Dynamic AOP instrumentation	50
5.4. Dynamic instrumentation with Javassist, HotSwap & Kieker	51
6.1. Part of the performance problem hierarchy	53
6.2. Online Banking System	55
6.3. Detection result expectations for each considered scenario	59
6.4. Adapted COV	62
6.5. COV for Scenario 8 and Scenario 9	63
6.6. Measurement data for the Separated Time Windows Analysis technique	67
6.7. Memory consumption measurements for Scenario 2	73
6.8. Memory consumptions for Scenario 2 and 3 using big transaction objects	73
6.9. Mean response time and blocking time behaviour for Scenario 6 and 7	75
6.10. Mean response time and blocking time behaviour of Scenario 8	76
6.11. Object locked by one method.	78
6.12. Object locked by multiple methods.	79
6.13. Mean blocking times for locked objects of Scenario 6 and 10	80
6.14. Mean response times and database CPU utilization for DB-Lock Scenarios	82
6.15. Mean response times and database CPU utilization for Scenario 7*	82
7.1. Customer Behaviour Model Graph (following [Men02])	85
7.2. Typical environment configuration for TPC-W (following [Smi00])	86

7.3. Experiment setup for evaluation of the antipattern detection approach . . .	87
7.4. Workload for evaluation	88
7.5. Response times in dependence on the arrival rate	89
7.6. Response times depicted over execution time for arrival rates $8s^{-1}$ and $16s^{-1}$	89
7.7. Evaluation result: COV for Varying Response Times	89
7.8. Response times over increasing warm-up time durations	90
7.9. Approximated blocking times for the Buy Request interaction	91
7.10. Response times of database related operations	91
7.11. CPU utilizations	92
7.12. Mean response times measured using different DBCP sizes	92
7.13. CPU times of the <code>get connection</code> operation	93
A.1. Validation scenarios	105
B.2. VRT detection measurements	106
B.3. Response times over increasing warm-up time durations	106
B.4. OLB detection measurements	107
B.5. Response times of database related operations	108
B.6. CPU utilizations	108

List of Tables

2.1. Overview on software performance antipatterns	8
6.1. Detection experiments with different values for the threshold T_{COV}	61
6.2. Response time gradients	65
6.3. T-test analyses with different confidence level values, second half analysis . .	66
6.4. Memory consumption analysis for different magnitudes of memory sizes . .	72
6.5. Comparison of detection techniques for the OLB antipattern	76

Nomenclature

AOP	Aspect Oriented Programming
APD	Antipattern Detection
API	Application Programming Interface
CPU	Central Processing Unit
DB	database
DBCP	Database Connection Pool
DBRCA	Database Lock Root Cause Analysis
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JDBC	Java Database Connectivity
JDK	Java Development Kit
JMS	Java Messaging Service
JPDA	Java Platform Debugger Architecture
JVM	Java Virtual Machine
OLB	One Lane Bridge performance antipattern
OS	Operating System
PCM	Palladio Component Model
RMI	Remote Method Invocation
SE	Software Engineering
SIGAR	System Information Gatherer
SMRCA	Synchronized Methods Root Cause Analysis
SoPeCo	Software Performance Cockpit
SPA	Software Performance Antipattern
SPE	Software Performance Engineering
UML	Unified Modeling Language
VRT	Varying Response Times performance problem