Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 343

Detecting Performance Anti-Patterns in Enterprise Applications by Analyzing Execution Traces

Alper Hidiroglu

Course of Study: Softwaretechnik

Examiner: Dr.-Ing. André van Hoorn (Prof.-Vertr.)

Supervisor: Dr.-Ing. André van Hoorn,
Dr. Dušan Okanović

Commenced: June 8, 2016

Completed: December 8, 2016

CR-Classification: B.8.2, D.4.8

Kurzfassung

Application Performance Management (APM) beschäftigt sich mit der Performanz von Anwendungen. Beim Application Performance Monitoring wird das Verhalten der Software mithilfe eines Programms, im folgenden Agent genannt, überwacht. Der Agent sammelt bei der Ausführung der Anwendungssoftware im Hinblick auf Performanz relevante Daten, die ausgewertet werden können. Zum Beispiel werden Methoden instrumentiert, sodass bei der Ausführung der Methode Daten gesammelt werden können. Typische Beispiele für solche Daten sind die Antwortzeit oder die Prozessorzeit einer Methode. Es existieren viele APM-Werkzeuge, die einen solchen Agenten zur Verfügung stellen. Moderne Tools bieten Monitoring an und können Warnungen aussprechen, jedoch bieten sie keine Ursachenanalyse an. Die NovaTec Consulting GmbH [NTC16b] und die Universität Stuttgart (Abteilung Reliable Software Systems) entwickeln aus diesem Grund das diagnoseIT [dIT15], um Probleme bezüglich Performanz in Anwendungssoftware systematisch zu erkennen und zu analysieren. Die Ergebnisse von diagnoseIT werden dem Benutzer in Form eines Berichts präsentiert. Um dem Benutzer konkrete Lösungsvorschläge für Probleme im Bericht bereitzustellen, befasst sich ein Teil der Analyse von diagnoseIT mit der Erkennung von Software Performance Anti-Patterns [ap00]. Ein Performance Anti-Pattern beschreibt einen häufig getätigten Fehler bei der Implementierung einer Software, welcher eben zu Problemen in der Performanz führt. Da die Fehler häufig auftreten und somit bekannt sind, können hierfür konkrete Lösungsvorschläge bereitgestellt werden. Um die Anti-Patterns automatisch in gesammelten Daten zu erkennen, benutzt das diagnoseIT eine Rule Engine. Eine Regel kann dabei ein Anti-Pattern repräsentieren, indem sie die gesammelten Daten mit den Merkmalen des Anti-Patterns vergleicht. Entdeckt eine Regel ein Anti-Pattern, so kann die Analyse automatisch entsprechende Lösungsvorschläge bereitstellen [HHO+15].
Da diagnoseIT bisher nur wenige Regeln besitzt, um Anti-Patterns zu erkennen, fügen wir diagnoseIT in dieser Bachelorarbeit weitere Regeln hinzu, um entsprechend Anti-Patterns zu erkennen. Bisher war es diagnoseIT zudem nicht möglich Anti-Patterns zu erkennen, die mit der Zeit sichtbar werden, weil diagnoseIT keine Möglichkeit hatte, Daten zu speichern. Wir erweitern diagnoseIT in dieser Bachelorarbeit darum um eine spezielle Datenbank, sodass auch Anti-Patterns erkannt werden können, die sich eventuell erst in einem längeren Beobachtungszeitraum von Performanz-Daten herauskristallisieren. Die abschließende Evaluation dieser Arbeit zeigt, dass die entwickelten Regeln Anti-Patterns richtig erkennen. Jedoch zeigt die Evaluierung auch Grenzen der Regeln auf, die in manchen Fällen zu falschen Resultaten führen.

Abstract

Application performance management (APM) deals with performance issues within applications. With application performance monitoring the software's behaviour is investigated with the help of a program, in the following called agent. When the applications software runs, the agent collects performance-relevant data, which can be evaluated. For example, methods get instrumented, so that data can be collected, when the method is executed. Typical examples for such data are the response time or the CPU time of a method. There are many APM tools, that provide such an agent to collect performance data. Modern tools provide monitoring and alerting, but not root-cause analysis. Therefore, the NovaTec Consulting GmbH [NTC16b] and the University of Stuttgart (Reliable Software Systems group) are developing the diagnoseIT [dIT15] to systematically detect and analyze performance problems in applications software. The results of the diagnoseIT analysis are summarized in a report, which is then presented to the user. To provide concrete approaches for a solution concerning a performance problem within the monitored software, one part of the diagnoseIT analysis deals with detection of software performance anti-patterns [ap00]. A performance anti-pattern describes a frequently made mistake, when implementing a software and which results in performance problems. Because the mistake is made often and well known, concrete approaches for a solution can be proposed. To detect anti-patterns in collected data, diagnoseIT uses a rule engine. Thereby, a rule can represent an anti-pattern by comparing the collected data with the characteristics of the anti-pattern. When a rule detects an anti-pattern, the diagnoseIT analysis automatically can provide concrete solutions for a performance problem [HHO+15].
In its current state, diagnoseIT has only a few rules to detect anti-patterns. So in this bachelor thesis, we add more rules to the diagnoseIT to detect other performance anti-patterns. Also, diagnoseIT has no possibility to detect anti-patterns, that get visible over time, because it has no possibility to save data. Therefore, in this bachelor thesis, we extend the diagnoseIT analysis with a special database, so that performance anti-patterns can be detected, which become visible in a longer observation period of performance data. The concluding evaluation of this work shows, that the implemented rules correctly detect anti-patterns. However, the evaluation also identifies limitations of the rules, that lead to wrong results in some cases.

Contents

1. Introduction 1
   1.1. Motivation 1
   1.2. Goals 3
   1.3. Overview of the Approach 4
   1.4. Document Organization 4

2. Foundations and State of the Art 7
   2.1. Terminology 7
   2.2. Software Performance Anti-Patterns 8
   2.3. Related Work 9
   2.4. diagnoseIT 11
   2.5. OPEN.xtrace 12
   2.6. Time Series Database 13

3. Rule-based Approach of Anti-Pattern Detection 15
   3.1. Overview of detectable Anti-Patterns 15
   3.2. Not detectable Anti-Patterns 19
   3.3. diagnoseIT Rule Engine 21

4. Detection based on Trace Data 27
   4.1. Approach 28
   4.2. Narrowing down a Trace 28
   4.3. Considered Anti-Patterns 31

5. Detection based on Time Series 41
   5.1. Approach 41
   5.2. Considered Anti-Patterns 42

6. Detection using combined Approach 49
   6.1. Approach 49

6.2. Considered Anti-Patterns 50

7. Evaluation 57
7.1. Evaluation of Rules based on Single Traces 57
7.2. Evaluation of Rules based on Time Series 64
7.3. Threats to Validity 78

8. Conclusion 81
8.1. Summary 81
8.2. Retrospective 82
8.3. Future Work 83

A. Rules 87

Bibliography 99

List of Figures

2.1. diagnoseIT approach [HHO+15] . . . 11
2.2. Simplified OPEN.xtrace class diagram . . . 13

4.1. Restricting the problem area within a trace . . . 30
4.2. Execution of the N+1 Query Problem rule . . . 34
4.3. Execution of the Stifle rule . . . 36

5.1. Detection based on time series approach . . . 42
5.2. Regression line through data points [Wik10] . . . 44

6.1. Detection using combined approach . . . 50
6.2. Box plot . . . 52
6.3. Application Hiccup . . . 54
6.4. Execution of the Garbage Collection Hiccups rule . . . 56

7.1. Example trace (excerpt) with N+1 Query Problem . . . 59
7.2. Log events in the XML file . . . 63
7.3. Vertical line through data points . . . 66
7.4. Perfect Ramp of data points' response times . . . 67
7.5. Rise of response times . . . 68
7.6. Rise of response times with outliers . . . 69
7.7. Variance of response times . . . 70
7.8. Time series that contains also GC times . . . 71
7.9. Time series that contains real trace data . . . 72
7.10. Time series that contains Ramp anti-pattern . . . 74
7.11. Time series that contains Ramp and Application Hiccups . . . 75
7.12. Time series that contains overload situations . . . 76
7.13. Time series that contains no overload situations . . . 77

List of Tables

1.1. Synonyms 5

3.1. Considered anti-patterns 17
3.2. Rule annotations 22

7.1. Test Case 1 - Results 66
7.2. Test Case 2 - Results 67
7.3. Test Case 3 - Results 68
7.4. Test Case 4 - Results 69
7.5. Test Case 5 - Results 70
7.6. Test Case 6 - Results 72
7.7. Test Case 7 - Results 73
7.8. Test Case 8 - Results 74
7.9. Test Case 9 - Results 75
7.10. Test Case 10 - Results 76
7.11. Test Case 11 - Results 78

8.1. Possible improvements 84

List of Acronyms

APM application performance management

GC garbage collector

SPA software performance anti-pattern

TSDB time series database

VM virtual machine

List of Listings

3.1. Rule schema 24

4.1. N+1 Query Problem example 32
4.2. N+1 Query Problem solution 32

A.1. N+1 Query Problem rule 88
A.2. The Stifle rule 89
A.3. Expensive Computation rule 90
A.4. Phantom Logging rule 91
A.5. The Ramp rule 92
A.6. Traffic Jam rule 93
A.7. More is Less rule 94
A.8. Application Hiccups rule 95
A.9. Alternative Application Hiccups rule 96
A.10.Garbage Collection Hiccups rule 97

Chapter 1

Introduction

This thesis presents an approach for rule-based detection of software performance anti-patterns (SPAs) [ap00], that are manifested within an applications software. Using the rules, the anti-patterns will be detected automatically.

The intention of this introduction is to clarify, why such an approach is needed. For this purpose, the chapter starts with a short motivation in Section 1.1. The next part, Section 1.2, names the goals of this thesis. An overview of the approach of this thesis is described in Section 1.3 and finally, in Section 1.4, some notations for this thesis are described and the document organization is presented.

1.1. Motivation

Application performance management (APM) provides solutions for monitoring applications to detect performance problems. Performance problems in an enterprise system can have a huge negative impact on the success of the enterprise. A little increase in page load time of the system can decrease sales drastically. Therefore, it is a necessity to detect performance problems as early as possible, at best before they affect end users [Wer15].

Many APM tools have been developed to tackle this problem. An APM tool tracks steps in executions of the monitored application and collects performance data. However, most of the existing APM tools only provide alerting and visualization of performance-relevant measures, not showing the root cause of the problem. There are experts with the required knowledge for the analysis of performance problems, but because they are rare, they are costly and their evaluation of performance problems can be very time consuming and can be wrong. Enterprises could try to setup APM by themselves, but

the initial setup and maintenance of APM is error-prone and frustrating manual task, which again requires experience and expertise.

The main task of diagnoseIT [dIT15] is to systematically detect and diagnose performance problems using formalized expert knowledge. The formalized expert knowledge allows diagnoseIT to automatically execute recurring APM tasks, like the configuration of a meaningful software instrumentation. Based on performance problem diagnosis, diagnoseIT also offers the possibility to isolate root causes of detected problems. The analysis results are provided to the user in the form of a comprehensive report, so that even non-experts can take advantage of the diagnoseIT. The report contains both, qualitative and quantitative information. The qualitative results contain, e.g., the problem's location, its type, and the matching performance anti-pattern for this problem, whereas the quantitative information hold the impact of the problem in numbers, e.g., response times or CPU times. As a part of the qualitative information presented to the user, SPAs have to be detected through the diagnosis to provide the user with appropriate solutions to achieve an increased application performance. The automated diagnosis of these anti-patterns is designed as follows: The expert knowledge contains possible symptoms for performance problems. These symptoms are defined as an extensible set of rules. Thereby, a rule can define symptoms for anti-patterns. When diagnoseIT receives a trace to analyze, it automatically starts applying one rule after another on that trace to get insights. An anti-pattern can be detected, when the analyzed data matches the defined symptoms of a rule [HHO+15]. However, diagnoseIT, in its present condition, has no sufficient amount of rules and thus can detect only few anti-patterns. So the main goal of this thesis is to develop further concepts and rules to detect SPAs in execution traces. Furthermore, as diagnoseIT has no possibility to detect performance anti-patterns, which become visible in a longer observation period of performance data, this thesis proposes a concept to extend the diagnoseIT, so that particular data from execution traces can be saved and then a sequence of data can be analyzed.

There are already existing works for detecting anti-patterns, but none of them specializes on anti-pattern detection by analyzing runtime data that agents from APM tools collected, when the monitored system was executed. The commonly used approach for detecting SPAs is static code analysis. The problem is, for example, that the code to analyze could not be available. Further works in this area specialize, for example, more on a model-based detection approach for anti-patterns by analyzing the software architecture [Tru11]. Approach by Wert uses experiments to systematically detect performance anti-patterns in systems [Wer15], but not considering runtime production data.

1.2. Goals

The goals of this thesis were defined earlier in a proposal submitted in June 2016. The results for each goal will be documented. The goals are listed again in the following.

Evaluation of diagnoseIT and the OPEN.xtrace: The first goal of this thesis is to evaluate the diagnoseIT and the OPEN.xtrace [OHH+16] approach. The OPEN.xtrace can be seen as a universal data format diagnoseIT can analyze. Therefore, every trace, which should be analyzed by diagnoseIT, first has to be converted into the OPEN.xtrace format. So only the metrics the OPEN.xtrace provides will be used in the diagnoseIT analysis. Therefore, the OPEN.xtrace has to be investigated to find out, which metrics are accessible. Furthermore, since diagnoseIT uses a rule engine for its analysis, it has to be investigated, how the rule engine analyzes execution traces.

Access history data in diagnoseIT: One further goal is to find a way to access history data in diagnoseIT, since there are anti-patterns, that become visible over a longer observation period. That means, for some anti-patterns, it is not sufficient to investigate a single trace, but a sequence of more traces is needed to detect the anti-pattern.

Anti-pattern research: The next goal of this thesis is to research, which SPAs exist, how they can be detected, and what data has to be visible for extracting the anti-pattern. Furthermore, deciding if the anti-pattern can be detected with the current diagnoseIT implementation and the available data in the OPEN.xtrace, is also an important objective of this thesis.

Developing concepts and rules: Since diagnoseIT uses a rule engine for its analysis, the anti-patterns have to be detected by applying rules to the analyzed traces. However, the concepts for detecting anti-patterns in single traces and anti-patterns, that need historical data, differ. This is why new concepts for detecting anti-patterns have to be developed. The rules based on the developed concepts have to be implemented in Java code.

Evaluation of the implementation: After prototyping the concepts and rules, the implemented rules will be systematically tested on problematic traces to evaluate the false positive and false negative rate of the rules. Therefore, the rules have to be applied to several traces generated using an example application, in which some of them the anti-pattern is manifested and in some not.

1.3. Overview of the Approach

In this thesis, the diagnoseIT approach is split into three parts. The first analysis approach dedicates itself to anti-patterns, that can be detected by analyzing a single execution trace. This approach is called **detection based on trace data**. The second analysis part is for anti-patterns, that cannot be detected by analyzing a single trace, and the analysis of more traces is needed to detect them. Therefore, trace data is saved into a time series database (TSDB). By investigating the time series data for inconsistencies, the diagnoseIT analysis can detect anti-patterns. The approach for detecting anti-patterns from time series data is called **detection based on time series**. The last part of the analysis is a combined approach. Thereby, when detecting inconsistencies in the time series data, a problematic data point can be assigned to a trace. This trace can then be analyzed further. The third approach is called **detection using combined approach**.

1.4. Document Organization

This section is for describing the typographical conventions in this thesis and for presenting the document structure.

1.4.1. Formalia

To better highlight certain parts of this thesis, the following typographical conventions are introduced:

Paragraphs, column names and important words: When a paragraph starts, the name of the paragraph is highlighted in **bold**. Furthermore, column names of tables and important words are highlighted in **bold**.

Example names: Names for entities or methods in an example are highlighted in *italic*.

`Implementation parts and class diagram entities:` Implementation parts are in `typewriter` font. Words, that are used in a implementation, are also in `typewriter` font. These include, for example, class names and method names. Furthermore, class diagram entities, as they are also used for implementation, are in `typewriter` font.

[References]: Whenever referring to foreign works like papers or referring to web pages, a reference is used, which is linked to an entry in the bibliography, where you can find more details about that reference.

1.4.2. Synonyms

In this thesis, for two terms synonyms are used. These are listed in Table 1.1.

Term	Synonym(s)
Software performance anti-pattern	Performance anti-pattern Anti-pattern
Execution trace	Trace

Table 1.1.: Synonyms

1.4.3. Structure

This thesis is structured as follows:

Chapter 2 – Foundations and State of the Art: In this chapter, the foundations for this thesis are described. The important terminology for this thesis is explained, and a short introduction to SPAs is provided. Also, related work according to this thesis is presented. The related work deals with the anti-pattern detection. The chapter ends with presenting the state of the art, where the diagnoseIT and the OPEN.xtrace are covered, and an introduction to time series databases and the InfluxDB [inf16a] is provided.

Chapter 3 – Rule-based Approach of Anti-Pattern Detection This chapter introduces the various performance anti-patterns, that are considered in this thesis and assigns the anti-patterns to one of the proposed detection approaches, that were introduced in Section 1.3. In this chapter, we also present anti-patterns, that cannot be detected with the current state of the diagnoseIT and the available data in the OPEN.xtrace. Furthermore, the diagnoseIT rule engine is described in more detail and the schema for implementing rules to detect anti-patterns is presented.

Chapter 4 – Detection based on Trace Data In this chapter, we deal with the first approach for anti-patterns, which can be detected by analyzing a single execution trace. We develop concepts and rules to detect the anti-patterns. The rules are implemented using Java [Ora16] and a detailed description of the implementation is provided.

Chapter 5 – Detection based on Time Series This chapter describes the second analysis approach. The second approach is for anti-patterns, that become visible over a longer observation period. For this purpose, the diagnoseIT is extended with a TSDB. The approach is applied to particular anti-patterns, that can now be detected using this approach. We develop concepts and rules for the anti-pattern detection and also implement the rules in Java code.

Chapter 6 – Detection using combined Approach Combining the first two analysis approaches leads to the third analysis approach, that is described in this chapter. The approach links time series data to a corresponding trace, that can then be further analyzed. We apply this approach on two anti-patterns. We develop concepts and rules for the anti-pattern detection and also implement the rules in Java code.

Chapter 7 – Evaluation This chapter presents results of the evaluation of the anti-pattern detection implementation. The evaluation goals are described and how the experiment will be executed. The experiment results are provided and finally discussed.

Chapter 8 – Conclusion The conclusion of this thesis sums up the outcomes of this thesis and outlines the future work.

Appendix A – Rules In this appendix, the implementations of the rules developed in this thesis are provided.

Chapter 2

Foundations and State of the Art

The introduction gave a short motivation on this topic and argued, why the anti-pattern detection in applications software is relevant. The main goal is to detect root causes of performance problems and to provide appropriate solutions for them.

To better understand important terms, that are used throughout this thesis, the first section of this chapter, Section 2.1, provides some descriptions for them. In Section 2.2, a short introduction to SPAs and how they can be detected is provided. Then, related work to this thesis about anti-pattern detection is presented in Section 2.3. After that, a description of the APM tool diagnoseIT is provided and can be found in Section 2.4. In the next part, Section 2.5, the OPEN.xtrace is described. Finally, in Section 2.6, an introduction to TSDBs is provided, and we also present InfluxDB.

2.1. Terminology

Software performance anti-pattern: An anti-pattern describes a poor recurrent approach to design problems which may have a negative impact on particular software quality attributes. In terms of SPAs, the affected quality attribute is the performance of the software [ap00].

Performance measurement: In the context of APM, performance measurement represents the collection of performance-relevant data from a monitored software. Performance measurements usually contain, e.g., the executed method names, method parameters, response times, execution times, exceptions and others [HHO+15].

Performance problem: In the context of performance measurement a performance problem is characterized by undesirable performance measurements decreasing the performance of the software, such as high response times or CPU times [IHE15].

Execution trace: The described structure of the execution trace in this paragraph refers to the OPEN.xtrace [OHH+16]. An execution trace [JSB97] subsumes a logical invocation sequence through the monitored software. A trace consists of a tree structure of subtrace instances. A subtrace is executed in exactly one location, e.g. in a particular computer node of a system. The subtrace, in turn, consists of a tree structure of different callable instances. These represent the execution operations (performance measurements), like CPU time, method name or SQL statement, when the application makes remote calls to a database. However, each APM tool has a different format for its execution traces.

APM tool: An APM tool monitors the performance of software systems. Usually, it uses an agent that instruments the monitored system to collect performance measurements. Every step between the request and the response from the system takes time and resources to compute. The APM tool tracks these steps with the help of the agent and records the time and resources each method call requires and builds an execution trace from it.

DVD Store: The DVD Store [Nov11] is the application we will use for the evaluation in this thesis. It is a web-based shop, where DVDs can be bought, searched and orders can be looked up. For performance testing, performance problem injection was implemented. For example, when enabling the 'Slow Search' function, the search for DVD titles within the store gets slower. We test with the diagnoseIT rule engine and the new implemented rules throughout this thesis, if the rules can diagnose performance problems in the DVD Store.

2.2. Software Performance Anti-Patterns

Software patterns describe common solutions and best practices from expert knowledge in order to solve recurring problems, when building a software. They refer to the software architecture as well as to the software design or to the software development process. Patterns are documented with their name, a description for the considered problem and a solution for the problem.

The counterparts to software patterns are software anti-patterns. Since using software anti-patterns is a frequently made mistake when building a software, they are also often documented with a name, description and a solution for solving the anti-pattern or how to avoid the anti-pattern. Software anti-patterns refer to different quality attributes that are affected by it. In Section 2.1, we already described what a SPA is. Concerning SPAs, the affected quality attribute of the software is the performance. Therefore, when a performance anti-pattern is manifested in a system, the system's performance decreases. Software anti-patterns in literature commonly do not refer to performance problems. They concentrate rather on other quality attributes, like the reusability, modularity and the maintainability of the software [ap00]. In this thesis, we investigate anti-patterns from a performance stand point.
We differentiate between technology-dependent and technology-independent performance anti-patterns. In case of technology-dependent performance anti-patterns misusing a specific technology leads to performance problems. Examples are Hibernate anti-patterns, that are described by Węgrzynowicz [Węg13]. In this thesis, we concentrate on technology-independent anti-patterns, that are caused, for example, by a bad software design or by wrong implementation approaches [Wer15].

The investigated anti-patterns in this thesis all have a negative impact on performance. Throughout this thesis, we will try to detect performance anti-patterns automatically only with the help of runtime data from systems, not with, for example, static code analysis or manual analysis.

2.3. Related Work

There are existing works on the area of anti-pattern detection. Some of them are described in the following.

Wert [Wer15] describes an approach for an automatic, experiment-based diagnosis of performance problems in enterprise systems. The approach tries to detect performance problems and performs a systematic search for the root causes of these performance problems. For this purpose, a taxonomy on recurrent performance problems is used. The approach is applied to 27 anti-patterns, which are described in this publication. However, the approach does not consider runtime production data from systems. Nevertheless, some of the presented anti-patterns will be considered for this thesis, since they can be detected with performance measurements.

Trubiani [Tru11] describes how to give automated feedback on architectural level of the software, when a performance problem occurs. This approach analyzes results from performance analysis (predictive quantitative results) and then suggests the best suitable

architectural reconfiguration. She proposes the framework PANDA (Performance Antipatterns and Feedback in Software Architectures), that performs three main activities. It first specifies anti-patterns, then tries to detect them in software architecture models and finally solves the anti-patterns. These anti-patterns can be the reason for a performance problem and can be used for finding a solution for architectural alternatives. The approach is model-based on the software architecture, and does not consider runtime informations from systems. Therefore, her framework cannot detect anti-patterns by analyzing performance measurements.

Parsons and Murphy [PM08] describe an approach for automatic anti-pattern detection in componend based enterprise systems. Therefore, they introduce several monitoring and analysis methods, that are for identifying relationships and patterns in the monitored data. In order to detect anti-patterns, they use a rule engine that analyzes the system's design model. However, their approach is limited to JEE (Java Platform, Enterprise Edition) performance anti-patterns.

The approach from Peiris and Hill [PH14] uses, like in this thesis, runtime data from systems, when they were executed, to detect anti-patterns. They present a non-intrusive machine learning approach, called NiPAD (Non-intrusive Performance Anti-pattern Detecter). It operates on the monitored system's performance metrics. The approach analyzes the data and then can detect and classify anti-patterns. NiPAD receives two data sets with system-level measurements, like CPU time, memory usage and network usage. In one of the data sets a performance anti-pattern is manifested, and in the other it is not. With machine learning techniques NiPAD learns over time in which data set an anti-pattern is manifested. We do not use machine learning techniques, but a rule-based detection approach. Therefore, our approach does not have to learn in which data set an anti-pattern is manifested.

Heger et al. [HHO+15] describe the idea behind anti-pattern detection in diagnoseIT. Rules from a rule engine are applied to problematic traces. The system can detect anti-patterns by analyzing runtime data from monitored systems. The current diagnoseIT approach has only a small set of rules to detect performance anti-patterns. It also cannot detect anti-patterns, whose effects become visible over a longer observation period. This disadvantage leads to the restriction that anti-patterns can only be detected by analyzing a single trace. Therefore, important anti-patterns, like the Ramp [SW02a], cannot be detected through the diagnoseIT analysis. For this purpose, throughout this thesis, we propose a solution for this and extend the diagnoseIT.

2.4. diagnoseIT

diagnoseIT [dIT15] is an APM tool that utilizes formalized APM expert knowledge to detect and diagnose performance problems and isolate their root causes. It is designed to be independent of specific APM tools, and can be connected with any APM tool to receive execution traces for analysis. Figure 2.1 shows how the diagnoseIT analysis works. The independence from a specific APM tool is achieved through the OPEN.xtrace. More details about the OPEN.xtrace are provided in Section 2.5. A trace is automatically analyzed by the diagnoseIT. diagnoseIT applies predefined rules to the trace, that are executed using the rule engine [Nov16]. Each execution of a rule provides additional insights to the analysis result. The result of the analysis process is a comprehensive report for the user [HHO+15]. Summarized, the diagnoseIT analysis works as follows:

- diagnoseIT receives traces from connected APM tools in OPEN.xtrace format.
- The diagnosis of traces is performed using an extensible set of rules which are applied to the traces.
- Each execution of a rule provides additional insights to the analysis result.
- Based on the insights diagnoseIT obtains by applying rules, it creates problem instances, which hold the performance problem.
- diagnoseIT can alert and propose solutions, when an anti-pattern is detected.

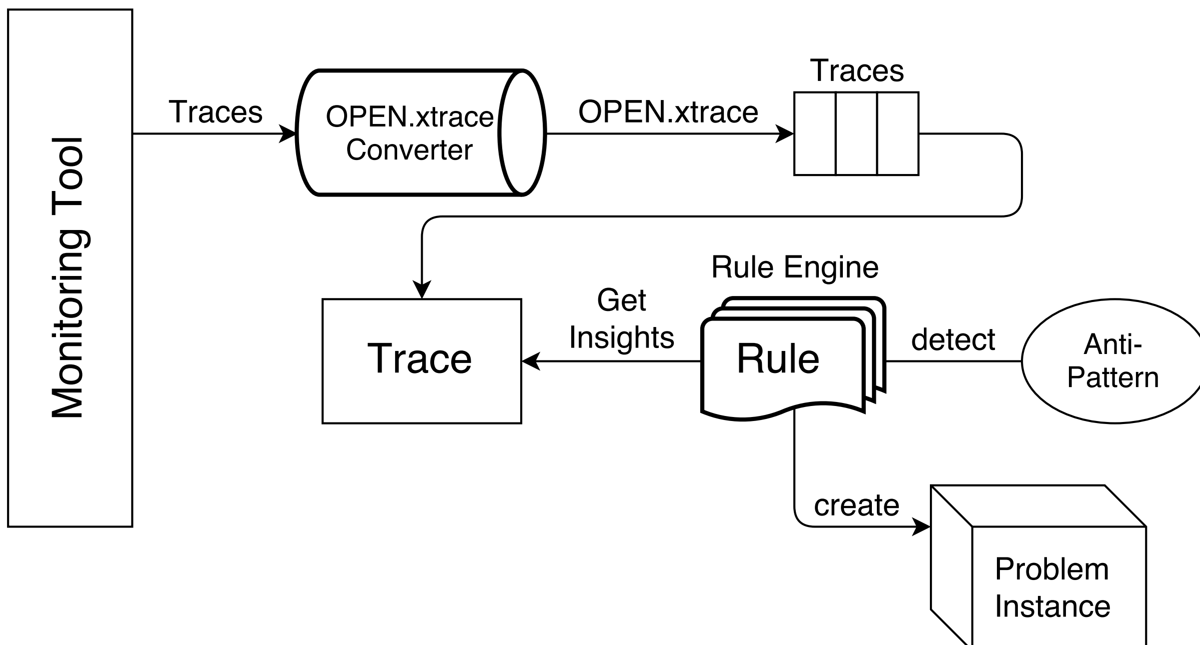

Figure 2.1.: diagnoseIT approach [HHO+15]

More implementation details concerning the diagnoseIT rule engine are provided in Section 3.3.

2.5. OPEN.xtrace

The OPEN.xtrace [OHH+16] is an interface-based specification of a common trace representation. Its intention is to provide a universal data format, that can contain data from different APM tools. This allows the interchange of data and interoperability between APM tools. Furthermore, it serves as an abstraction layer between diagnoseIT and the APM tool diagnoseIT is connected to. As mentioned in the previous section, diagnoseIT uses the data in the OPEN.xtrace for its analysis. The advantage of using the OPEN.xtrace is that diagnoseIT can analyze traces from any APM tool. The traces only have to be converted into the OPEN.xtrace using adapters [OHH+16].

Figure 2.2 shows a simplified class diagram of the OPEN.xtrace. Only the OPEN.xtrace entities are pictured. A `Trace` consists of a tree structure of `SubTrace` instances. So the root of a `Trace` is always a `SubTrace`. A `SubTrace` is executed in exactly one `Location`. When the `Location` changes, another `SubTrace` instance for this `Location` exists.

`SubTrace` instances consist of a tree structure of `Callable` instances, which represent the actual methods, that get executed. The root of a `SubTrace` is a `Callable`. If the `Callable` can be a parent of further `Callable` instances, that are executed in the same `Location`, then it is a `NestingCallable`. When the `Callable` calls further methods, that are not executed in the same `Location`, it cannot be a `NestingCallable`. Only `MethodInvocations` and `HTTPRequestProcessings` are `NestingCallable` instances.

A `MethodInvocation` can call further methods, that are in the same `Location`. In case of a `HTTPRequestProcessing` instance, a service is requested. The service then can execute further methods to satisfy an incoming request.

The `RemoteInvocations`, `DatabaseInvocations` and `LoggingInvocations` cannot be `NestingCallables`. `RemoteInvocations` are methods, that call methods in other `Locations`. Therefore, they also cannot be a `NestingCallable`. By executing a `RemoteInvocation`, the `SubTrace` instance changes. A `LoggingInvocation` is a log event in the monitored software and cannot call further methods. A `DatabaseInvocation` is a method, that calls the database. Because the `Location` changes and no further methods are called by this method, it is not a `NestingCallable`.

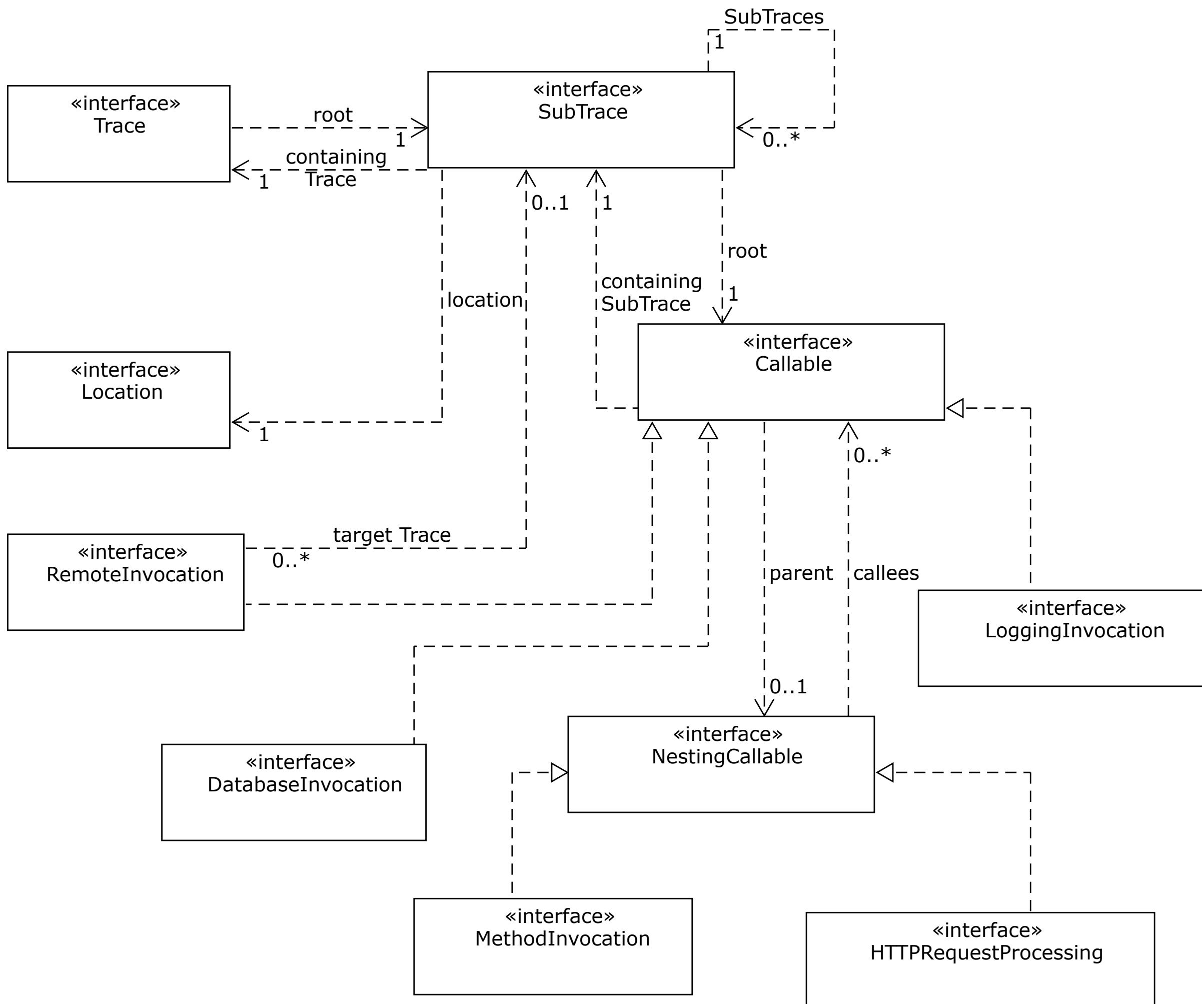

Figure 2.2.: Simplified OPEN.xtrace class diagram

2.6. Time Series Database

Time series data [Mit10] consist of a series of data points, where one data point describes a measurement sample. A data point is characterized by a timestamp, the name of the measurement, e.g., response time and at least one value the measurement has, e.g., 10 ms. The data points are also known as time series data.

A time series database (TSDB) is a database that is optimized for operating on time series data. It saves the data in a chronological order and provides classical database operations like create, read, update and delete. Besides, a TSDB allows the user to organize the time series data with the help of, for example, metadata. Metadata can be placed into the data points. For analyzing time series data, the TSDB provides

mathematical operations and queries. For example, a helpful query is to aggregate time series data [inf16b].

InfluxDB

InfluxDB [inf16a] is a TSDB, that has a SQL-like database language. A data point in InfluxDB consists of `time` (timestamp), a `measurement` and at least one measured value, the `field`. It allows to save metadata, called `tags`, together with the data. This is helpful for organizing the time series data. In InfluxDB, the `measurement` is the database table, in which the `time` is the primary key. `fields` and `tags` are further columns of the database table. The following line creates a database table and simultaneously inserts a data point to it

```
INSERT someMeasurement,tag1=someTag,tag2=anotherTag timeValue=10
```

where `someMeasurement` is the database table and simultaneously the measurement name of the data point, `tag1` and `tag2` are metadata, and `timeValue` is the measurement value itself.

The written data can be queried as follows:

```
SELECT * FROM someMeasurement
```

Further possibilities all around the InfluxDB can be found in its documentation [inf16b].

Chapter 3

Rule-based Approach of Anti-Pattern Detection

In this chapter, we give an overview of the SPAs, that we will consider for the diagnoseIT analysis. We also provide examples for anti-patterns, that cannot be detected in the scope of this thesis. In this chapter, technical details about the diagnoseIT rule engine are provided, so that in the following chapters we can present rules for the performance anti-pattern detection.

In Section 3.1, the three diagnoseIT analysis approaches are shortly introduced and then an overview of the anti-patterns, that will be considered for the next chapters, is provided. Each anti-pattern is assigned to one of the three analysis approaches.

In Section 3.2, we provide examples for anti-patterns, than cannot be detected due to the current diagnoseIT implementation or missing OPEN.xtrace data.

In Section 3.3, design and implementation details on the diagnoseIT rule engine are provided. Besides, the schema for a rule that can be processed by the rule engine is presented.

3.1. Overview of detectable Anti-Patterns

In this section, anti-patterns are presented, that will be investigated further in the next chapters. As already described in Section 2.4, diagnoseIT can detect SPAs by analyzing a single execution trace. We consider these anti-patterns for the **detection based on trace data** approach in Chapter 4. Other anti-patterns cannot be detected by analyzing single traces, and the analysis of more traces is necessary to detect them. Such anti-patterns are considered for the **detection based on time series** approach

in Chapter 5, where particular data from traces is saved into a TSDB. We call data from execution traces, that were executed in the past, also historical data. By saving trace-specific data into the TSDB, a sequence of traces (respectively trace data) can be analyzed. For some anti-patterns, that need a longer observation period of trace data to be detectable, it is meaningful to consider them for the third analysis approach, namely the **detection using combined approach**, which is described in Chapter 6. Actually, these anti-patterns belong to the detection based on time series approach. The difference is, that a rule from the detection using combined approach can, beside detecting an anti-pattern, assign data points from the time series data to a corresponding trace, that can then be analyzed further in the detection based on trace data analysis. This gets relevant, when an analyzed data point from the TSDB is a cause for an inconsistency within the time series data, and therefore maybe shows a symptom of a performance problem.

Considered Anti-Patterns

We only consider SPAs, that can be detected through performance measurements from monitored systems, and not through, e.g., static code analysis or manual analysis. This is because diagnoseIT analyzes data from the OPEN.xtrace, which holds performance-relevant runtime informations from monitored systems. The detectable SPAs are restricted to the data available in the OPEN.xtrace. For detecting a particular SPA specific performance measurements, that match the anti-pattern's characteristics, have to be available. If the OPEN.xtrace does not provide these data, the anti-pattern cannot be detected. Examples for not detectable anti-patterns are provided in Section 3.2.

In this section, the considered SPAs are shortly described and then we assign the anti-patterns to one of the three analysis approaches. A more detailed description and detection concepts for the performance anti-patterns is provided in the particular chapter it belongs to. For all of the presented anti-patterns, except the Circuitous Treasure Hunt anti-pattern [ap00], rules for the diagnoseIT rule engine will be implemented, so that the rule engine automatically can detect the anti-pattern. The rules can also be found in the particular chapter the anti-pattern belongs to.

Table 3.1 provides an overview of the following SPAs. The table shows to which approach the anti-patterns are assigned.

In the following, when talking about traces, it is assumed that the trace is in OPEN.xtrace. The considered anti-patterns are described now.

Anti-pattern	Considered Approach
N+1 Query Problem	detection based on trace data
Circuitous Treasure Hunt	detection based on trace data
The Stifle	detection based on trace data
Expensive Computation	detection based on trace data
Phantom Logging	detection based on trace data
The Ramp	detection based on time series
Traffic Jam	detection based on time series
More is Less	detection based on time series
Application Hiccups	detection using combined approach
Garbage Collection Hiccups	detection using combined approach

Table 3.1.: Considered anti-patterns

N+1 Query Problem: The N+1 Query Problem anti-pattern occurs, when an application first makes an initial query to the database with a large result set (the '1' in 'N+1') and then performs a high amount of similar queries related to the items obtained by the initial query (the 'N' in 'N+1') [SW03]. For investigating the monitored system with regard to the N+1 Query Problem, the analysis of a single trace is sufficient, since a trace holds all required performed queries from the monitored application for the diagnoseIT analysis in `DatabaseInvocations`.

Circuitous Treasure Hunt: When an application communicates with the database to obtain items from a first database table and then uses those results to obtain items from a second database table and so on until the final needed results are on hand, then the Circuitous Treasure Hunt anti-pattern exists in the application [ap00]. For investigating the monitored system for the Circuitous Treasure Hunt anti-pattern, the analysis of a single trace is sufficient, since a trace holds all required performed queries from the monitored application for the diagnoseIT analysis in `DatabaseInvocations`.

The Stifle: In case of the Stifle anti-pattern, an application performs a high amount of similar or equal fine-grained database queries to obtain data from a database [DAKW03]. For investigating the monitored system with regard to the Stifle anti-pattern, the analysis of a single trace is sufficient, since a trace holds all required performed queries from the monitored application for the diagnoseIT analysis in `DatabaseInvocations`.

Expensive Computation: An expensive computation within a software is a method, that causes CPU intensive computations [Wer15]. The Expensive Computation anti-pattern reveals the method that spends most on the CPU. To detect such a method within a trace, the analysis of one trace is sufficient, since the trace holds the executed methods from a monitored application in `Callables`.

Phantom Logging: In an application a Phantom Logging anti-pattern [Nov08] is manifested, when a trace holds a log message, that the application user actually did not want to log. The analysis of a single trace is sufficient, since a trace holds logged messages in `LoggingInvocations`.

The Ramp: When during the execution of a monitored software the time to satisfy a request rises over time, then the Ramp anti-pattern occurs [SW02a]. Since a longer observation period of performance data is necessary to detect the Ramp, more than one traces have to be analyzed and therefore it is considered for the detection based on time series approach.

Traffic Jam: When during the execution of a monitored software a high variance of response times can be observed, then the Traffic Jam anti-pattern occurs [SW02b]. Since a longer observation period of performance data is necessary to detect a Traffic Jam, more than one traces have to be analyzed and therefore it is considered for the detection based on time series approach.

More is Less: When a system is permanently busy with "thrashing", then it will not accomplish real work. When more processes are active than resources exist, such situations appear often within a system and lead to the More is Less anti-pattern [SW02a]. Since a longer observation period of performance data is necessary to detect it, more than one traces have to be analyzed and therefore it is considered for the detection based on time series approach.

Application Hiccups: When during the execution of a monitored software periodically a rise of response times can be observed, then the Application Hiccups anti-pattern occurs [Ten14]. Since a longer observation period of performance data is necessary to detect Application Hiccups, more than one traces have to be analyzed. The Application Hiccups anti-pattern is as an example also considered for the detection using combined approach, since data points, that cause the hiccups, have to be investigated further using the data analysis.

Garbage Collection Hiccups: In case of the Garbage Collection Hiccups anti-pattern, the periodic increase of response times is caused by the garbage collector (GC) [Ten14]. Since a longer observation period of performance data is necessary to detect Garbage Collection Hiccups, more than one traces have to be analyzed. The Garbage Collection Hiccups anti-pattern is as an example also considered for the detection using combined approach, since data points, that cause the hiccups, have to be investigated further using the data analysis.

3.2. Not detectable Anti-Patterns

In the following, we give examples for anti-patterns, that cannot be detected in the scope of this thesis. For each anti-pattern we provide a description and propose a solution. We explain why the anti-pattern cannot be detected.

Large Temporary Objects

Description: Creating large objects in memory causes more memory management activities like, for example, swapping. This can have a negative impact on the system's performance [Kop11].

Solution: One possible solution is to process big files with pipelining. This can prevent that large objects are loaded into main memory [Wer15].

Discussion: The Large Temporary Objects anti-pattern is an example for an anti-pattern, that cannot be detected by the diagnoseIT analysis. The OPEN.xtrace holds no informations about sizes of objects for classifying them as 'large'. Furthermore, to detect if the object is used temporarily, we need static code analysis, but the diagnoseIT analysis is limited to the analysis of runtime data.

Dormant References

Description: Dortmant References are pointers pointing to objects, that will not be used anymore in the future. Due to missing clean-up operations, data structures are steadily growing. This may also lead to increased GC activities [RM07].

Solution: Remove pointers to data structures that will not be used anymore [Wer15].

Discussion: Dormant References lead to higher response times. These are provided by the OPEN.xtrace and can be detected. However, the size of data structures has to be tracked. Such information is missing in the OPEN.xtrace.

One Lane Bridge

Description: One Lane Bridges are points in the execution of software, where only one or a few processes can continue to execute further. This is often due to database locking or synchronization points [ap00].

Solution: One solution would be to decrease the time a thread holds a resource in a synchronization point, so that other threads can access that resource faster. Another solution is to find a way, so that multiple resources can access the resource simultaneously [Clo15].

Discussion: The One Lane Bridge anti-pattern cannot be detected with the available data in the OPEN.xtrace. What would be needed to detect it, is, for example, samplings of thread status to analyze, what the thread is doing during execution.

Dispensable Synchronization

Description: The Dispensable Synchronization describes the problem of unnecessary over-synchronization. Long code sequences, that are synchronized, can have a negative impact on performance [Gra10].

Solution: Refactoring the code and removing unnecessary locking areas is one solution to the Dispensable Synchronization. Another one is to reduce the holding time of a resource [Wer15].

Discussion: The Dispensable Synchronization anti-pattern cannot be detected with the current diagnoseIT approach, because static code analysis is required to identify unnecessary locking areas.

The Blob

Description: When there is one controller class in the software, that does everything, then this class is the Blob. Other classes only serve as data containers [ap00].

Solution: The software design has to be refactored [ap00].

Discussion: The extreme message traffic between the one controller class and the smaller classes leads to performance problems, like high response times. These can be detected with the OPEN.xtrace. However, the controller class has to be detected with static code analysis, but the diagnoseIT analysis is limited to the analysis of runtime data.

Empty Semi Trucks

Description: In case of the Empty Semi Trucks anti-pattern, a software sends many small messages to another communication point. Each message contains beside actual payload, additional overhead, like message meta-data. The size of actual payload of such small messages is low [SW03].

Solution: The solution is to aggregate many small messages into one message to save processing overhead [Wer15].

Discussion: To detect the Empty Semi Trucks anti-pattern, we need the payload size of sended messages. The OPEN.xtrace does not provide this data.

3.3. diagnoseIT Rule Engine

diagnoseIT uses a rule engine [Nov16] to systematically apply predefined rules on incoming traces. The traces are obtained from the APM tool diagnoseIT is connected to. In the diagnoseIT implementation, the rule engine is called `DiagnosisEngine`. Its core element is a `Session`, that executes the rules on a trace. Each execution of a rule adds additional insights to the analysis result. A `Session` can store these intermediate results with the help of a `SessionResultCollector` and according to them it decides, which rule to execute next. An execution of a rule can enrich a trace with a `Tag`, which represents

such an intermediate result. An enrichment with a `Tag` can activate further rules, that will be executed on that trace. This process can lead to a chain of rule executions in a `Session`, which together form the final result of the trace analysis.

Annotations

Rules, tags and intermediate results from rules are implemented using annotations. An annotation looks as follows

```
@annotation(param1, param2, ...),
```

where `param` is a parameter, that can be passed to the annotation. Table 3.2 provides an overview of the available annotations with their parameters, that can be used in connection with rules (optional parameters are marked with a *):

Annotation	**Parameters**
@Rule	`name`: String * `description`: String * `fireCondition`: String[] *
@TagValue	`type`: String `injectionStrategy`: InjectionStrategy *
@SessionVariable	`name`: String `optional`: boolean *
@Condition	`name`: String * `hint`: String *
@Action	`resultTag`: String `resultQuantity`: Action.Quantity *

Table 3.2.: Rule annotations

The available annotations are described now in more detail.

Rule This annotation is for annotating a Java class as a rule, that can be executed using the `DiagnosisEngine`. Parameters, that can be passed to this annotation are a `name`, a `description` and a `fireCondition` for this rule. The first two are Strings and the `fireCondition` is an array of Strings. The `fireCondition` parameter is a condition, that can define at what time the rule will be fired through the rule engine, that means, which `Tags` have to be already available.

TagValue This annotation is for injecting results from previously executed rules to a rule that is supposed to execute next. The results are stored in `Tags`. The rule can only fire, if all requested `Tags` are available. The annotation has two parameters. The `type` parameter must be passed to the annotation and is a String. It defines the type/name of the `Tag`, that should be injected. The second parameter is the `injectionStrategy`, that should be used. Two strategies are possible. The first strategy is the `BY_VALUE` strategy, where only the value of a `Tag` is injected. The second strategy is the `BY_TAG` strategy, where the `Tag` itself will be injected to the rule. The default `injectionStrategy` is `BY_VALUE`. The type of the parameter is `InjectionStrategy`. Notice that the rule, that should be executed, can only request results from rules, that are on the same execution path.

SessionVariable Rules are executed in a `Session`. When the diagnoseIT analysis starts, `SessionVariables` can be passed to the rule engine. This annotation is used to inject a variable from the `SessionVariables` to a rule. The first parameter of the annotation is the `name` of the variable, that should be injected to the rule and must be passed. Here, a String is expected. A second parameter of boolean type, called the `optional` parameter, can be passed to the annotation and defines, if the variable must be provided or not, so that the rule can execute. Here, a boolean value is expected. When the value is `false`, the variable must be provided. If this is not the case, the analysis will stop and the rule engine throws an exception. The other possible value is `true` and defines, that the existence of the variable is optional. The default value is `false`.

Condition The rule engine checks this annotation, after all requested `Tags` and `SessionVariables` are injected to the rule. The method within the rule annotated with this annotation represents a condition, that must be true in order to fire the rule. If the condition fails, the rule will not be executed. The annotation has two parameters, that are optional. The first is the `name` for the condition and the second parameter are `hints` for how the condition can be passed. Both parameters are Strings.

Action A rule can specify only one method with this annotation. It is also called the action method of the rule. The action method can be executed, when:

- the `fireCondition` of a rule is satisfied and
- all requested `Tags` and `SessionVariables` are available and
- all condition methods are satisfied.

The annotation has two parameters. The first one, the `resultTag` parameter, must be passed to the the annotation. It is a String. This is the `Tag` the rule can produce, when it is executed. Any object or value can be returned by the rule, that is then wrapped into the `resultTag`. A second parameter can be passed to the rule. It defines, if only a single `Tag` can be produced by the rule, or multiple `Tags`. This is the `resultQuantity` parameter, that is of type `Action.Quantity`. Two values are possible. When a single `Tag` should be produced by this rule, the value of this parameter has to be `SINGLE`, otherwise the value has to be `MULTIPLE`. When multiple `Tags` are produced by the rule, the return type of the action method has to be either an array or a collection and for each produced `Tag` an element within the array/collection with the type of the `resultTag` is created. The default value of the `resultQuantity` parameter is `SINGLE`.

Rule Schema

After the different annotations for rules were discussed, the rule schema will be presented. The reader of this thesis then should know, how he can write his own rules for analyzing traces. Especially for future work on this topic this could be helpful.

Listing 3.1 shows the rule schema as Java code.

Listing 3.1 Rule schema

```
1  @Rule(name = "Rule name", description = "Some description", fireCondition = { "TagXY" })
   public class RuleSchema {

       // InjectionStrategy.BY_TAG also possible
5      @TagValue(type = "RequestedTag", injectionStrategy = InjectionStrategy.BY_VALUE)
       private String ValueOfRequestedTag;

       // optional = true also possible
       @SessionVariable(name = "SomeVariableToInject", optional = false)
10     private String sessionVariableToInject;

       @Condition(name = "SomeCondition", hint = "This condition can be passed by ...")
       public boolean condition() {
           return true;
15     }

       // Action.Quantity.MULTIPLE also possible
       @Action(resultTag = "resultOfThisRule", resultQuantity = Action.Quantity.SINGLE)
       public String action() {
20         return "Some value, that is wrapped into the resultOfThisRule Tag";
       }
   }
```

A rule is a Java class and is annotated with `@Rule`. An intermediate result is requested by the rule by annotating a variable with `@TagValue`. The `type` parameter of the annotation specifies the type/name of the requested `Tag`, whose value should be injected to the variable. A session variable can be injected to the rule by annotating a variable with `@SessionVariable`. The `name` parameter specifies, which session variable to inject. The `@Condition` annotation annotates a method within the rule. The condition of the annotated method must be true in order to fire the rule. A method annotated with `@Action` contains the code the rule can execute. When the method is executed, the rule can produce `Tags`.

Chapter 4

Detection based on Trace Data

In this chapter, SPAs that can be detected based on a single trace data are presented. In Section 4.1, a brief overview of the approach is presented.

In Section 4.2, generic rules for restricting the problem area within a trace are explained. These rules exclude branches of the trace tree, that do not contribute to performance problems. They exclude subtraces and callables, that do not cause high response times. These rules were available before this thesis for an older diagnoseIT version and are only adapted for the new diagnoseIT rule engine. We later describe in more detail the rules for detecting SPAs. The rule for detecting the N+1 Query Problem and the rule for detecting the Stifle anti-pattern make use of the results from the generic rules described in this part, while the other rules for detecting anti-patterns are independent from them.

In Section 4.3, first the investigated performance anti-patterns are described and then solutions to the anti-pattern are proposed. We analyze, how the anti-pattern can be detected with the current diagnoseIT implementation and the available data in the OPEN.xtrace. For all anti-patterns, except the Circuitous Treasure Hunt anti-pattern, we will implement a rule for detecting the anti-pattern. We propose a detection concept for the implementation of the rule. Furthermore, implementation details for the detection are provided. Java code excerpts for the rules are in Appendix A.

In the following, when talking about analyzing traces, it is assumed that the trace is in OPEN.xtrace format.

4.1. Approach

The detection based on trace data approach was already explained in large part in Section 2.4, and Figure 2.1 shows the approach for detecting anti-patterns with single traces.

When diagnoseIT receives a problematic trace from an APM tool, the detection based on trace data analysis is automatically triggered. The diagnoseIT rule engine applies rules to the trace to get insights. Beside performance problems, the rules also can detect performance anti-patterns. Therefore, a rule can represent an anti-pattern by describing its characteristics. The rule then tries to detect inconsistencies within the trace data and investigates if the analyzed data matches the anti-pattern's characteristics. If so, the rule detects the anti-pattern. For example, if an anti-pattern is detected through high response times, the high response times have to be manifested in the trace.

4.2. Narrowing down a Trace

The generic rules for restricting the problem area within a trace are implemented as follows. diagnoseIT receives an incoming trace from a connected APM tool and starts analyzing it. In Figure 4.1, we see an example trace. The trace has a response time of 5000 ms. In the first step the trace is narrowed down to its subtraces with a specific rule. Thereby, problematic subtraces are marked with a tag. A subtrace is considered as problematic, when the response time of the subtrace is equal or higher than a baseline value, which is by default 1000 ms.

After marking the problematic subtraces, the condition of the next rule gets true and the rule analyzes the problematic subtraces further. The problematic subtrace in this example is the one with root *A*. The nodes in the subtrace represent callables. By analyzing the subtrace, the rule begins at root *A* and traverses the tree to find the **Global Problem Context** within the subtrace, finding the node *E* with a response time of 3800 ms. A node is the Global Problem Context within a subtrace, when its response time is at least 1000 ms (default baseline value) and when the sum of the response times of all other nodes on the same level with the same parent is under 1000 ms. For this purpose, the rule searches the deepest level possible in the subtrace, for which this applies. In Figure 4.1, the deepest level is the one with node *D* and *E* (with parent node *B*), because in the level with node *G*, *H* and *I*, there exist two nodes with a response time $>=$ 1000 ms. By defining a node as the Global Problem Context, other nodes on the same level and on higher levels are not considered anymore for the analysis. That is the reason why node *H* is not considered as the Global Problem Context. If that would be the case,

node *I* would not be considered anymore for the analysis, because it is on the same level. However, it has a response time of 1000 ms and therefore maybe contains a performance problem, which would not be analyzed.

After finding the Global Problem Context within the subtrace (a tag was added to node *E*), the next rule searches for the **Root Causes** of the high response times, thus for the time wasting operations. Only node *H* calls further methods, namely 500 *executeQuery()* methods. Each of these queries is a `MethodInvocation`, that holds a `DatabaseInvocation` as its child. The `DatabaseInvocation`, in turn, holds the query to the database and each of these queries has an exclusive time of 4 ms. The rule aggregates `DatabaseInvocations` with same SQL Statements to a root cause object, when their summed up exclusive time is high (>=1000ms exclusive time). Assuming that all `DatabaseInvocations` have the same SQL statement, all of them are aggregated into the root cause object. When, for example, there would be one `DatabaseInvocation` with a different SQL statement that has a high exclusive time (>=1000ms exclusive time), and the exclusive times of the other `DatabaseInvocations` are low, then only this `DatabaseInvocation` would be aggregated into the root cause object. However, the exclusive times of the Root Causes with same SQL statements in the tree are summed up. The total exclusive time of all `DatabaseInvocations` behind the *executeQuery()* methods amounts 2000 ms. So the exclusive time of node *H* amounts actually 0 ms, because it has a response time of 2000 ms and it calls only further methods that hold `DatabaseInvocations` as children, which have the same amount of exclusive time. Node *G* is a leaf node and has an exclusive time of 400 ms. Node *I* is also a leaf node with an exclusive time of 1000 ms. The exclusive times are arranged:

1. The `DatabaseInvocations` behind the *executeQuery()* methods total 2000 ms
2. Node *I* total 1000 ms
3. Node *G* total 400 ms
4. Node *H* total 0 ms

The `DatabaseInvocations` behind the *executeQuery()* methods (>=1000ms exclusive time) are tagged as Root Causes and node *I* is also tagged as root cause (>=1000ms exclusive time), since it does not call further methods. Node *G* and node *H* both will not be tagged as Root Causes (<=1000ms exclusive time).

After finding the Root Causes, the rule for finding the **Local Problem Context** within the subtrace is applied. The node *H* is tagged as a Local Problem Context within the subtrace, since in its subtree there are Root Causes and because of its high response time (2000 ms). Since node *I* has no children, it will also be tagged as a Local Problem Context, making it a root cause and a Local Problem Context at the same time. From the nodes considered as a Local Problem Context, diagnoseIT builds problem instances.

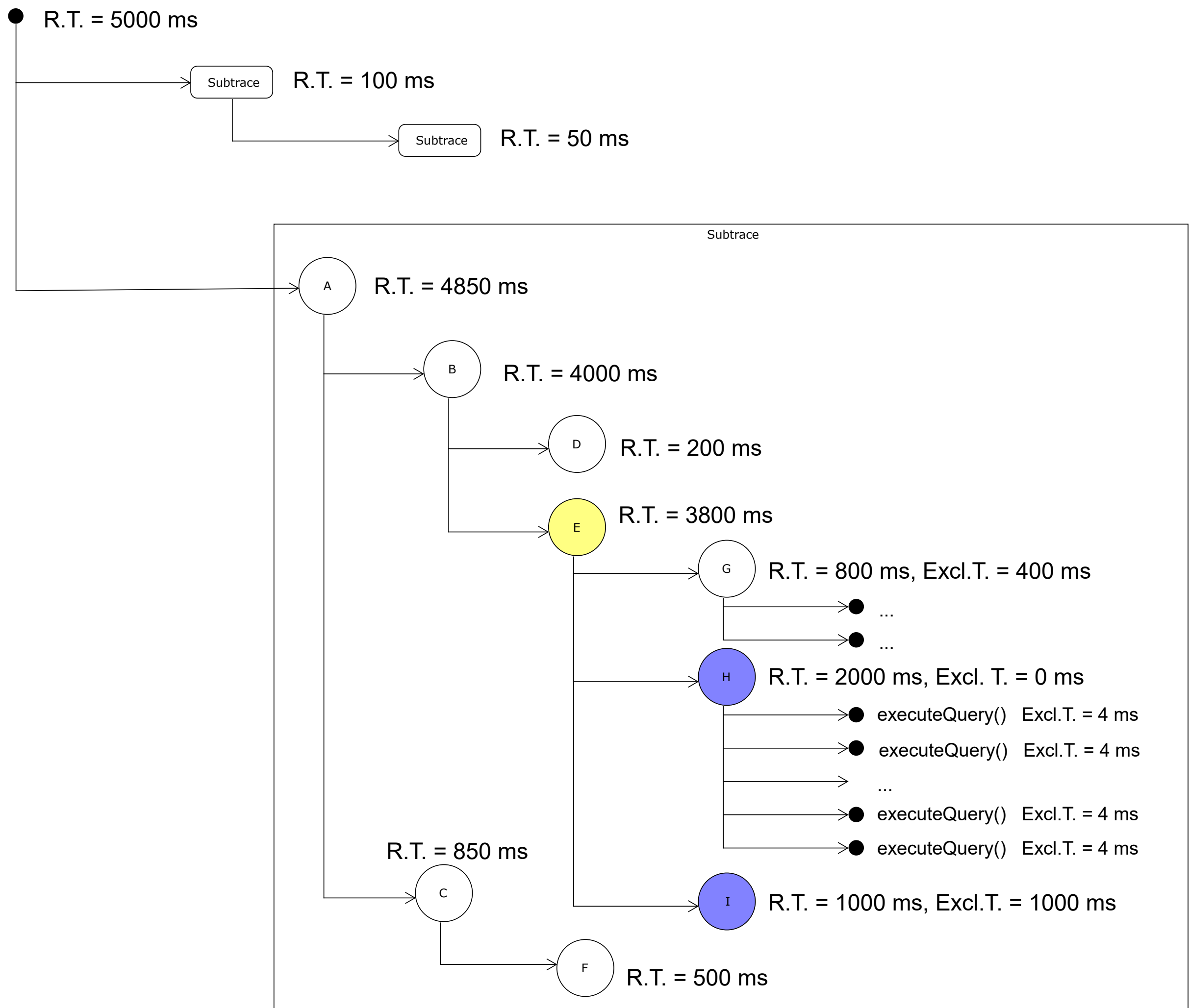

Figure 4.1.: Restricting the problem area within a trace

Summarized the steps for building problem instances are the following:

1. diagnoseIT receives a problematic trace, containing a performance problem
2. diagnoseIT narrows down from the trace to its subtraces and finds the problematic subtraces
3. diagnoseIT searches for the **Global Problem Context** within the problematic subtraces
4. diagnoseIT searches for the **Root Causes** within the problematic subtraces and aggregates them to an object
5. diagnoseIT searches for the **Local Problem Contexts** within the problematic subtraces and creates problem instances from them

6. By further analyzing the Local Problem Context and its associated Root Causes diagnoseIT can detect anti-patterns, like the N+1 Query Problem

diagnoseIT tells the user in its report what the Global Context (Global Problem Context), Problem Context (Local Problem Context), Root Causes and the matching anti-pattern for the performance problem of a trace is.

4.3. Considered Anti-Patterns

The following anti-patterns can all be detected by analyzing a single execution trace. For each of them we provide a description, propose a solution, and describe how it can be detected. For all anti-patterns, except the Circuitous Treasure Hunt anti-pattern, we then propose a design for a rule that we are going to implement. How the rule is concretely implemented is described after the proposed rule design. Simplified versions of Java code from the rules are in Appendix A.

The first three presented anti-patterns are the N+1 Query Problem, the Stifle and the Circuitous Treasure Hunt. We describe in the following that the communication with a database leads to performance problems. However, the anti-patterns are not only related to databases. The communication with other remote data sources can also cause one of the anti-patterns. In this thesis, we are limited to databases as we can access SQL statements from the OPEN.xtrace.

N+1 Query Problem

Description: The N+1 Query problem is a performance anti-pattern related to performing many remote calls to a database. One database query with a larger result set is followed by many small database queries to items in the initial query.

Listing 4.1 shows an example. The problem is that `getStudents()` issues one database query and the call of `getGrades(Student student)` in the for loop issues n further database queries, where each query adds some overhead to the processing. If the `getStudents()` method loads a large result set into the `Student_List`, the overhead could be not acceptable anymore [Pha].

Listing 4.1 N+1 Query Problem example

```
// ...

// Database query behind getStudents(): 'SELECT * FROM Student'
Student_List = getStudents();

for (Student : Student_List) {

      // Database query behind getGrades(Student student):
      // 'SELECT * FROM grades WHERE matriculationNumber = ...'
      Student_Grades = getGrades(Student);

      //...
}
```

Solution: Two solutions are proposed. The first solution to avoid the communication overhead is to use one query with a join operation to fetch all data at once, so that the for loop in Listing 4.1 is not needed anymore [HHO+15].

The second solution is to first query all results, and then to iterate over the results. Listing 4.2 shows the solution. There are no more database queries in the for loop in Listing 4.1. All grades are fetched at once before the for loop, and then by iterating through the results, the grades for a particlar student are selected [Pha].

In both solutions, the 'N' part of the 'N+1' is omitted.

Listing 4.2 N+1 Query Problem solution

```
// ...

// Database query behind getStudents(): 'SELECT * FROM Student'
Student_List = getStudents();

// Database query behind getGrades(List Student_List):
// 'SELECT * FROM grades WHERE matriculationNumber in (...)'
All_Student_Grades = getAllGrades(Student_List);

for (Student : Student_List) {

      // getGrades selects the grades for a particular student
      Student_Grades = getGrades(All_Student_Grades,
            Student.matriculationNumber);

      //...
}
```

Detection: After the generic rules for restricting the problem area within a trace are executed, the only OPEN.xtrace measurements that are needed for detecting the N+1 Query Problem are SQL statements. Therefore, the N+1 Query Problem can be detected. When a method within a trace performs a lot of SQL statements, that are similar, it can be assumed that the N+1 anti-pattern is manifested in the trace. But this approach is not sophisticated enough. It would be better to check if the first query differs from the remaining queries, which are similar to each other. The assumption then would be that the first query is the one with the larger result set followed by the 'N' remaining queries. For a safe detection of the N+1 anti-pattern the SQL statements would have to be analyzed to detect relationships betweeen them, but this would go beyond the scope of this thesis.

Rule design: For this thesis, the N+1 Query Problem will be detected as follows. When diagnoseIT receives a trace, the generic rules for restricting the problem area within a trace are applied to it. The N+1 Query Problem rule then investigates, if there is a `NestingCallable` that holds `DatabaseInvocations` as children. If so, the rule analyzes the SQL statements from the `DatabaseInvocations` and checks, whether one SQL statement is followed by many equal SQL statements, that differ from the first query.

Implementation: The N+1 Query Problem Rule is fired after the generic rules to restrict the problem area within a trace are executed (see Figure 4.2). The N+1 Query Problem analyzes the `DatabaseInvocations` behind the *executeQuery()* methods from Figure 4.1 further to check if the N+1 Query Problem can be indicated from them. A snippet of Java code is provided in Listing A.1. The implementation is as follows. The rule for detecting the N+1 Query Problem requests in line 8 the root causes, that were aggregated to an object by the generic rules. We first check, whether the type of the object is `DatabaseInvocation`. When not, the rule returns false. When the type is `DatabaseInvocation`, the parent node of one of the root causes is accessed. This is an `executeQuery()` method and it is from type `MethodInvocation`. Then, in turn, we access the parent node of the `executeQuery()` method. Its children are all `executeQuery()` methods on the same tree level. This step is important, since the aggregated root cause object only holds `DatabaseInvocations` with same SQL statements. When the `executeQuery()` method has no parent, the rule returns false. Otherwise, we use in line 21 of the rule the `TreeIterator`, to iterate over the subtree with the parent as the root node. We check in line 26 for each `Callable`, if its type is `DatabaseInvocation`. If so, the rule converts the `Callable` into `DatabaseInvocation` to access the `getSQLStatement` method.

After iterating through all nodes of the subtree, we have a list of `DatabaseInvocations`. We then check in line 37, whether the second `DatabaseInvocation` holds another SQL statement than the first `DatabaseInvocation`. If true, we count how many of the remaining `DatabaseInvocations` in the list have the same SQL statement as the second `DatabaseInvocation`. We compare in line 50 the amount of the 'N' queries with a predefined threshold. If the amount exceeds the value of the threshold, the rule returns true and has detected with high probability the N+1 Query Problem. The default value for the threshold is set to 10.

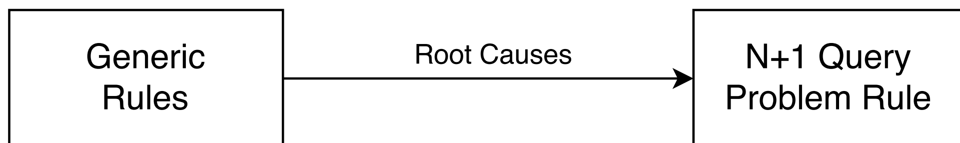

Figure 4.2.: Execution of the N+1 Query Problem rule

The Stifle

Description: In case of the Stifle anti-pattern, an application performs a lot of small similar fine-grained database queries to obtain data from the database. The problem is that each query adds a certain amount of a data overhead. The higher the amount of these small queries, the higher the risk that the performance of the system decreases.

Solution: The solution to the Stifle is to aggregate the small queries to one larger or just a few database queries, so that the overhead for the database communication reduces.

Detection: The Stifle anti-pattern can be detected in a similar way to the N+1 Query Problem. The N+1 Query Problem is also a Stifle, since a lot of small queries are performed from the system. After the generic rules prepared the trace, the only needed data from the OPEN.xtrace are SQL statements to detect it. Therefore, this anti-pattern can be detected by the diagnoseIT. To detect if the performed queries are fine-grained, the analysis of the SQL statements is required. So we choose a simplified implementation and check if the monitored system performs a high amount of similar SQL statements.

Rule design: For this thesis, The Stifle will be detected as follows. The Stifle rule is executed after the rule for detecting the N+1 Query Problem within a trace. When the result of the N+1 Query Problem rule is true, then the Stifle rule immediately

stops executing and also returns true. Otherwise the Stifle rule investigates, if there is a `NestingCallable` that holds `DatabaseInvocations` as children. When yes, the rule counts how many of the `DatabaseInvocations` hold equal SQL statements. If there is particular amount of same statements, then the rule detects with high probability the Stifle anti-pattern.

Implementation: The idea of the Stifle rule is to count the `DatabaseInvocations` behind the *executeQuery()* methods from Figure 4.1 to check, if there are many equal queries. A snippet of Java code is provided in Listing A.2. The implementation is as follows. The rule for detecting the Stifle anti-pattern requests in line 12 the result of the N+1 Query Problem rule and in line 8 an aggregated root cause object from the generic rules executed before (see Figure 4.3). When the N+1 Query problem was detected within the aggregated object, then the Stifle rule immediately stops executing for that object and returns in line 19 true, since the N+1 Query Problem is more specific than the Stifle anti-pattern. However, if the N+1 Query Problem was not detected within that object, the rule executes further.

The rule accesses the parent node of the root causes. This is an `executeQuery()` method and it is from type `MethodInvocation`. Then, in turn, we try to access the parent node of the `executeQuery()` method. Its children are all `executeQuery()` methods on the same tree level. This step is important, since the aggregated root cause object only holds `DatabaseInvocations` with same SQL statements. We use the `TreeIterator` to iterate over the subtree with the parent as the root node. We check for each `Callable`, if its type is `DatabaseInvocation`. If so, the rule converts the `Callable` into `DatabaseInvocation` to access the `getSQLStatement` method. After iterating through all nodes of the subtree, we have a list of `DatabaseInvocations`.

If the `executeQuery()` methods have no parent, the rule works with the available root causes, converts them into `DatabaseInvocations` and puts them into the list.

We create in line 28 of the rule a `HashMap` with key SQL statement and a amount value. For each `DatabaseInvocation`, that has a different SQL statement, in line 41 an entry in the HashMap is created and the amount value is set to one. If then a next `DatabaseInvocation` holds a SQL statement that is already considered in the `HashMap`, the `HashMap` does not create a new entry, but in line 37 the associated amount value to that SQL statement is increased by one. After this process for all `DatabaseInvocations` is finished, the rule iterates through the entries of the `HashMap`, and compares the amount value for each SQL statement with a predefined threshold. The default value for the threshold is set to 10. If one of the amount values exceeds the threhold, then the rule detected with high probability the Stifle anti-pattern.

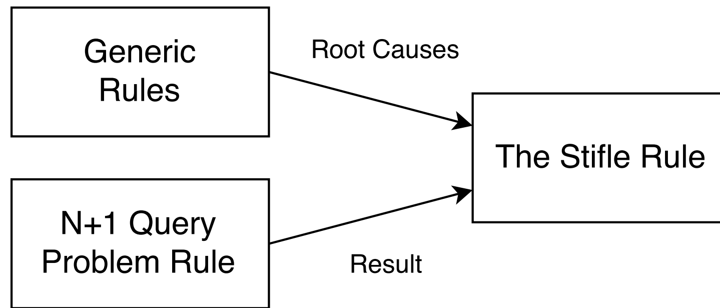

Figure 4.3.: Execution of the Stifle rule

Circuitous Treasure Hunt

Description: The Circuitous Treasure Hunt anti-pattern is similar to the N+1 Query Problem. An application communicates with a database to retrieve data from one of the database tables. The results then are used to retrieve data from another database table. This process repeats as long as the final results are on hand.

The Circuitous Treasure Hunt anti-pattern is a N+1 Query Problem, when the application only uses one query that is followed by n further queries to items in the initial result.

Solution: The solution to this anti-pattern is to refactor the data organization of the application. The refactoring should ensure that connected data from the database is obtained in a simpler way [Wer15].

Detection: The Circuitous Treasure Hunt anti-pattern can be detected in similar way to the N+1 Query Problem and the Stifle anti-pattern. After the generic rules prepared the trace, the only needed data from the OPEN.xtrace are SQL statements to detect it. Therefore, this anti-pattern can be detected by the diagnoseIT rule engine. However, to differentiate from the N+1 anti-pattern, for the Circuitous Treasure Hunt analysis of SQL statements is needed to detect relationships between the statements. The analysis of SQL statements would go beyond the scope of this thesis and therefore there will be no implementation for the detection of this anti-pattern.

Expensive Computation

Description: If an application has methods with CPU intensive computations, the detection of the Expensive Computation anti-pattern reveals the method, that executed

most on the CPU. An Expensive Computation is more like a generalization of other anti-patterns, like the Tower Of Babel [SW03] or the Insufficient Caching anti-pattern [NR09], which lead to CPU intensive applications. However, the detection of an Expensive Computation can be helpful, because the performance problem can be restricted to one problematic method, which then can be further analyzed for more concrete anti-patterns.

Solution: The solution to the Expensive Computation anti-pattern depends on concrete root causes. Root causes are, for example, the Tower Of Babel or Insufficient Caching anti-pattern. The Tower of Babel is caused by processes in a system that exchange a lot of data between each other, but internally use different representation formats for their data. Since translation of formats is expensive and problematic paths are used frequently, Smith and Williams [SW03] propose to use the Fast Path performance pattern [SW].

To save overhead when performing repeated database queries, a cache should be used. Insufficient Caching means the improperly usage of a cache. The cache should be properly set up for an efficient usage of data [Wer15].

Detection: Expensive Computations can be detected by the diagnoseIT. Required data from the OPEN.xtrace are exclusive CPU times. Exclusive CPU time describes the time a method spends on the CPU itself and excludes the CPU time from the methods it calls. The CPU bound slowest method is then found through a comparison of the exclusive CPU times from all methods an application executes.

Rule design: The rule for detecting an Expensive Computation within a trace is always fired, when the detection based on trace data analysis is triggered. The diagnoseIT rule engine receives a trace and investigates if the trace is problematic. The trace is considered as problematic, when its response time is equal or higher than 1000 ms. If so, the rule compares all exclusive CPU times of the trace `Callables` to find the `Callable` with the highest exclusive CPU time. The rule considers this `Callable` as an Expensive Computation, when the ratio of its exclusive CPU time to the trace response time is high.

Implementation: The Java code of this rule is provided in Listing A.3. When the diagnoseIT rule engine receives a trace for analysis, the rule checks in line 23, whether its response time is equal or higher than 1000 ms. If so, the rule iterates in line 35 through the trace and compares for each `MethodInvocation` their exclusive CPU times (only `MethodInvocation` provides CPU times). The rule then takes the `MethodInvocation`

with the highest exclusive CPU time, and calculates in line 44 the ratio of its exlusive CPU time to the trace response time. If the ratio exceeds a threshold, the rule returns the `MethodInvocation`. The default value for the threshold is 0.10.

Phantom Logging

Description: When during the execution of an application log messages are created, which actually do not have to be created in the current log level, then in the application the Phantom Logging anti-pattern is manifested. For example, when in the info log level, the following String is created, but is not logged:

```
logger.debug("Example log message" + var)
```

If the String consists of high amount of characters and the application creates many of them, the performance of a system can seriously decrease [Nov08].

Solution: The solution to the Phantom Logging anti-pattern is easy [Nov08]:

```
if (logger.isDebugEnabled()) logger.debug("Example log message" + var)
```

The String is not built until the logger is in debug mode.

Detection: The Phantom Logging anti-pattern in its classical meaning cannot be detected by the diagnoseIT analysis, because a trace only holds messages, that were really logged by the monitored system. Therefore, for this thesis, the meaning of this anti-pattern is modified, but is still relevant. The OPEN.xtrace provides the `LoggingLevel` of a `LoggingInvocation`. When the trace contains a `LoggingInvocation` with debug `LoggingLevel`, but the user of the monitored application assumed that the logger of his application is in info mode, then he logged a message which he actually did not want to log. Or perhaps such a message was logged due to a bug in the system. We will call this also a 'Phantom Log'.

Rule design: When the diagnoseIT receives an incoming trace, the rule engine iterates through the `Callables` of the trace and searches for `LoggingInvocations`. A threshold describes the assumption in which log mode the monitored system was when the trace was created. If now a `LoggingInvocation` with a `LoggingLevel` lower than the threshold is found, the Phantom Logging anti-pattern is detected.

Implementation: The Java code of this rule is provided in Listing A.4. The Phantom Logging Rule is always fired, when diagnoseIT receives a trace to analyze. With the help of an `enum` the different `LoggingLevel` are described. In the constructor of the `enum` an integer value, called `level`, is passed to the different `LoggingLevels` to define in line 38 that `ALL` is level zero, `TRACE` is level one, `DEBUG` is level two, `INFO` is level three, `WARN` is level four, `ERROR` is level five, `FATAL` is level six and `OFF` is level seven. The threshold in line 5 describes the assumption in which of the `LoggingLevel` the monitored system was when the trace was created. The rule works as follows: It iterates in line 14 over all `Callables` of the trace and searches for `LoggingInvocations`. If one is found, in line 22 the `LoggingLevel` of that `LoggingInvocation` is accessed. To get the level behind the `LoggingLevel` the current analyzed `LoggingInvocation` has, it is compared to the `LoggingLevel` enum and in line 25 the integer value is calculated. If now the current analyzed `LoggingInvocation` has a lower `level` than the theshold `level`, the Phantom Logging anti-pattern is detected.

Chapter 5

Detection based on Time Series

Before this thesis, diagnoseIT could not detect performance anti-patterns, that get visible in a longer observation period, as they cannot be detected with the analysis of a single trace and therefore the analysis of more traces is needed.

In this chapter, we propose a new approach to detect anti-patterns, that get visible over a longer observation period. The new approach will be applied to three performance anti-patterns, that can now be detected by the diagnoseIT and is called from now on **detection based on time series**. More details on the approach are provided in Section 5.1.

In Section 5.2, the considered anti-patterns for the new diagnoseIT approach are investigated. First, they are briefly described and then a solution to the anti-pattern is proposed. Because all investigated anti-patterns in this chapter are detectable through the new approach, for each of the anti-patterns a rule for detecting the anti-pattern is implemented. Code snippets for the rule are attached in Appendix A.

5.1. Approach

Figure 5.1 visualizes, how this part of diagnoseIT analysis works. The idea of the detection based on time series approach is to save particular trace metrics from monitoring tools into a TSDB (see Section 2.6). One database table contains required anti-pattern specific information, that is needed for the detection. For example, when an anti-pattern is detected through response times and CPU times, these measures will be saved from incoming traces into one table. The primary key of the table is the timestamp. The chronological order of the measured values describes an observation period of the monitored system. When the detection based on time series analysis is periodically triggered, data is fetched from the database and rules are applied to the data. By analyzing the

data, the rule engine obtains insights concerning the state of the monitored system at a particular time and anti-patterns can be diagnosed. For example, if from time to time the response time of the monitored system changed, the diagnoseIT can notice that by applying rules.

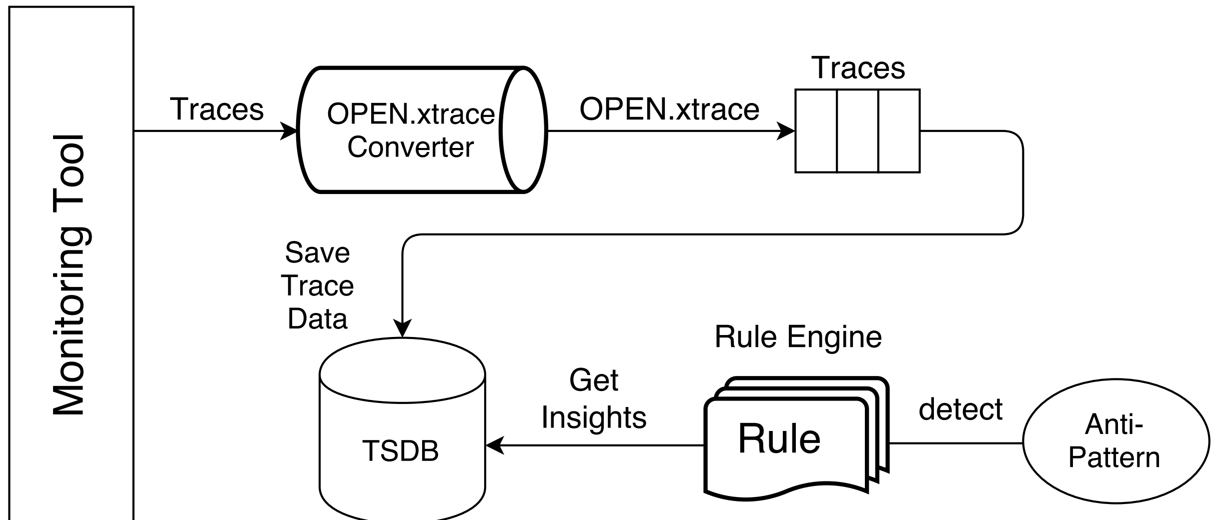

Figure 5.1.: Detection based on time series approach

5.2. Considered Anti-Patterns

In this section, rules for detecting the Ramp, Traffic Jam and More is Less anti-patterns are described. To detect them, the analysis of a series of traces is required. For the concrete implementation, InfluxDB [inf16a] (see Section 2.6) is used for the TSDB and is integrated to the diagnoseIT. For the implementations of the anti-pattern detection some assumptions are necessary:

1. One database table holds performance measurements from one particular sequence of actions. An example for a sequence is to search for a DVD within the DVD Store and then to add the DVD to shopping cart. This excludes the possibility that high values, e.g. high response times, are not caused by performance problems, but by other sequences.

2. It is assumed, that the data from the traces is already in the database table. So when the detection based on time series analysis is triggered, there is already data that can be analyzed.

Methods for fetching particular data from the TSDB are implemented, but are not further explained. Fetching of data from the InfluxDB is shown in its documentation [inf16b].

For each of the following presented anti-patterns we provide a description, propose a solution, and describe how it can be detected. For all anti-patterns we then propose a design for a rule that we are going to implement. How the rule is concretely implemented is described after the proposed rule design. Concrete Java code for the rules can be found in Appendix A.

The Ramp

Description: The Ramp anti-pattern occurs, when the response times rise over time. Due to increasing response times, the performance of the system decreases. Often cause of the Ramp is an increase of the amount of the data the longer the application executes. Root causes for the Ramp can be other anti-patterns, especially the Sisyphus Database Retrieval [DGS02] and the Dormant References anti-pattern. In case of the Sisyphus Database Retrieval, the increasing amount of the data are due to growing database tables over time and, in case of the Dormant References, growing data structures [Wer15].

Solution: Solutions to the Ramp depend on concrete root causes. In case the Ramp is actually caused by the Sisyphus Database Retrieval anti-pattern, a proposed solution is to formulate proper SQL queries [Wer15]. A solution for the Dormant References anti-pattern is provided in Section 3.2. Smith and Williams [SW02a] propose to use algorithms or data structures based on maximum size, when the amount of data increases over time. Algorithms that adapt to the increasing amount of data also can be helpful.

Detection: To detect this SPA, only timestamps and corresponsing response times are needed. These values are stored into the TSDB.

Rule design: Using the linear regression [ZSNS09], we put a regression line through the data points of the TSDB. Figure 5.2 shows an example of a regression line through data points with positive slope. For the rule implementation, on the x axis are the timestamps and on the y axis are the data points' response times. When the slope of the regression line is positive, then there is a rise of response times within the time series data. To detect the Ramp anti-pattern, the slope has to be above a setted threshold.

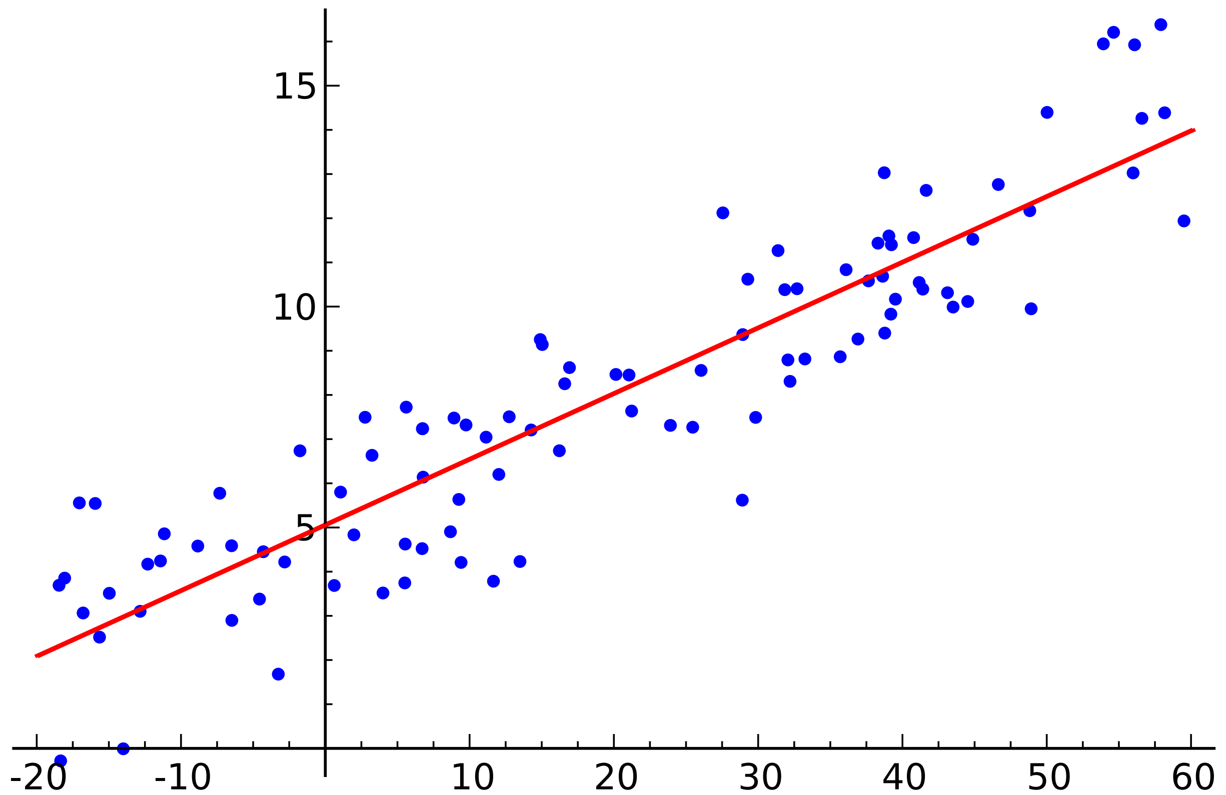

Figure 5.2.: Regression line through data points [Wik10]

Implementation: The concrete Java code for this rule is provided in Listing A.5. When the detection based on time series analysis is triggered, the rule for detecting a Ramp fetches in line 17 the data from the TSDB and puts them into a list. The analysis of the data is performed using in line 20 the `SimpleRegression` class from the Apache Commons Mathematics Library [The16]. `SimpleRegression` provides statistical methods for regression lines. The regression line is calculated with the ordinary least squares [CI11] algorithm. The formula that describes the regression line is in Equation (5.1):

(5.1) $y = a + b * x$

where y is the dependent variable, in our case the response time, a is the intercept of the regression line, b is the slope of the regression line and x is the independent variable, in our case the corresponding timestamp to y.
The `addData(double x, double y)` method of `SimpleRegression` adds data points to the `SimpleRegression`, so that the regression line can be put through data points. By iterating in line 23 through the list of data points for each object, the timestamp is passed as the `x` parameter and the response time is passed as the `y` parameter to the `addData(double x, double y)` method. The call of `getSlope()` on the

`SimpleRegression` instance returns the slope of the regression line. When the slope is above a predefined threshold, then the Ramp is detected. The default value for the slope threshold is set to 0.05.

Traffic Jam

Description: In case of the Traffic Jam anti-pattern, a high variance of response times can be observed in the monitored system. This is often due to overload situations, where some requests cannot access resources and therefore have to wait. Other requests can process and so the variance of response times is observed. Other performance anti-patterns can also be a cause for the Traffic Jam, like the One Lange Bridge anti-pattern, where a lock for resources only lets one or few processes to access a resource. Other causes are also a CPU intensive application or a congestion on the database.

Solution: Solutions to the Traffic Jam anti-pattern within a software depend on concrete root causes. A solution to the One Lane Bridge anti-pattern is provided in Section 3.2. Smith and Williams [SW02a] propose to provide enough processing power, so that some requests do not get stuck by accessing resources.

Detection: To detect this SPA, only response times are needed. These values are stored into the TSDB.

Rule design: Using the linear regression, we put a regression line through the data points of the TSDB. On the x axis are the timestamps and on the y axis are the data points' response times. To distinguish from the Ramp anti-pattern detection, the slope of the regression line has to be below or equal a predefined threshold. If so, we calculate the variance and the standard deviation [KK54] of all response times. The higher the value of the standard deviation, the more the response times vary from each other. To get a variance value, that is independent from the mean of response times, we calculate the coefficient of variation [Eve98], where we divide the standard deviation by the mean of response times. If the value of the coefficient of variation exceeds a threshold, the Traffic Jam anti-pattern is detected.

Implementation: The concrete Java code for this rule is attached in Listing A.6. The rule for detecting a Traffic Jam fetches in line 20 the data from the TSDB and puts them into a list. The analysis of the data is performed using `SimpleRegression` class from the Appache Commons Mathematics Library. `SimpleRegression` provides statistical methods for regression lines. The `addData(double x, double y)` method of `SimpleRegression` adds data points to the `SimpleRegression`, so that the regression line can be put through data points. The regression line is calculated with the least squares algorithm. By iterating through the list of data points, for each object the timestamp is passed as the `x` parameter and the response time is passed as the `y` parameter to the `addData(double x, double y)` method. The call of `getSlope()` on the `SimpleRegression` instance returns the slope of the regression line. To differentiate from the Ramp anti-pattern detection, the slope of the regression line has to be equal or below a predefined threshold. The default value for the slope threshold is 0.05.

When the slope is below this threshold, then the response times from the data points are analyzed. First of all, we calculate in line 30 of the rule the sum of response times. In line 33, we divide the sum by the amount of data points to calculate the mean of response times. The result is used for calculating the variance of the response times. The equation for the variance V^2 that is used for the implementation is in Equation (5.2):

$$V^2 = \frac{(x_1 - \overline{x})^2 + (x_2 - \overline{x})^2 + ... + (x_n - \overline{x})^2}{n-1} \tag{5.2}$$

where $\overline{x}$ is the mean of all response times, $x_1 ... x_n$ are the response times and n is the amount of response times.

Then, in line 43 the standard deviation of the response times is calculated. The equation for the standard deviation S, that is used for implementation, is in Equation (5.3):

$$S = \sqrt{V^2} \tag{5.3}$$

where V^2 is the variance of response times that was calculated before.

Because a higher mean of response times directly leads to a higher standard deviation, in the next step the coefficient of variation is calculated, so that the standard deviation is independent of the mean of response times. The formula to calculate the coefficient of variation c_v that is used for the implementation is in Equation (5.4):

$$c_v = \frac{S}{\overline{x}} \tag{5.4}$$

where S is the standard deviation of response times and $\overline{x}$ is the mean of response times. The higher the value of the coefficient of variation, the higher the variance of response times. When the value of the coefficient of variation exceeds in line 48 the predefined threshold, the Traffic Jam anti-pattern is detected. The default value for the threshold is 0.3.

More is Less

Description: The More is Less anti-pattern occurs, when a system tries to accomplish more work than resources allow. When too many processes run within a system, its operation system is permanently busy with communicating with hard disk, because of, e.g., paging, so that it accomplishes no useful work. This process is also called "thrashing" and can lead to high response times, because the hard disk is in general slower than the systems' main memory.

Solution: It has to be assessed, with the help of thresholds, to a which degree thrashing is acceptable [Tru11]. If the system architecture cannot satisfy its performance goals, while not exceeding the thresholds, the system architecture has to be reconsidered.

Detection: Overload situations, where the system spends more time doing other things than accomplishing real work, can be detected by the diagnoseIT. CPU times and response times are required from the TSDB.

Rule design: A data point from the TSDB is analyzed, if it has a considerable higher response time than the average. If that is the case, we calculate the ratio of its CPU time to its response time, and compare it with the average ratio of all data points. When the data point's ratio is considerable lower than the average, we assume that the data point represents an overload situation of the system.

Implementation: The concrete Java code for this rule is attached in Listing A.7. The More is Less rule first fetches in line 17 the data from the TSDB and puts them into a list. We calculate the sum of response times and the sum of CPU times. Then, in line 27 and line 28, the mean of response times and the mean of CPU times is calculated by dividing the sum by the amount of data points.

Afterwards, the rule iterates through the list of data points, and compares in line 34 the response time of the current data point with the average response time. When the

response time multiplied by a predefined percent is higher than the average, the rule investigates the data point further. The default value for the percent is set to 0.4. The rule divides the current data point's CPU time by its response time to get its CPU time to response time ratio. We apply the same process to calculate the average ratio of all data points. The rule checks in line 40, whether the current data point's ratio is lower than the average ratio multiplied by another predefined percent (default value is 0.5). If this is true, the rule alerts and an overload situation is detected.

Chapter 6

Detection using combined Approach

In this chapter, we combine the detection based on trace data analysis and the detection based on time series analysis. This is the third analysis part of diagnoseIT and we call it from now on detection using combined approach. The third analysis approach links a data point from the TSDB to a corresponding trace, which then can be analyzed further using the detection based on trace data analysis.

In Section 6.1, more details on the detection using combined approach are provided. It is explained, how a data point from the TSDB can be linked to a corresponding trace.

Two anti-patterns, that can be considered for the detection using combined approach, are shown in Section 6.2. Actually, these anti-patterns also can be considered for the second approach, the detection using time series, since their detection needs a longer observation period of trace data.

6.1. Approach

As mentioned in the introduction, both approaches, the detection based on trace data and detection based on time series approaches, can be combined. The combination gets accessible through a unique trace identifier, the TraceID (Section 2.5). Beside the timestamp and particular OPEN.xtrace metrics, the TSDB stores also the TraceID into the tables. When detecting inconsistencies in the time series data, e.g., when at some point an undesirable high response time is detected, the affected data can be connected to traces. The traces then can be analyzed further using the detection based on trace data analysis.

The rules implemented in this chapter analyze the time series data and try to detect anti-patterns from it. When the rule detects inconsistencies within the data, the problematic

data points are saved into a list with their corresponding TraceID. Another rule from the detection based on trace data analysis can access this list. Traces have to be also stored, so that they can be retrieved. The retrieval of traces is not covered by this thesis. We only detect inconsistencies in the TSDB and do not analyze them further.

Figure 6.1 shows an overview of the detection using combined approach.

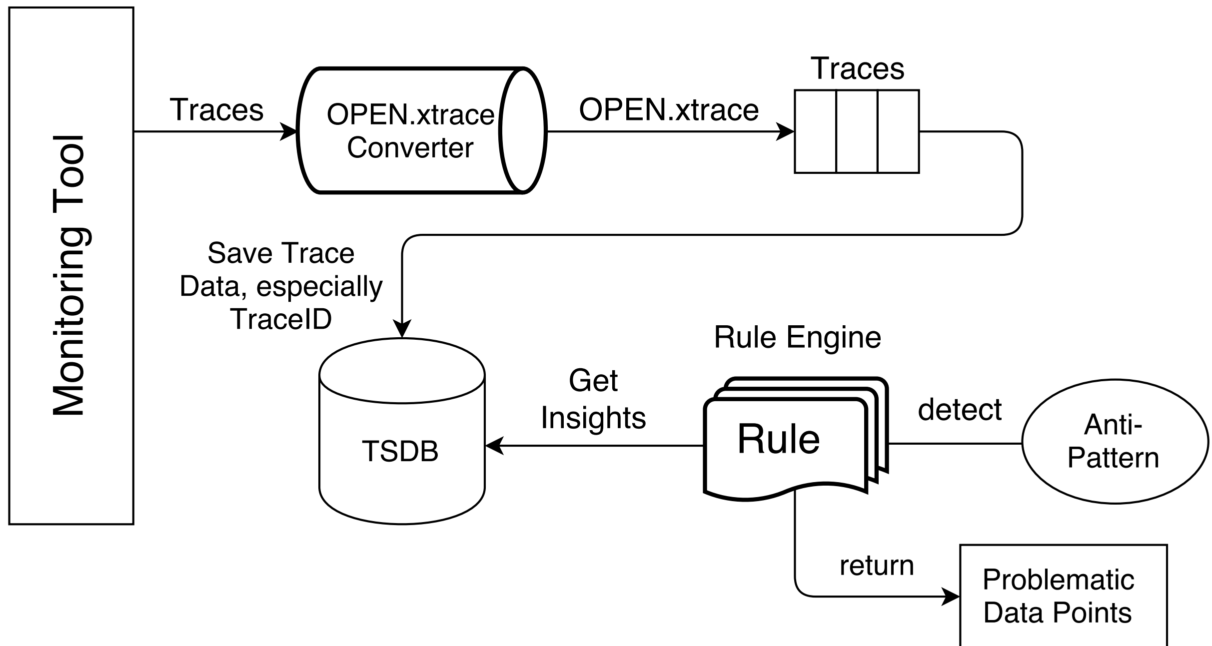

Figure 6.1.: Detection using combined approach

6.2. Considered Anti-Patterns

In this section, we implement rules for detecting the Application Hiccups and Garbage Collection Hiccups anti-patterns. As they are also considered as anti-patterns, that need historical data, the assumptions from Section 5.2 also apply here. Beside the required performance metrics from the OPEN.xtrace, the TraceID is also saved into the InfluxDB, to link a problematic data point with a trace. The trace then can be analyzed further in the detection based on trace data analysis. Further analysis of problematic traces is not covered by this thesis. The following rules collect problematic data points and return them with their TraceID, so that another rule from the detection based on trace data analysis can fetch these data points and link them to traces with corresponding TraceIDs.

For both of the following presented anti-patterns we provide a description, propose a solution, and describe how it can be detected. For both anti-patterns we then propose a design for a rule that we are going to implement. It is also explained how inconsistent data points will be detected. How the rule is concretely implemented is described after the proposed rule design. The Concrete Java codes of the rules can be found in Appendix A.

Application Hiccups

Description: The Application Hiccups anti-pattern is a SPA, where periodic increase in response times of the monitored software can be observed. When the increase of the response times settles down, the system performance becomes acceptable again. The causes for Application Hiccups are often periodic tasks like GC activities, which temporary block other threads from further executing. Other root causes are synchronization points or database locking, and also other anti-patterns, like the Dispensable Synchronization anti-pattern.

Solution: Solutions for Application Hiccups within an applications software depend on concrete root causes. If Application Hiccups are actually caused by a Dispensable Synchronization, a solution is provided in Section 3.2. When the cause are GC activities, then actually the Application Hiccups are caused by Garbage Collection Hiccups, which are described in Section 6.2.

Detection: To detect this SPA, only response times are needed. These values are stored into the TSDB.

Rule design: We use a box plot [MTL78] for the detection of outliers within the response times of the TSDB. Figure 6.2 shows an example diagram of a box plot. A box plot describes a box with two whiskers. Inside the box the average 50 percent of the values is located. The lower threshold of the box is called lower quartile and the upper threshold of the box is called upper quartile. Typically the lower quartile is a percentile set to 25 percent and upper quartile is a percentile set to 75 percent. The range between those quartiles is called interquartile range, where the average fifty percent of the data is located. For this thesis, inside the box the average 50 percent of all time series data points' response times is included. The whiskers describe a range for data that is outside the box, but is still not considered as an outlier. Data, that exceeds the threshold of the whiskers, is considered as an outlier. We only need the upper whisker to detect

an Application Hiccup. The range of the upper whisker is defined as a line from the upper quartile till the largest value of all data points' response times, that is within 1.5 interquartile ranges from the upper quartile [MTL78]. This defines the range for response times which are not inside the box, but still are not considered as outliers. When one of the data points' response times exceeds the threshold of the upper whisker, then the data point is an outlier. That means that its response time is much higher than the other data points' response times and represents a inconsistency within the data. The rule saves outliers, that are also considered as hiccups, into a list. The size of the list tells how many Application Hiccups could be detected in the time series data, and is provided by the rule. Since it is meaningful to investigate a trace that causes an outlier further, the rule returns the outliers with corresponding TraceIDs, so that they can be requested from another rule in the detection based on trace data analysis.

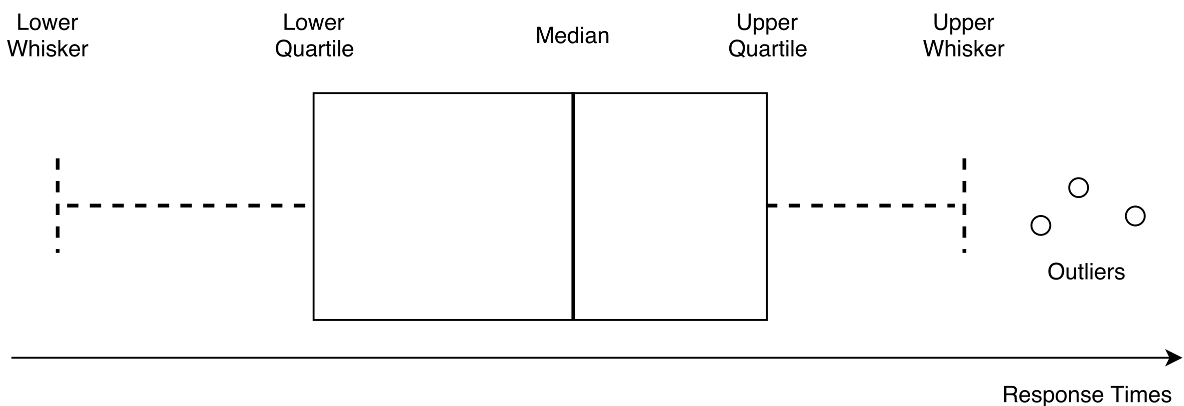

Figure 6.2.: Box plot

Implementation: The concrete Java code for this rule is provided in Listing A.8. The Application Hiccups rule is always fired, when the detection based on time series analysis is triggered. The rule fetches in line 13 the data from the TSDB and puts them into a list. GC times are included in the data. The GC times are not needed to detect Application Hiccups, but are needed later for the Garbage Collection Hiccups rule, that is executed after this rule.

For calculating quartiles we use the Apache Commons Mathematics Library [The16] that allows us to calculate percentiles. Therefore, we instantiate an object from the class `Percentile`, on which we can call the `evaluate(double[] values, double p)` method. The method takes an array `values` and for the second parameter a percent `p` and returns an estimate of the `p`th percentile of the `values`. For the `values` parameter we pass the data points' response times to the method, but the response times have to be first put into an array of type `double`.

After iterating in line 20 through the list of data points and putting their response times into an array, we calculate first the lower quartile, by instantiating in line 23 a `Percentile` object and call on it the `evaluate(double[] values, double p)` method, passing the response times array as first parameter and for the second parameter we pass 25 percent. The upper quartile is calculated almost in the same way, but we pass 75 percent for the `p` parameter.

We calculate in line 29 the interquartile range by subtracting the lower quartile value from the upper quartile value. Since we only want to find outlier response times, we do not have to calculate the exact value of the upper whisker threshold. This would require to find the largest value within the response times, that is within 1.5 interquartile ranges from the upper quartile. So we calculate the threshold in line 31 only by adding 1.5 interquartile ranges to the upper quartile value. The response times, that exceed the threshold, are the outliers. We then iterate in line 34 through the list of data points and compare, if a data point's response time has a higher value than the upper whisker threshold value. If so, the data point that holds the response time is saved into a list. The number of elements in the list describe how many Application Hiccups are manifested in the time series data. The rule returns the list with outliers, which can then be analyzed further.

Alternative rule design: When the detection based on time series analysis is triggered, the diagnoseIT rule engine fetches the data from the TSDB and puts them into a list. By iterating through the list the rule investigates how often the response times are crossing a threshold. Thereby, two crossings of the threshold describe one hiccup.

Figure 6.3 shows one hiccup and the crossing threshold, which is the red line. The rule saves response times, that are above the crossing threshold, with their corresponding TraceIDs into a list. The rule returns the list.

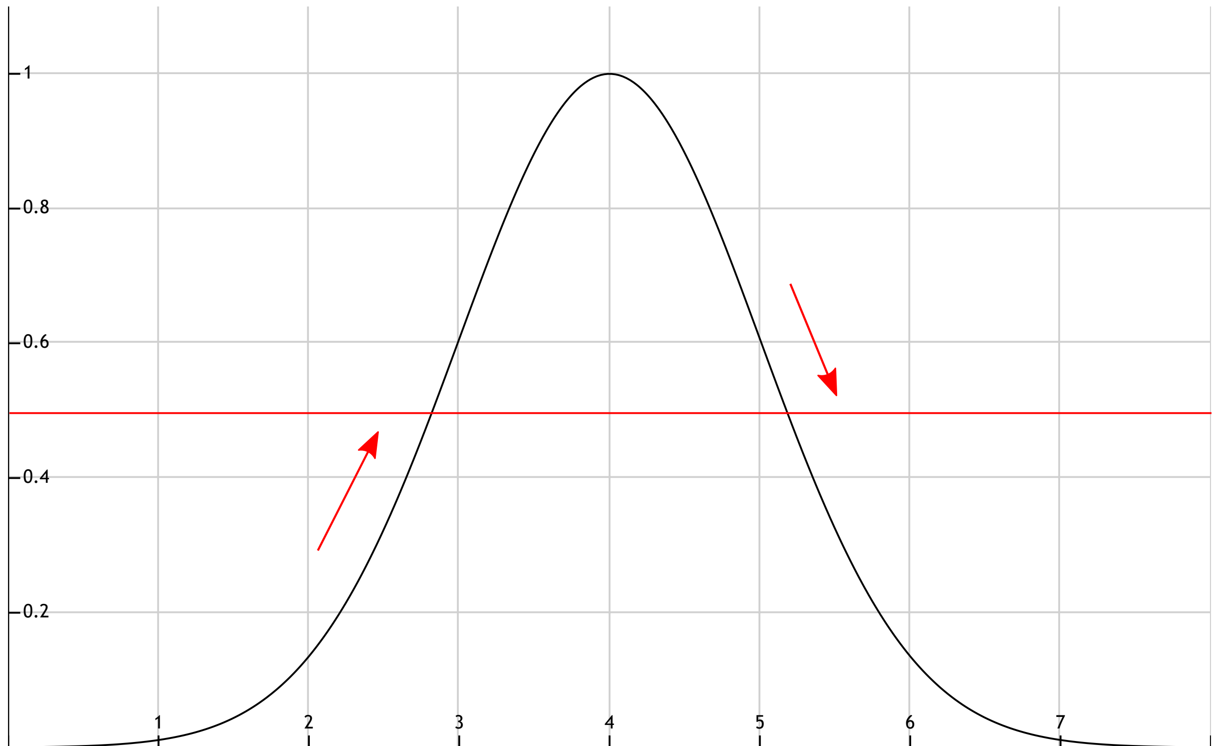

Figure 6.3.: Application Hiccup

Alternative implementation: The concrete Java code for this rule is provided in Listing A.9. The rule for detecting Application Hiccups fetches in line 16 the data from the TSDB. The rule detects one hiccup, when a data point's response time first exceeds a predefined crossing threshold value (default value is set to 50000 ms) and then it is followed later by another data point's response time that is below the crossing threshold. This describes the temporary increase and decrease of response times.

The rule initializes in line 22 a variable `numberOfCrossings` with zero to count how often the predefined crossing threshold is crossed by response times. A boolean variable `isLastTimeAboveThreshold` is also initialized. It is initialized in line 25 with true, when the response time from the first data point is above the crossing threshold, otherwise with false. The rule iterates in line 28 through the data points and for the current data point's response time it checks in line 36, if either the `isLastTimeAboveThreshold` is true and the current data point's response time is below the crossing threshold, or if the `isLastTimeAboveThreshold` is false and the current data point's response time is above the crossing threshold. If one of the conditions is true, the `numberOfCrossings` variable is increased by one and in line 40 the `isLastTimeAboveThreshold` is negated. For example, when the current data point's response time is above the crossing threshold, for a crossing of the threshold the previous data point's response time had to be below

the threshold, and therefore the `isLastTimeAboveThreshold` must be false. Because the current data point's response time is now above the crossing threshold, the value from `isLastTimeAboveThreshold` must be set to true.

After counting all crossings, they are in line 44 divided by two. This provides the actual amount of hiccups. The amount of hiccups is provided by the rule. The rule returns a list with data points, whose response times are above the crossing threshold, since they represent inconsistent high values, that should be analyzed further.

Garbage Collection Hiccups

Description: The Garbage Collection Hiccups anti-pattern is a cause for Application Hiccups and occurs when the periodic increase of response times in a monitored software is caused by GC activities. When the GC periodically executes, the virtual machine (VM) gets temporary blocked. During GC activities the VM cannot satisfy other requests. When the GC tasks are finished and the VM is free again, the VM starts now to satisfy a backlog of requests leading to increased response times.

Solution: Concrete solutions to the Garbage Collection Hiccups depend on concrete root causes, since it is often caused by other anti-patterns. Some examples are Large Temporary Objects, Sisyphus Database Retrieval, Session as a Cache [Kop11] and Wrong Cache Strategy [Gra10] anti-patterns. In case the hiccups are caused by Large Temporary Objects, a solution is provided in Section 3.2. A solution to the Sisyphus Database Retrieval is provided in the Ramp solution (Section 5.2).

The misusage of a HTTP session as a cache to store data leads to the Session as a Cache anti-pattern. The solution is to use a cache instead of a HTTP session [Wer15].

The Wrong Cache Strategy anti-pattern is caused by a strategy that caches too many objects, which can lead to increased GC activities. The solution is to investigate first which objects to cache and then to implement a appropriate caching strategy [Gra10].

Detection: Garbage Collection Hiccups can be detected in the same way as detecting the Application Hiccups anti-pattern, since it is a special case of the Application Hiccups. Both rule designs and implementations from the detection of Application Hiccups can be also considered for the Garbage Collection Hiccups detection. Instead of analyzing response times, the GC times have to be analyzed. Therefore, the only required data from the OPEN.xtrace are GC times. Because the implementation would be the same as for the Application Hiccups anti-pattern detection, for this thesis, we choose another approach. We investigate Application Hiccups further to check, whether they are caused

by the GC. The rule design and implementation of this rule corresponds to the general rule design of the Application Hiccups rule, not to its alternative.

Rule design: The rule for detecting Garbage Collection Hiccups requests the problematic data points that were returned by the Application Hiccups rule. The data points hold, beside TraceID and response times, also the corresponding GC times. So the rule checks, whether the GC time of an outlier is high. If yes, the outlier was likely caused by the GC, and therefore it can be assumed that the Application Hiccup is actually a Garbage Collection Hiccup.

Implementation: The concrete Java code for this rule is provided in Listing A.10. The Garbage Collection Hiccups rule is fired after the Application Hiccups Rule. The Application Hiccups rule returned a list of outliers, that are requested in line 11 by the Garbage Collection Hiccups rule (see Figure 6.4). The rule first checks in line 18, whether the amount of outliers is higher than zero. If not, the rule immediately stops executing, because when there are no Application Hiccups there also cannot be Garbage Collection Hiccups.

The rule iterates in line 27 through the Application Hiccups outliers, and compares the response time with the GC time of an outlier. If the GC time is in line 34 higher than the certain percentage of the response time (default value is 50 %), then we assume that the hiccup is caused by the GC. The corresponding data point is then saved into a list. The list is returned later by the rule. The rule tells the user how many hiccups are caused by the GC by returning the size of the GC times outlier list.

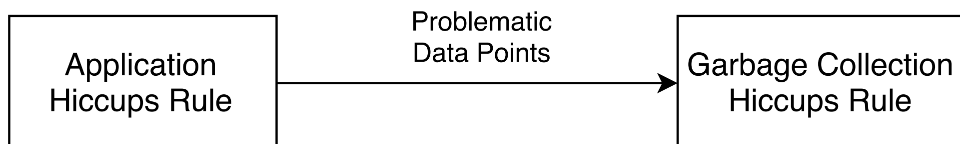

Figure 6.4.: Execution of the Garbage Collection Hiccups rule

Chapter 7

Evaluation

In this chapter, we evaluate the rules we implemented in the previous chapters. We use different approaches for the evaluation of the detection based on trace data rules and the evaluation of the detection based on time series rules. Since the considered rules for the detection using combined approach are actually rules that analyze time series data, we consider them also for the second evaluation approach.

For both evaluation approaches we introduce first evaluation goals. Afterwards, we describe the experiment setup. The experiment is then conducted in several test cases. For each test case, the results are documented. Besides, test cases include a result discussion, where we analyze the results.

In Section 7.1, we evaluate the rules that analyze a single trace to detect anti-patterns.

In Section 7.2, we evaluate the rules that analyze a series of trace data to detect anti-patterns.

We investigate if our evaluation is valid. We consider internal threats, external threats, construct threats and conclusion threats in Section 7.3.

7.1. Evaluation of Rules based on Single Traces

In this section, we evaluate the implemented rules from Section 4.3. The rules try to detect anti-patterns by analyzing a single trace. The anti-patterns we want to detect are the N+1 Query Problem, the Stifle, the Expensive Computation and the Phantom Logging anti-pattern.

Evaluation Goals

By evaluating the rules, we especially want to investigate if the rules correctly detect SPAs. That means, when in a trace an anti-pattern is manifested, then a rule, that describes the characteristics of that anti-pattern, should detect it. Hence, the first research question is

RQ1: Do the implemented rules correctly detect anti-patterns?

When the rule wrongly detects not an anti-pattern, it would deliver a false negative result. Vice versa, when there is no anti-pattern in a trace manifested, the rule should not detect a SPA and should not thereby deliver a false positive result. False negative results are more critical, since they suggest false safety. However, we want to uncover both, false positive and false negative results from the rules.

The second research question is

RQ2: How can we improve the implemented rules?

We will answer both research questions based on the evaluation results. For RQ2, we will try to identify shortcomings of the rules and how they can be improved by optimizing the analysis algorithm.

Experiment Setup

We investigate the N+1 Query Problem rule and the Stifle rule as follows. We apply the rules on the trace data, which we received through the monitoring of a real application. We use the NovaTec DVD Store (Section 2.1) application and monitor it with the APM tool CA Introscope [CA16]. The DVD Store allows us to enable different performance problems. In this case, we inject the N+1 Query Problem into the application. We export execution traces from CA Introscope and convert them into the OPEN.xtrace with an adapter. How traces from the APM tool can be exported and how they can be converted into the OPEN.xtrace with the help of adapters, is shown in [OHH+16]. To show the limitations of the N+1 Query Problem rule, we generate synthetic traces with a class `TraceCreator`. The results are shown in the tests 1-3.

The Expensive Computation rule is applied to synthetic traces to investigate whether the rule correctly detects the method within a trace, that executes most on the CPU.

For the Phantom Logging rule, we use a exported dashboard file from the APM tool Dynatrace [Dyn16], that was provided by the NovaTec GmbH. These traces include log

messages from monitored applications. Test 4 presents the results of the evaluation of this rule.

All tests are structured as follows. The test begins with a description, where we describe exactly how the rules are tested. Afterwards we present what results we expect. Then, we present the results of the test and analyze them.

Test 1 - Detection of the N+1 Query Problem

Description: With the loadIT tool [Nov] we generate load for the DVD Store application. We simulate 300 users simultaneously access the store. In the application itself, we enable the "Slow Search" function, where the N+1 Query Problem is injected to the application. The DVD Store is monitored using CA Introscope. We pick 20 problematic traces (with high response times) from the tool, where the N+1 Query Problem is manifested. Then, we convert the traces into the OPEN.xtrace and apply the N+1 Query Problem rule on the traces.

Figure 7.1 shows an excerpt of a trace from CA Introscope with manifested N+1 Query Problem, that is converted into the OPEN.xtrace.

```
----------------- SubTrace (108783130626980) : [ Response Time (ms)| Execution Time (ms)|
CPU Time (ms)] ----------------
# HTTP Optional[POST] (/dvdstore/browse.seam) [ 19626 | 0 ]
   # HTTP Optional[POST] (/dvdstore/browse.seam) [ 19626 | 0 ]
      Optional[LifecycleImpl].Optional[execute] [ 19480 | 35 ]
      Optional[ActionListenerImpl].Optional[processAction] [ 17224 | 0 ]
            Optional[FullTextSearchAction].Optional[doSearch] [ 17224 | 6259 ]
               Optional[PerformanceSettingsBean].Optional[getSlowLongSearch] [ 0 | 0 ]
               Optional[WrappedConnection].Optional[prepareStatement] [ 1398 | 0 ]
                  Optional[JdbcConnection].Optional[prepareStatement] [ 1398 | 1398 ]
               Optional[WrappedPreparedStatementJDK6].Optional[executeQuery] [ 1252 | 0 ]
                  # SQL (SELECT PRODUCT0_.PRO) [ 1252 | 1252 ]
               Optional[WrappedResultSet].Optional[close] [ 0 | 0 ]
                  Optional[JdbcResultSet].Optional[close] [ 0 | 0 ]
               Optional[WrappedConnection].Optional[close] [ 0 | 0 ]
               Optional[WrappedConnection].Optional[prepareStatement] [ 0 | 0 ]
                  Optional[JdbcConnection].Optional[prepareStatement] [ 0 | 0 ]
               Optional[WrappedPreparedStatementJDK6].Optional[executeQuery] [ 3 | 0 ]
                  # SQL (SELECT INVENTORY0_.I) [ 3 | 3 ]
               Optional[WrappedResultSet].Optional[close] [ 0 | 0 ]
                  Optional[JdbcResultSet].Optional[close] [ 0 | 0 ]
               Optional[WrappedConnection].Optional[close] [ 0 | 0 ]
               Optional[WrappedConnection].Optional[prepareStatement] [ 0 | 0 ]
                  Optional[JdbcConnection].Optional[prepareStatement] [ 0 | 0 ]
               Optional[WrappedPreparedStatementJDK6].Optional[executeQuery] [ 2 | 0 ]
                  # SQL (SELECT INVENTORY0_.I) [ 2 | 2 ]
               Optional[WrappedResultSet].Optional[close] [ 0 | 0 ]
                  Optional[JdbcResultSet].Optional[close] [ 0 | 0 ]
               Optional[WrappedConnection].Optional[close] [ 0 | 0 ]
               Optional[WrappedConnection].Optional[prepareStatement] [ 0 | 0 ]
                  Optional[JdbcConnection].Optional[prepareStatement] [ 0 | 0 ]
                ...
```

Figure 7.1.: Example trace (excerpt) with N+1 Query Problem

We let the rule also execute on traces, where the N+1 Query Problem is not manifested. For this purpose, we turn off the injection of the N+1 Query Problem and let the rule execute on 10 traces.

We build also synthetic traces with a class `TraceCreator`, that we implemented to show the limitations of the rule. We build one trace, where one different query is followed by 11 equal queries, but actually the 11 equal queries have nothing to do with the first query. In one further synthetic trace, we let the N+1 Query Problem appear not at the beginning of the investigated subtree, but after some other queries.

Expectations: The rule detects problematic N+1 Query Problems, that are manifested within traces, in all 20 traces correctly. The traces also contain N+1 queries that are causing no performance problems, but these are excluded by the generic rules.

The rule detects in the 10 traces without N+1 Query Problem correctly not the anti-pattern. Nevertheless, the synthetic traces will show the limitations of the rule. It will detect wrongly the N+1 Query Problem within the trace, where one different query is followed by 11 equal queries, that have nothing to do with the first query. For the second synthetic trace, the rule will not detect the N+1 Query Problem, but actually it is manifested within the synthetic trace.

Results: Once the generic rules provide an aggregated root cause object with type DatabaseInvocation, the N+1 Query Problem rule detects in all 20 traces the N+1 Query Problem anti-pattern from the aggregated object correctly.

The rule detects no N+1 Query Problem within the 10 traces, where the anti-pattern is not manifested.

The rule delivers a false positive result for the first synthetic trace. It detects wrongly the N+1 Query Problem. The rule delivers a false negative result for the second synthetic trace. It detects wrongly not the N+1 Query Problem.

Result discussion: The rule detected the N+1 Query Problem correctly in all traces from the monitored DVD Store application. Therefore, when in a trace the N+1 Query Problem is manifested and the generic rules provide an aggregated root cause object, then the rule will always detect the anti-pattern, except when the 'N+1' queries do not start at the root of the investigated subtree, as we have seen in the results for the synthetic traces.

We have a false positive result for the first synthetic trace, because the rule does not analyze the SQL statements of the `DatabaseInvocations`. As long as one different query

is followed by more than 10 equal queries, the rule detects the N+1 Query Problem, no matter if it is manifested in the trace or not. For a safe detection of the N+1 Query Problem, the SQL statements have to be analyzed and is a recommended work for the future.

We get the false negative result for the second synthetic trace, because the rule can only detect N+1 Query Problems, when the first query starts at the top of the investigated subtree, and the 'N' queries follow right after the first query. When the N+1 Query Problem is nested between other database queries at the same subtree level, the rule does not detect it. The analysis algorithm of the rule should be improved in future work.

Test 2 - Detection of the Stifle

Description: A Stifle anti-pattern is always detected, when the generic rules executed before provide a particular amount of same database queries. In test 1, every time a N+1 Query Problem was detected, the Stifle rule immediately also detected a Stifle anti-pattern within the traces (see Listing A.2, line 18 and 19). The main part of the rule (after line 20) did not execute. Therefore, we modify the code of the Stifle rule (remove line 18, 19 and 20), and let the rule completely execute on the same traces, where a N+1 Query Problem was detected in test 1. The traces contain more than 10 equal database queries. Besides, we execute the rules on the 10 traces, where no N+1 Query Problem is manifested (these contain less than 10 equal database queries).

Expectations: The rule detects correctly the Stifle anti-pattern in each trace containing more than 10 equal database queries. The rule detects no Stifle anti-pattern in traces with less than 10 equal queries.

Results: In all 20 traces with more than 10 equal database queries, the Stifle anti-pattern was detected. In the 10 traces with less than 10 equal queries, the Stifle anti-pattern was not detected.

Result discussion: When the generic rules provide an aggregated root cause object with a high amount of equal queries, the Stifle rule always alerts. It detects that there are many equal queries, which cause a performance problem. What the rule cannot detect is, if the queries are small and fine-grained. It would also consider queries for a Stifle, when they have a larger result set obtained by same SQL statements. Therefore,

the SQL statements have to be analyzed and this is a recommendation for the future work.

Test 3 - Detection of Expensive Computations

Description: We let the rule execute on 10 synthetic traces. The traces are built with the class `TraceCreator`. We create a `Trace` and set a `SubTrace` as its root. We create a `MethodInvocation`, and set it as the root of the `SubTrace`. The `MethodInvocation` serves as a parent for further `MethodInvocations`. For each `MethodInvocation`, we set a corresponding CPU time.

We cannot use real traces for evaluation, since we have no adapter to convert traces from APM tools, that measure CPU times, into the OPEN.xtrace.

Expectations: The rule always detects the method within the trace, that has the highest exclusive CPU time. It correctly detects an Expensive Computation, where the ratio of its exclusive CPU time to the trace response time exceeds a predefined threshold (default value is 0.10).

Results: For all synthetic traces, the rule correctly detected all methods with the highest exclusive CPU time. The rule thereby detects correctly Expensive Computations in the context of this thesis (exclusive CPU time to trace response time ratio has to be higher than 0.1, so that the method is considered as an Expensive Computation).

Result discussion: The rule worked as expected. A trace with an Expensive Computation should be investigated further to find the cause of the high computation. This is a recommended work for the future, as the cause can be other SPAs. Furthermore, it is not sophisticated enough to consider a method as an Expensive Computation, when the ratio of its exclusive CPU time to the trace response time exceeds a predefined threshold. The rule surely can be improved in this regard.

Test 4 - Detection of Phantom Logs

Description: For the Phantom Logging rule, we use a XML [Wor16] file with exported dashboard from Dynatrace, that is provided by the NovaTec GmbH. Dynatrace captures logging events from the monitored application. The file contains traces with captured logging events. We modify for this test real traces. We search for locations in the file,

where we have log events, and manually set the level of a log message lower than its current level. In our example in Figure 7.2, we changed the log level from the log message in the middle from ERROR to DEBUG. Then, we convert the trace into OPEN.xtrace with a Dynatrace adapter and execute the rule on the trace.

```
<node method="log4j [ERROR]"
class="dummy.url.jif.presentation.ivas.DialogSteuerung"
  <attachment type="LoggingNodeAttachment">
    <property key="logger"
    value="dummy.url.jif.presentation.ivas.DialogSteuerung"/>
    <property key="severity" value="40000"/>
  </attachment>
</node>
<node method="log4j [DEBUG]"
class="dummy.url.jif.presentation.ivas.DialogSteuerung"
  <attachment type="LoggingNodeAttachment">
    <property key="logger"
    value="dummy.url.jif.presentation.ivas.DialogSteuerung"/>
    <property key="severity" value="40000"/>
  </attachment>
</node>
<node method="log4j [ERROR]"
class="dummy.url.jif.common.context.TechnicalContext"
  <attachment type="LoggingNodeAttachment">
    <property key="logger" value="dummy.url.jif.common.context.TechnicalContext"/>
    <property key="severity" value="40000"/>
  </attachment>
</node>
```

Figure 7.2.: Log events in the XML file

Expectations: When the rule detects a log message with a log level lower than the predefined threshold, the rule always correctly detects a Phantom Log.

Results: The rule works as expected. When changing nothing in the file, no Phantom Log is detected. As soon we set the level of a log message lower, the Phantom Logging rule correctly detects the anti-pattern.

Result discussion: The rule always detects log messages, that are lower than the predefined threshold. If the rule detects a Phantom Log, the user knows that he logged a message he actually did not want to log.

7.2. Evaluation of Rules based on Time Series

This section is for evaluating rules, that analyze time series data. The assumption from Section 5.2 is, that the TSDB is already filled with trace data from traces. So when the detection based on time series analysis is periodically triggered, there is always trace data to analyze. Therefore, the idea for the evaluation of the rules is to fill the InfluxDB first with synthetic trace data and then to execute the rules.

After testing the rules on synthetic trace data, we use the NovaTec DVD Store as a real application, that we monitor with the APM tool inspectIT [NTC16a]. inspectIT allows us to export more than one trace at once. Trace data from the exported traces is then saved into the TSDB. How traces from inspectIT can be exported is described in [OHH+16].

We check for both, synthetic trace data and real trace data, what the output of the rules is.

Evaluation Goals

By evaluating the rules, we especially want to investigate if the rules correctly detect SPAs. That means, when in the time series data an anti-pattern is manifested, then a rule, that describes the characteristics of that anti-pattern, should detect it. Hence, the first research question is

RQ1: Do the implemented rules correctly detect anti-patterns?

When the rule wrongly detects not an anti-pattern, it would deliver a false negative result. Vice versa, when there is no anti-pattern in the time series data manifested, the rule should not detect a SPA and should not thereby deliver a false positive result. False negative results are more critical, since they suggest false safety. We want to uncover both, false positive and false negative results from the time series rules.

The second research question is

RQ2: How can we improve the implemented rules?

We will answer both questions based on the evaluation results. For RQ2, we will try to identify shortcomings of the rules and how they can be improved by optimizing the analysis algorithm.

Experiment Setup

For the Ramp rule, Traffic Jam rule and the Application Hiccups rule, we first fill the InfluxDB for the test cases 1-6 with different synthetic trace data. Then, for the test cases 7-11, we fill the InfluxDB with the real trace data from a monitored application. According to the principle of equivalence partitioning [Bur06], we cover with all test cases that a particular anti-pattern is manifested at least once in the time series data and at least once not.

For the test cases 1-6, we test the rules on 100 synthetic data points. The 100 data points are saved into the TSDB with a time distance of 1 second between each point. The only rule that is affected by the time distance is the Ramp rule, because the slope of the regression line is dependent on the length of an observation interval. The threshold for the slope then can be adapted to correspond to the timestamps within the time series data.

For the real trace data, we use the NovaTec DVD Store application, and monitor it with the inspectIT. The DVD Store lets us enable different peformance problems and we can export more than one trace at once from inspectIT. The trace data is then saved into the TSDB.

For the Garbage Collection Hiccups rule, we manifest with synthetic trace data Application Hiccups into the time series data and also save synthetic corresponding GC times to the response times into the database. Thereby, we test, if the rule detects that the Application Hiccups are actually caused by the GC. The inclusion of corresponding GC times is in test case 6.

For the More is Less rule, we fill the TSDB again with the real trace data from the NovaTec DVD Store and try to detect overload situations within the data. We save from traces with high response times their response times and corresponding CPU times into the database. The inclusion of CPU times to the InfluxDB is performed in the test cases 10 and 11.

The test cases are structured as follows. A test case begins with a description, where we describe the detailed setup for the execution of the rules. Afterwards we present what results we expect. Then, we present the results of the test case and analyze them.

Grafana [ÖR15] is used for visualization of data. On the x axis are timestamps, and on the y axis are measurements in ms.

Test Case 1 - Vertical Line through Data Points

Description: We put a perfect horizontal line through the time series data. That means all data points have the same response time. For the response time we set 5000 ms. Figure 7.3 shows the InfluxDB for the 100 data points.

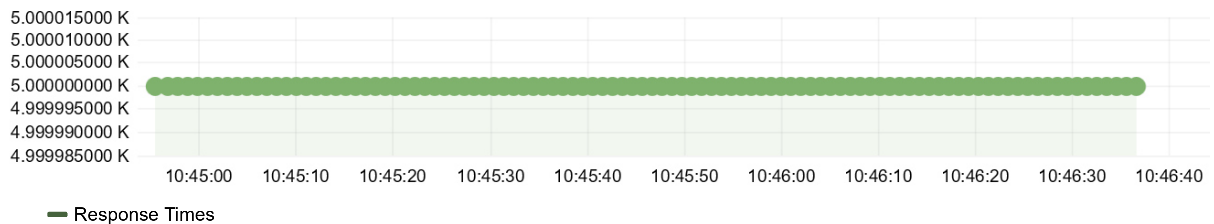

Figure 7.3.: Vertical line through data points

Expectations: Since the response times do not change over time, none of the anti-patterns should be detected.

Results: None of the rules detected anti-patterns. We have no slope for the regression line and the value of the coefficient of variation is 0. Both Application Hiccups rules detected 0 hiccups.

Slope	Coeff. of Var.	App. Hiccups	(ALT) App. Hiccups	SPA
0.0	0.0	0	0	-

Table 7.1.: Test Case 1 - Results

Result discussion: The rules worked as expected.

Test Case 2 - Perfect Ramp

Description: We fill the InfluxDB with response times. The response time for each data point is higher than the prior data point's response time. The data points' response times are all positioned on one straight line. For the response times we start at 3000 ms. The next data point then has the prior data point's response time increased by 100 ms. Figure 7.4 shows the InfluxDB for the 100 data points.

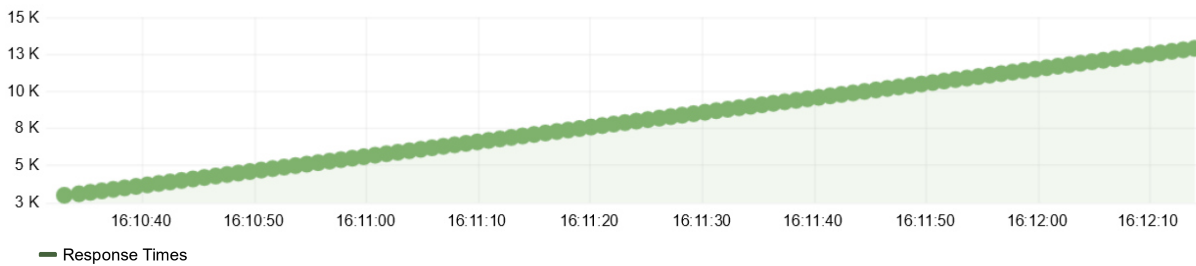

Figure 7.4.: Perfect Ramp of data points' response times

Expectations: The Ramp rule should detect the Ramp anti-pattern. There should be no Application Hiccups, since there is no outlier data point. The Traffic Jam rule should not execute, since the slope is to high, when the Ramp anti-pattern is detected.

Results: The slope of the regression line is ~ 0.1. Since the default value for the slope threshold is set to 0.05, the rule detects the Ramp anti-pattern.

Slope	Coeff. of Var.	App. Hiccups	(ALT) App. Hiccups	SPA
~ 0.1	-	0	0	The Ramp

Table 7.2.: Test Case 2 - Results

Result discussion: The results match our expectations. The rules worked correctly.

Test Case 3 - Rise of Response Times

Description: We fill the InfluxDB with response times, that rise over the time. The data points are not on the same straight line, but are distributed around a regression line with positive slope. For the response times we start at 3000 ms. One data point has the prior data point's response time increased by a random number between 3500 ms and 7000 ms, and the next data point has the prior data point's response time decreased by a random number between 300 ms and 3000 ms. This alternating pattern is continued through entire data. It provides a rise of response times, but adds some variance to the response times. Figure 7.5 shows the InfluxDB and a curve that goes through the 100 data points.

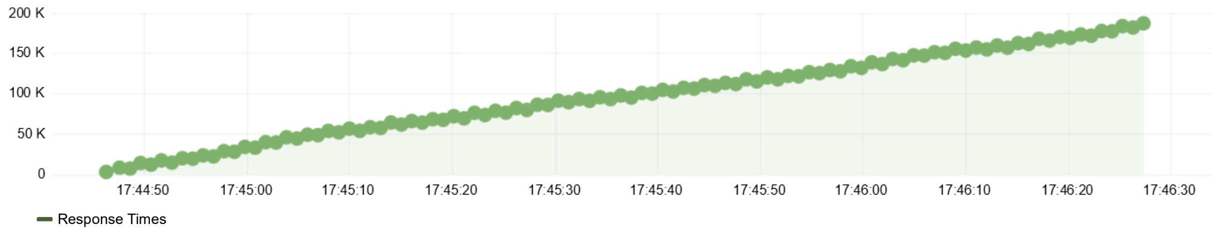

Figure 7.5.: Rise of response times

Expectations: The Ramp rule should detect the Ramp anti-pattern. There should be no Application Hiccups, since there is no outlier data point. The Traffic Jam rule should not execute, since the slope is to high when the Ramp anti-pattern is detected.

Results: We have a high slope for the regression line. Its value is ~ 1.7. Thereby, the Ramp anti-pattern is detected. Neither the general Application Hiccups rule nor its alternative detected an outlier data point.

Slope	Coeff. of Var.	App. Hiccups	(ALT) App. Hiccups	SPA
~ 1.7	-	0	0	The Ramp

Table 7.3.: Test Case 3 - Results

Result discussion: The results match our expectations. The slope in this test case is higher than the slope from the prior test case. This is because the last data point in the InfluxDB has nearly a response time of 200000 ms. There is also some variance in the response times, but the high slope prevents the Traffic Jam rule from further executing. This is what we wanted to achieve.

Test Case 4 - Rise of Response Times with Outliers

Description: We fill the InfluxDB with response times, that rise over the time. The data points are not on the same straight line, but are distributed around a regression line with positive slope. For the response times we start at 3000 ms. One data point has the prior data point's response time increased by a random number between 600 ms and 1000 ms, and the next data point has the prior data point's response time decreased by a random number between 100 ms and 500 ms. This alternating pattern is continued through entire data. It provides a rise of response times, but adds also some variance to the response times. For the 25th, the 50th and the 75th data point we set an outlier

response time. For the outlier values we set a random value between 50000 ms and 100000 ms. Figure 7.6 shows the InfluxDB and a curve that goes through the 100 data points.

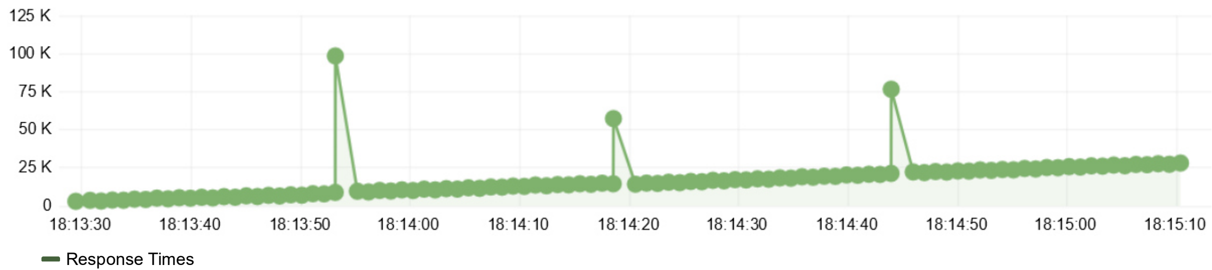

Figure 7.6.: Rise of response times with outliers

Expectations: The Ramp rule should detect the Ramp anti-pattern. There should be also Application Hiccups, since there are outlier data points. The Traffic Jam rule should not execute, since the slope is to high when the Ramp anti-pattern is detected.

Results: The slope of the regression line is ~ 0.24. The Ramp is detected. Both of the Application Hiccups rules detect three hiccups. The values for the outlier data points are 98481 ms, 57329 ms and 76601 ms for the third outlier.

Slope	Coeff. of Var.	App. Hiccups	(ALT) App. Hiccups	SPA
~ 0.24	-	3	3	The Ramp Appl. Hiccups

Table 7.4.: Test Case 4 - Results

Result discussion: The results match our expectations. The rules correctly did detect the Ramp anti-pattern and Application Hiccups. But here we notice the limitation of the alternative Application Hiccups rule. When we set the value of the crossing threshold to a value higher than one of the values of the three outlier data points, the rule detects less Application Hiccups than the general Application Hiccups rule. The default value for the crossing threshold is 50000 ms and therefore it detects also all three hiccups. So the disadvantage of the alternative Application Hiccups rule is, that the value of the crossing threshold has to be adapted to correspond to the current values in the TSDB.

The Ramp rule does not remove outliers before calculating the slope. The slope is higher because of them. The rule should be modified to exclude outliers first. This is a recommended work for the future.

Test Case 5 - Variance of Response Times

Description: We fill the InfluxDB with random values between 1000 ms and 10000 ms. For this purpose, we use the class `ThreadLocalRandom` [Jav16]. With the following Java code, we can generate a random Integer value.

```
ThreadLocalRandom.current().nextInt(int origin, int bound)
```

where `origin` is the least value generated and `bound` is the maximum value generated. After filling the InfluxDB with the 100 random values, there should be a high variance of response times. Figure 7.7 shows the InfluxDB and the distribution of the 100 data points.

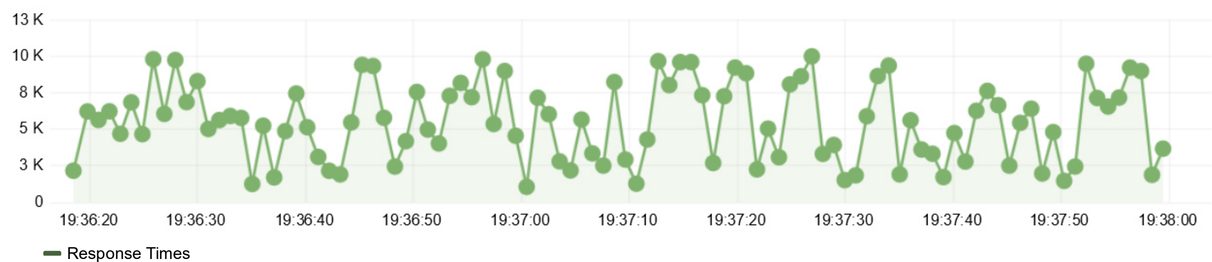

Figure 7.7.: Variance of response times

Expectations: There should be no Ramp detected, since the slope of the regression line is to low. Thereby, the Traffic Jam rule executes and should detect a high variance of response times, because the data points are distributed between 1000 ms and 10000 ms. There could be Application Hiccups, if the Application Hiccups rule detects an outlier response time.

Results: The slope of the regression line is ~ 0.0. The coefficient of variation has a value of ~ 0.48 and therefore the Traffic Jam anti-pattern is detected. The default value for the coefficient of variation threshold is 0.3. None of the Application Hiccups rules detected hiccups.

Slope	Coeff. of Var.	App. Hiccups	(ALT) App. Hiccups	SPA
$\sim$ 0.0	$\sim$ 0.48	0	0	Traffic Jam

Table 7.5.: Test Case 5 - Results

Result discussion: The Traffic Jam rule correctly detects a high variance of response times. The alternative Application Hiccups rule did not detect a hiccup, since all response times are below 50000 ms. The general Application Hiccups rule did also not detect any hiccup, since high and low response times are distributed similar.

Test Case 6 - Detecting Garbage Collection Hiccups

Description: We fill the InfluxDB with 100 random response times between 3000 and 5000 ms. We modify the 25th, the 50th and the 75th data point to become an outlier with the values between 50000 ms and 75000 ms. This time, we add to each data point also a synthetic corresponding GC time beside the response times into the the database. The GC time is for each data point a random value between 500 and 3000 ms, except for the 25th, the 50th and the 75th data point, where we have outlier response times. We here choose a random value between 25000 ms and 50000 ms. Figure 7.8 shows the InfluxDB and the distribution of the 100 data points' response times and GC times.

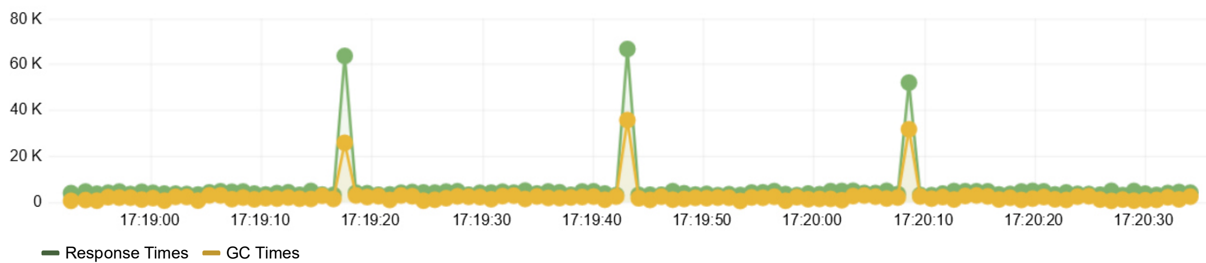

Figure 7.8.: Time series that contains also GC times

Expectations: When an Application Hiccup is caused by the GC, then the GC time is higher than 50 percent of the hiccups' response time. We expect that from three Application Hiccups at least one is caused by the GC. There should be no Ramp, since the regression line through the data points is horizontal. There should be Traffic Jam, because the random outlier response times are way higher than the other response times. This causes a high standard deviation.

Results: The slope of the regression line is ~ 0.0. The value of the coefficient of variation is ~ 1.74, so the Traffic Jam anti-pattern is detected. From three Application Hiccups, two are caused by the GC. This applies to the second and the third hiccup.

Slope	Coeff. of Var.	App. Hiccups	GC Hiccups	SPA
~ 0.0	~ 1.74	3	2	Application Hiccups Garbage Collection Hiccups Traffic Jam

Table 7.6.: Test Case 6 - Results

Result discussion: We detected a Traffic Jam, although Figure 7.8 shows no high variance of response times. There are only three outliers that lead to the high value of the coefficient of variation. The rule therefore detects wrongly a Traffic Jam and thereby delivers a false positive result. What would be better for the Traffic Jam rule is to exclude outlier data points. This is recommended work for the future.

The rule for detecting Garbage Collection hiccups worked as expected. Two of the outlier response times have a corresponding GC time value, that is higher than 50 percent of the response time. Since the predefined percent in the end decides, whether an Application Hiccup is a Garbage Collection Hiccup, the rule can be improved by reconsidering its concept and thereby making the rule independent of such a value.

Test Case 7 - Random Real Trace Data with Performance Problems

Description: For the first test case considering real trace data, we fill the InfluxDB with random trace data from the monitored DVD Store. We enabled performance problems within the store, like the 'Slow Search' function, where unnecessary complicated database queries are performed, and the 'Slow Display of Latest Orders Page' function, where an inefficient sorting algorithm is used. We set the threshold for the alternative Application Hiccups rule to 10000 ms, otherwise some of the hiccups would not be detected. Figure 7.11 shows the InfluxDB with the response times.

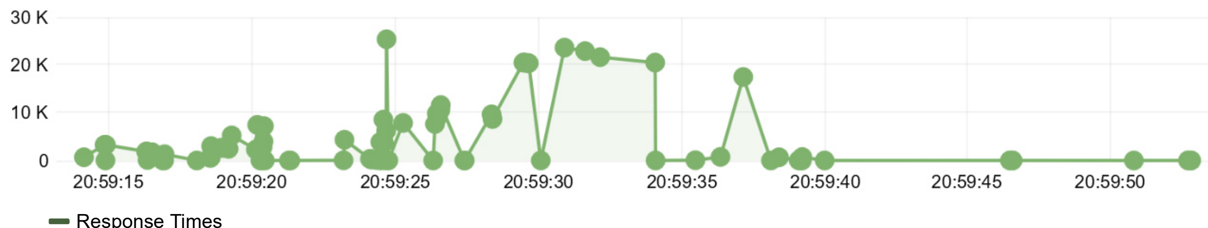

Figure 7.9.: Time series that contains real trace data

Expectations: There should be Application Hiccups, since there are outlier response times. In the TSDB there is no rise of response time visible, but a high variance of response times. Therefore, the Traffic Jam anti-pattern should be detected.

Results: The slope of the regression line is ~ 0.0, so no Ramp is detected. The value of the coefficient of variation is ~ 1.54 and therefore the Traffic Jam anti-pattern is detected. The general Application Hiccups rule detects 9 hiccups, whereas the alternative Application Hiccups rule detects 5 hiccups.

Slope	Coeff. of Var.	App. Hiccups	(ALT) App. Hiccups	SPA
~ 0.0	~ 1.54	9	5	Application Hiccups Traffic Jam

Table 7.7.: Test Case 7 - Results

Result discussion: Our expectations are fulfilled, except that the alternative Application Hiccups rule detected less hiccups than the general Application Hiccups rule. This is because some of the outlier response times follow right after an outlier value, that already exceeded the crossing threshold. The general Application Hiccups rule, however, counts all outliers as hiccups. The approach of the alternative Application Hiccups rule is relating to this aspect better, because a hiccup is defined as a rise of response times followed by a down of response times. The disadvantage of the alternative Application Hiccups rule is that the value of the crossing threshold has to be adapted to correspond to the current values in the TSDB.

Test Case 8 - Real Trace Data with Ramp

Description: To manifest the Ramp anti-pattern within the NovaTec DVD Store, we enable several performance problems within the store, like the 'Slow Search' function, where unnecessary complicated database queries are performed, and the 'Slow Display of Latest Orders Page' function, where an inefficient sorting algorithm is used. We let inspectIT collect traces in a longer observation period, since the longer LoadIT produces load on the DVD Store, the more the response times rise over time. We take an excerpt of traces and save them into the TSDB. For the alternative Application Hiccups rule, we set the crossing threshold value to 10000 ms. Figure 7.11 shows the InfluxDB for the rise of response times.

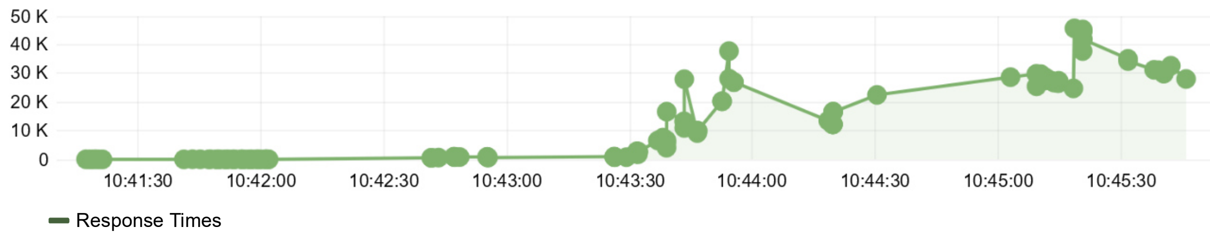

Figure 7.10.: Time series that contains Ramp anti-pattern

Expectations: The rules should detect the Ramp anti-pattern, since the slope of the regression line is high enough. Thereby, no Traffic Jam anti-pattern should be detected. There could be Application Hiccups, when an outlier response time is detected.

Results: The slope of the regression line is ~ 0.14 and therefore the Ramp anti-pattern is detected. The general Application Hiccups rule detected no hiccup, whereas the alternative Application Hiccups rule detected one hiccup.

Slope	Coeff. of Var.	App. Hiccups	(ALT) App. Hiccups	SPA
~ 0.14	-	0	1	The Ramp

Table 7.8.: Test Case 8 - Results

Result discussion: The Ramp anti-pattern is detected correctly within the time series data, like expected. The alternative Application Hiccups rule detected one hiccup, since one data point exceeds the value of the crossing threshold and is followed by a data point that is below the value. It appears open to interpretation, whether the result we obtained here is wrong, since Figure 7.11 shows a periodic rise of response times and is followed by a down of response times just before the response times rise again. For future work it would be good to define exactly, what an Application Hiccup is.

Test Case 9 - Real Trace Data with Ramp and Application Hiccups

Description: We enable the 'Slow Search' function in the DVD Store, where unnecessary complicated database queries are performed to the database. For this test case, we only let 6 users access the store, whereas in the other test cases with real trace data 300 users access the store. With this test case, we only want to show limitations of the Ramp rule and the Traffic Jam rule. Figure 7.11 shows the InfluxDB with filled trace data. We

set the threshold for the alternative Applications Hiccups rule to 5000 ms, otherwise it would not detect a single hiccup.

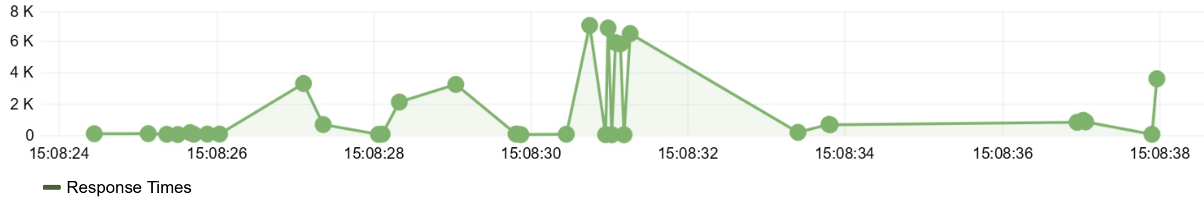

Figure 7.11.: Time series that contains Ramp and Application Hiccups

Expectations: There should be Application Hiccups, since there are existing outlier response times. There should be no Ramp, since the slope of the regression line is to low. There should be Traffic Jam, because of the high variance of response times.

Results: The slope of the regression line is ~ 0.1. The Ramp is detected. Therefore, we have no Traffic Jam. Both Application Hiccups rules detected 4 hiccups.

Slope	Coeff. of Var.	App. Hiccups	(ALT) App. Hiccups	SPA
~ 0.1	-	4	4	Application Hiccups The Ramp

Table 7.9.: Test Case 9 - Results

Result discussion: Contrary to our expectations, we detected the Ramp anti-pattern and not the Traffic Jam anti-pattern. When we set the slope threshold to a value higher than 0.1, than we detect the Traffic Jam anti-pattern with a coefficient of variation value of ~ 1.56. The problem for both rules is, that sometimes the thresholds have to be adapted to correspond to the current data points in the TSDB. In Figure 7.11, we see rather a Traffic Jam than a Ramp in the time series data and therefore both rules deliver a wrong result. The Ramp rule provides a false positive result, and the Traffic Jam rule provides a false negative result. The problem is also, that the Traffic Jam rule is dependent from the Ramp rule. We recommend in future work to improve the rules, so that the differentiation of both anti-pattern detections is more sophisticated.

As already discussed in test case 4, the slope of the regression line would be lower if the Ramp rule would exclude outliers before calculating the slope.

Test Case 10 - Real Trace Data with Overload Situations

Description: Enabling performance problems within the DVD Store leads to high response times, where in the most cases the system does not execute much on the CPU. Figure 7.12 shows the InfluxDB with filled data from inspectIT traces. Response times are high, and CPU times are low. We set the threshold for the alternative Application Hiccups rule to 5000 ms, otherwise it would not detect a single hiccup.

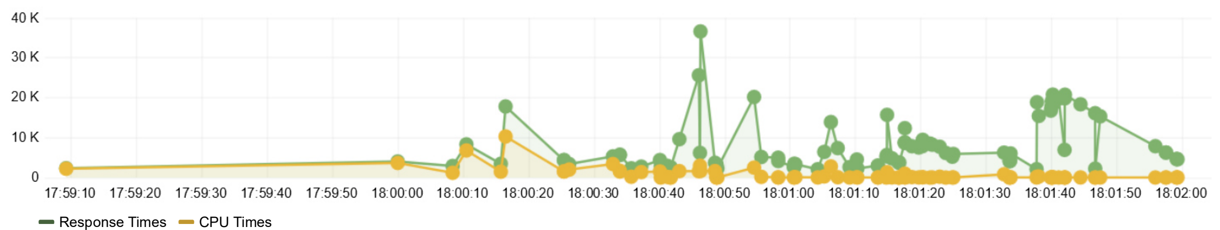

Figure 7.12.: Time series that contains overload situations

Expectations: Several anti-patterns should be detected. There must be Application Hiccups, since the InfluxDB contains outlier data points. The variance of response times is also high, so there should be Traffic Jam. The More is Less rule should detect an overload situation, since there are high response times with corresponding low CPU times.

Results: The alternative Application Hiccups rule detected 11 hiccups, the general Application Hiccups rule 14. The coefficient of variation is ~ 0.83, and the Traffic Jam anti-pattern is detected. The More is Less rule detected an overload situation, where the problematic data point has a response time of ~ 20783.64 ms and only a CPU time of 93.75 ms, whereas the average response time within the time series data is ~ 8034.18 ms and the average CPU time is ~ 665 ms. The CPU time to response time ratio from the problematic data point is ~ 0.005, the ratio within the time series data is ~ 0.04.

Slope	Coeff. of Var.	App. Hiccups	(ALT) App. Hiccups	SPA
~ 0.0	~ 0.83	14	11	Application Hiccups Traffic Jam More is Less

Table 7.10.: Test Case 10 - Results

Result discussion: The rules worked as expected. The alternative Application Hiccups rule detected less hiccups than the general rule. This is because some of the outlier response times follow right after an outlier value, that already exceeded the crossing threshold. The general Application Hiccups rule, however, counts all outliers as hiccups. The approach of the alternative Application Hiccups rule is relating to this aspect better, because a hiccup is defined as a rise of response times followed by a down of response times.

The More is Less rule detected an overload situation. The rule worked as expected, since we have a high response time, that is higher than the average response time, with a very low corresponding CPU time to response time ratio, that is lower than the average ratio of all data points.

Test Case 11 - Real Trace Data with no Overload Situations

Description: Enabling performance problems within the DVD Store leads to high response times, where in the most cases the system does not execute much on the CPU. In contrast to test case 10, we let the DVD Store execute longer, so that we have more load on the Store. Figure 7.13 shows the InfluxDB with filled data from inspectIT traces. Response times are high, and CPU times are low. This time we do not change the value for the threshold of the alternative Application Hiccups rule (default value is 50000 ms).

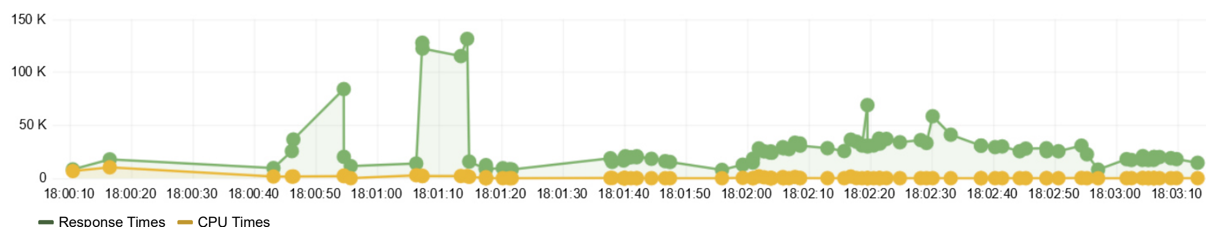

Figure 7.13.: Time series that contains no overload situations

Expectations: Several anti-patterns should be detected. There must be Application Hiccups, since the InfluxDB contains outlier data points. The variance of response times is also high, so there should be Traffic Jam. The More is Less rule should detect an overload situation, since there are high response times with corresponding low CPU times.

Results: The general Application Hiccups rule detected 7 hiccups, whereas the alternative Application Hiccups rule detected 4 hiccups. The value of the coefficient of variation is $\sim$ 0.83, so the Traffic Jam anti-pattern is detected. The More is Less rule detected no overload situation. The average response time of the monitored system is $\sim$ 28298.74 ms and the average CPU time is $\sim$ 542.97 ms.

Slope	Coeff. of Var.	App. Hiccups	(ALT) App. Hiccups	SPA
$\sim$ -0.07	$\sim$ 0.83	7	4	Application Hiccups Traffic Jam

Table 7.11.: Test Case 11 - Results

Result discussion: The alternative Application Hiccups rule detected less hiccups than the general rule. This is because some of the outliers do not exceed the threshold of 50000 ms. When improving the general Application Hiccups rule relating to the aspect, that it counts all outliers as hiccups, then the rule definitely is better than its alternative, since it is independent from a threshold.

The More is Less rule this time did not alert. The reason is that the average CPU time to response time ratio from all data points is only $\sim$ 2 percent. There are very high response times with a duration of over 100000 ms and corresponding low CPU times, as we can see in Figure 7.13. There is for example a data point, that has a response time of $\sim$ 131544.28 ms and a CPU time of 2093.75 ms, but the rule did not detect it as an overload situation, since the average CPU time of all data points is only $\sim$ 542.97 ms. The rule could not fulfill our expectation, but worked correctly, because all data points have a low corresponding CPU time. However, the predefined percent of the More is Less rule finally decide, whether there is an overload situation in the monitored system or not. For future work, we recommend to elaborate a More is Less rule, that is independent from such predefined values.

7.3. Threats to Validity

Internal threats: For the evaluation of the N+1 Query Problem, in addition to traces from the sample application, we used self made synthetic traces to show its limitations. With the real traces we could not show the limitations, since they do not cover the following cases. The first case is a trace, that contains one different query followed by n equal queries, but actually the first query has nothing to do with the other queries. The second case is a trace, where the N+1 Query Problem does not start right at the beginning of a subtree, but is followed by some other queries.

To obtain realistic traces instead of the synthetic traces, problem injection can be used. With the injection it can be defined, which SQL queries the application performs. The present problem injection to the DVD Store leads to the traces we already used for evaluation, but cannot show the limitations of the rule.

For the Expensive Computation rule evaluation, and for test cases 1-6 in the evaluation of rules based on time series, we used synthetic traces or synthetic trace data for evaluation. The results should be the same when applying the rule on real traces or real trace data, but this cannot be guaranteed. For a better evaluation, the tests should be reconsidered and real traces or real trace data should be used.

For the Expensive Computation rule, traces from inspectIT can be used, since they include CPU time data. To do this, an OPEN.xtrace adapter for the tool has to be implemented. For the test cases 1-6 of the evaluation of rules based on time series, instead of using synthetic trace data, problem injection and then to export traces from inspectIT would be more appropriate.

External threats: Since we obtained wrong results in some tests when using particular sample traces or sample trace data, the results cannot be generalized. Wrong results are obtained when detecting an anti-pattern, although it is not manifested in a trace or trace data (false positive), and not detecting an anti-pattern, although it is manifested in a trace or trace data (false negative). For the rules that delivered these wrong results, the analysis algorithm has to be optimized.

Construct threats: We applied the N+1 Query Problem and the Stifle rule only on traces from CA Introscope. We cannot guarantee, that the rules deliver the same results on traces from other APM tools. The measured data in the TSDB is from a short time interval. We cannot guarantee, that the rules deliver the same results when they are applied on data from longer time intervals.

Conclusion threats: The number of data points within the TSDB may not be sufficient to reflect real use cases. The rules have to be executed on the monitoring data obtained over longer time intervals, where more data points are obtained.

Chapter 8

Conclusion

In this chapter, we summarize what we did throughout this thesis and discuss, whether we have reached our goals. The summary is in Section 8.1, and the discussion about reaching the goals is in Section 8.2. Furthermore, we recommend work for the future in Section 8.3, that could not be done in the scope of this thesis.

8.1. Summary

Throughout this thesis, we developed approaches for detecting SPAs in enterprise applications. The approaches are based on analyzing runtime data from the application. We get the runtime data with the help of APM tools, that are coupled with the tool diagnoseIT. We split the diagnoseIT analysis into three analysis parts.

The first part deals with the detection of SPAs based on a single execution trace and is called detection based on trace data. diagnoseIT receives a problematic trace and starts a root cause analysis by applying rules to the trace to get insights. The rules are stored inside the diagnoseIT rule engine. A rule describes the characteristics of an anti-pattern and when the analyzed runtime data matches the characteristics, the rule detects the anti-pattern. We investigated several anti-patterns. Some of them can be detected by the analysis of a single trace, and others cannot. We implemented the rules for detecting the N+1 Query Problem, the Stifle, Expensive Computation and Phantom Logging anti-pattern.

For anti-patterns, that cannot be detected by analyzing a single trace, we introduced a completely new approach. These anti-patterns become visible over a longer observation period and require the analysis of a series of data. We integrated a TSDB into the diagnoseIT, so we can save particular trace data. This detection, based on time series analysis, is triggered periodically. The time series data is fetched from the database and

rules are applied to the data. Through this new approach, we now can detect the Ramp, Traffic Jam and More is Less anti-pattern.

The detection using combined approach combines the previous analysis parts, where a problematic data point from the TSDB can be linked to a trace, that can then be analyzed further. For this purpose, we use the TraceID, that is assigned to each incoming trace for analysis by the diagnoseIT. We implemented two rules based on this approach. One detects Application Hiccups within the time series. Data points, that cause the hiccups, can be analyzed further in the detection based on trace data analysis. The rule returns a list with problematic data points with corresponding TraceIDs. The second rule investigates Application Hiccups further and checks, whether they are caused by the GC. Like the Application Hiccups rule, the rule returns data points, that cause the Garbage Collection Hiccups.

Finally, we evaluated the implemented rules to check, whether they correctly detect SPAs. The evaluation showed the applicability of the rules, but has also uncovered limitations of the rules. We used different evaluation approaches for the rules that work on single traces, and the rules that work on time series. For the rules that work on single traces, we applied the rules simply on trace data from a real application or synthetic trace data, and investigated, whether the rules correctly detect an anti-pattern. For the rules that work on time series data, we filled the TSDB first with synthetic trace data, and then with trace data from a real application. We then applied the rules on the time series data and checked, whether the results did match our expectations.

8.2. Retrospective

In Section 1.2, we described the goals for this thesis. Here we discuss whether we have reached the goals.

The first goal was to evaluate the diagnoseIT approach and the OPEN.xtrace. We investigated the structure of the OPEN.xtrace and described it in Section 2.5. For the implementation of the rules, we defined which metrics from the OPEN.xtrace are necessary to detect the anti-patterns. We investigated the diagnoseIT rule engine and how it works. Results from this task are in Section 3.3.

The second goal of this thesis was to access historical data in diagnoseIT. That means, we wanted to find a way to detect anti-patterns, that can be detected by analyzing a series of trace data. We successfully reached this goal. We extended the diagnoseIT with a TSDB, to save particular trace data from incoming traces to the diagnoseIT into a database.

The third goal of this thesis was to research SPAs in literature. Furthermore, we wanted to investigate, whether the anti-pattern can be detected with the current diagnoseIT approach and the available data in the OPEN.xtrace. We reached this goal, since we did find out, which anti-patterns can be detected and which not. Detectable anti-patterns are listed in Section 3.1, and not detectable anti-patterns are listed in Section 3.2. We explained for each anti-pattern, why it can be detected or not.

The fourth goal of this thesis was to develop concepts and rules for the detectable anti-patterns. Furthermore, they had to be implemented in Java code. In Chapter 4, Chapter 5 and in Chapter 6, we presented concepts and rules for the detection of the anti-patterns. We split the diagnoseIT analysis intro three parts, where each of them describes another concept of the diagnoseIT analysis. Thereby, we implemented rules for each analysis part, that can all be found in Appendix A.

The last goal of this thesis was the evaluation of the rule implementations. We created a test plan, where we used different approaches for the evaluation of rules, that work on single traces and for the evaluation of rules, that work on time series data. We applied the rules to synthetic traces as well as to traces, that we obtained by monitoring a real application. We saw that the rules, in the majority of tests, work as expected, but we saw also some limitations of the rules, where false positive or false negative results are delivered. For each test, where we saw wrong results, we made recommendations how the rules can be improved, so that the rules in all scenarios work as expected. Therefore, the evaluation of the rules was successful.

8.3. Future Work

In this section, we present possible future work, that could not be done in the scope of this thesis.

The first recommended work is to improve the implemented rules from this thesis. In Chapter 7, we proposed how the rules can be improved. We summarize for the rules in the following, how an improvement can be achieved. Table 8.1 provides the possible improvements.

Rule	Improvement
N+1 Query Problem	Analyze SQL statements Detect also N+1 Query Problems, that do not start right at the beginning of a subtree
The Stifle	Analyze SQL statements
Expensive Computation	Redefine, what an Expensive Computation is Investigate further Expensive Computations
The Ramp	Clear distinction from Traffic Jam rule Exclude outlier data points
Traffic Jam	Clear distinction from the Ramp rule Exclude outlier data points
More is Less	Reconsider the rule concept and make it independent from predefined values
Application Hiccups	Do not consider each outlier as a hiccup
Alternative Application Hiccups	Detect hiccups independent from a threshold Redefine, what a hiccup is
Garbage Collection Hiccups	Detect hiccups independent from a predefined value

Table 8.1.: Possible improvements

If SQL statements can be analyzed, a rule for detecting the Circuitous Treasure Hunt anti-pattern can be implemented, since by analyzing the SQL statements it can be differentiated from the N+1 Query Problem. Furthermore, the rules should provide solutions, when they detect an anti-pattern. In the previous chapters, we proposed solutions to a particular anti-pattern, but the rules do not consider these.

If using the combined approach detects a problematic data point, it is not further analyzed in the detection based on trace data analysis. It is recommended, to analyze problematic data points further, since they are the cause for the performance problems within the time series data.

Furthermore, the evaluation of the rules that work on time series data was not performed on the data from longer observation periods. The investigated time periods were really short and did not last longer than few minutes. Therefore, the rules should be tested on trace data, that is collected over hours or even days of observations.

What the thesis did not consider is the detection of technology-dependent anti-patterns, where technologies are misused during the implementation of a software. All investigated anti-patterns throughout this thesis are technology-independent. Examples

for technology-dependent anti-patterns are Hibernate anti-patterns [Węg13]. For the detection of such anti-patterns further rules can be implemented.

The literature can be researched further for SPAs, that can be detected with the help of the APM tool diagnoseIT. When an anti-pattern cannot be detected due to unavailable data in the OPEN.xtrace, the OPEN.xtrace could be extended by including the missing data. Some of the listed anti-patterns in Section 3.2 cannot be detected due to missing data in the OPEN.xtrace, like the Empty Semi Trucks or the Dormant References anti-pattern.

Appendix A

Rules

In this chapter, we provide Java codes of the rule implementations to detect SPAs. For some of the rules, we provide only excerpts with the main part of the rule.

Listing A.1 N+1 Query Problem rule

```
@Rule(name = "NPlusOneRule")
public class NPlusOneRule {

      // Threshold for the 'N' queries
      private static final int NUMBER_OF_CALLS_THRESHOLD = 10;

      // Requesting aggregated object with root causes
      @TagValue(type = rootCauses)
      private AggregatedObject cause;

      @Action(resultTag = N+1_Tag)
      public boolean action() {
            ...
            // Check, if type of root causes is DatabaseInvocation.
            // If so, parent of root causes is accessed. This is an executeQuery() method.
            // To get all executeQuery() methods on the same tree level, again the
            // parent of an executeQuery() method is accessed.
            ...
            LinkedList<DatabaseInvocation> listDatabaseInvocations = new
                  LinkedList<DatabaseInvocation>();
            TreeIterator<Callable> callableIterator = parent.iterator();

            // Iterate over all Callables and search for DatabaseInvocations
            while (callableIterator.hasNext()) {
                  Callable current = callableIterator.next();
                  if (current instanceof DatabaseInvocation) {
                        DatabaseInvocation databaseInvocation = (DatabaseInvocation)
                              current;
                        listDatabaseInvocations.add(databaseInvocation);
                  }
            }
            DatabaseInvocation firstInvocation = listDatabaseInvocation.get(0);
            DatabaseInvocation secondInvocation = listDatabaseInvocation.get(1);
            long amountOfNQueries = 0;

            // First query has to differ from the other queries
            if (!(secondInvocation.getSQLStatement().equals(firstInvocation
                        .getSQLStatement()))) {

                  // Count amount of the 'N' queries
                  for (int i = 2; i < causeInvocations.size(); i++) {
                        DatabaseInvocation databaseInvocation = (DatabaseInvocation)
                              causeInvocations.get(i);
                        if (databaseInvocation.getSQLStatement().equals(
                                    secondInvocation.getSQLStatement())) {
                              amountOfNQueries++;
                        }
                  }
            }
            if (amountOfNQueries > NUMBER_OF_CALLS_THRESHOLD) {
                  return true;
            }
      return false;
      }
}
```

Listing A.2 The Stifle rule

```
@Rule(name = "TheStifleRule")
public class TheStifleRule {

      // Threshold for amount of queries
      private static final int NUMBER_OF_CALLS_THRESHOLD = 10;

      // Requesting aggregated object with root causes
      @TagValue(type = rootCauses)
      private AggregatedObject cause;

      // Requesting result of N+1 Query Problem rule
      @TagValue(type = N+1_Tag)
      private boolean resultOfNPlusOneRule;

      @Action(resultTag = Stifle_Tag)
      public boolean action() {

            if(resultOfNPlusOneRule == true){
                  return true;
            }
            ...
            // Check, if type of root causes is DatabaseInvocation. If not, return false.
            // Parent of a root cause is an executeQuery() method. Try to access
            // parent of executeQuery() method, to get all children on the same tree level
            // and put the children into listDatabaseInvocations. If executeQuery()
            // has no parent, put available root causes into listDatabaseInvocations.
            ...
            HashMap<String, Long> queryMap = new HashMap<String, Long>();

            for (DatabaseInvocation invocation : listDatabaseInvocations) {
                  String sqlCommand = invocation.getSQLStatement();

                  // Check, if HashMap already contains the SQL statement
                  if (queryMap.containsKey(sqlCommand)) {

                        // Increase amount value by one
                        queryMap.put(sqlCommand, queryMap.get(sqlCommand) + 1);
                  } else {
                        // Create new entry and set amount value to one
                        long amount = 1;
                        queryMap.put(sqlCommand, amount);
                  }
            }
            ...
            // Check, if one of the entries' amount value exceeds the threshold.
            // If so, the rule returns true
            ...
            return result;
      }
}
```

Listing A.3 Expensive Computation rule

```
@Rule(name = "ExpensiveComputationRule")
public class ExpensiveComputationRule {

      // Threshold ratio of exclusive CPU time to trace response time
      private static final double CPU_TIME_RATIO = 0.10;

      // Requesting a trace to analyze
      @TagValue(type = Tags.ROOT_TAG)
      private Trace trace;

      // The baseline value is 1000 ms, problematic traces exceed this value
      @SessionVariable(name = VAR_BASELINE, optional = false)
      private double baseline;

      @Action(resultTag = ExpensiveComputation_Tag)
      public TimedCallable action() {

            long traceResponseTime = trace.getResponseTime();
            MethodInvocation methodInvoHighestExclCPUTime = null;
            long highestExclusiveCPUTime = 0l;
            List<MethodInvocation> methodInvocations = new LinkedList<MethodInvocation>();

            if (traceResponseTime >= baseline) {
                  for (Callable callable : trace) {

                        // Find MethodInvocations that provide excl. CPU times
                        if (callable instanceof MethodInvocation) {
                              MethodInvocation methodInvo = (MethodInvocation) callable;
                              if (methodInvo.getCPUTime().isPresent()) {
                                    methodInvocations.add(methodInvo);
                              }
                        }
                  }
                  if (methodInvocations.size() >= 1) {
                        for (int i = 0; i < methodInvocations.size(); i++) {
                              if (methodInvocations.get(i).getExclusiveCPUTime().get() >
                                    highestExclusiveCPUTime) {
                                    highestExclusiveCPUTime = methodInvocations.get(i)
                                                .getExclusiveCPUTime().get();
                                    methodInvoHighestExclCPUTime =
                                                methodInvocations.get(i);
                              }
                        }
                        double CPUTimeRatio = ((double) methodInvoHighestExclCPUTime
                                    .getExclusiveCPUTime().get())
                                    / ((double) traceResponseTime);

                        if (CPUTimeRatio > CPU_TIME_RATIO) {
                              return methodInvoHighestExclCPUTime;
                        }
                  }
            }
            return null;
      }
}
```

Listing A.4 Phantom Logging rule

```
@Rule(name = "PhantomLoggingRule")
public class PhantomLoggingRule {

      // The assumed log level of the monitored system
      private static final LoggingLevel LOG_THRESHOLD = LoggingLevel.DEBUG;

      // Requests trace to analyze
      @TagValue(type = Root_Tag)
      private Trace trace;

      @Action(resultTag = PhantomLogging_Tag)
      public boolean action() {

            for (Callable callable : trace) {

                  if (callable instanceof LoggingInvocation) {
                        LoggingInvocation loggingCallable = (LoggingInvocation) callable;

                        if (loggingCallable.getLoggingLevel().isPresent()) {

                              // Get the log level of current LoggingInvocation
                              String logLevel = loggingCallable.getLoggingLevel().get();

                              // Access integer value of log level
                              LoggingLevel level = LoggingLevel.valueOf(logLevel);

                              if(level.level < LOG_THRESHOLD.level){
                                    return true;
                              }
                        }
                  }
            }
            return false;
      }

      // The enum assigns an Integer value to each of the log levels
      enum LoggingLevel {
            ALL(0), TRACE(1), DEBUG(2), INFO(3), WARN(4), ERROR(5), FATAL(6), OFF(7);
            int level;

            LoggingLevel(int level) {
                  this.level = level;
            }
      }
}
```

Listing A.5 The Ramp rule

```
@Rule(name = "TheRampRule")
public class TheRampRule {

      // Threshold for the slope of the regression line
      private static final double SLOPE_THRESHOLD = 0.05;

      @TagValue(type = Root_Tag)
      // Connection to the InfluxDB
      private InfluxDBConnector connector;

      @Action(resultTag = Ramp_Tag)
      public boolean action() {

            List<DataPoint> dataPoints = new LinkedList<DataPoint>();

            // Fetches data points from InfluxDB
            dataPoints = getDataPoints();

            // Regression line
            SimpleRegression regression = new SimpleRegression();

            // Add data points
            for (DataPoint dataPoint : dataPoints) {
                  long timestamp = (long) dataPoint.getTimestamp();
                  long responseTime = (long) dataPoint.getResponseTime();
                  regression.addData(timestamp, responseTime);
            }

            // Slope of regression line has to be higher than threshold
            if (regression.getSlope() > SLOPE_THRESHOLD) {
                  return true;
            }

            return false;
      }
}
```

Listing A.6 Traffic Jam rule

```java
@Rule(name = "TrafficJamRule")
public class TrafficJamRule {

      // Threshold for coefficient of variation
      private static final double COEFFICIENT_OF_VARIATION_THRESHOLD = 0.3;

      // Threshold for the slope of the regression line
      private static final double SLOPE_THRESHOLD = 0.05;

      @TagValue(type = Root_Tag)
      // Connection to the InfluxDB
      private InfluxDBConnector connector;

      @Action(resultTag = TrafficJam_tag)
      public boolean action() {

      List<DataPoint> dataPoints = new LinkedList<DataPoint>();

      // Fetches data points from InfluxDB
      dataPoints = getDataPoints();

      ...
      // To differentiate from the Ramp rule first check, whether slope
      // of the regression line is below the SLOPE_THRESHOLD.
      // If so, rule executes further.
      ...

      double sumOfResponseTimes = 0;
      for(DataPoint dataPoint : dataPoints){
            sumOfResponseTimes += dataPoint.getResponseTime();
      }
      // Calculates the mean of response times
      double meanResponseTime = sumOfResponseTimes / dataPoints.size();

      // Calculates the variance of response times
      double tempVar = 0;
      for (DataPoint dataPoint : dataPoints)
            tempVar += (dataPoint.getResponseTime() - meanResponseTime)
                        * (dataPoint.getResponseTime() - meanResponseTime);
      double variance = tempVar / (dataPoints.size() - 1);

      // Calculates standard deviation of response times
      double standardDeviation = Math.sqrt(variance);

      // Calculates coefficient of variation
      double coeffOfVariation = standardDeviation / meanResponseTime;

      if (coeffOfVariation > COEFFICIENT_OF_VARIATION_THRESHOLD) {
            return true;
      }
      return false;
      }
}
```

Listing A.7 More is Less rule

```
@Rule(name = "MoreIsLessRule")
public class MoreIsLessRule {

      private static final double PERCENT_ONE = 0.4;
      private static final double PERCENT_TWO = 0.5;

      @TagValue(type = Root_Tag)
      // Connection to the InfluxDB
      private InfluxDBConnector connector;

      @Action(resultTag = MoreIsLess_Tag)
      public boolean action() {

            List<DataPoint> dataPoints = new ArrayList<DataPoint>();

            // Fetches data points from InfluxDB
            dataPoints = getDataPoints();

            double sumOfResponseTimes = 0;
            double sumOfCPUTimes = 0;

            for (DataPoint dataPoint : dataPoints) {
                  sumOfResponseTimes += dataPoint.getResponseTime();
                  sumOfCPUTimes += dataPoint.getCPUTime();
            }
            // Calculate average response time and average CPU time
            double avgResponseTime = sumOfResponseTimes / dataPoints.size();
            double avgCPUTime = sumOfCPUTimes / dataPoints.size();

            // Iterate through all data points and search for overload situations
            for (int i = 0; i < dataPoints.size(); i++) {
                  DataPoint currentObject = dataPoints.get(i);

                  if (currentObject.getResponseTime * PERCENT_ONE > avgResponseTime) {

                        double currentObjectCPUTimeRatio = currentObject.getCPUTime()
                                    / currentObject.getResponseTime();
                        double avgCPUTimeRatio = avgCPUTime / avgResponseTime;

                        if (currentObjectCPUTimeRatio < avgCPUTimeRatio * PERCENT_TWO) {
                              return true;
                        }
                  }
            }
            return false;
      }
}
```

Listing A.8 Application Hiccups rule

```
@Rule(name = "ApplicationHiccupsRule")
public class ApplicationHiccupsRule {

      @TagValue(type = Root_Tag)
      // Connection to the InfluxDB
      private InfluxDBConnector connector;

      @Action(resultTag = ApplicationHiccups_Tag)
      public List<DataPoint> action() {

      // Fetches data points from InfluxDB
      List<DataPoint> dataPoints = new LinkedList<DataPoint>();
      dataPoints = getDataPoints();

      // List for saving later outlier data points
      List<DataPoint> problematicDataPoints = new LinkedList<DataPoint>();

      // From list to array
      double[] responseTimesArray = new double[dataPoints.size()];
      for (int i = 0; i < dataPoints.size(); i++)
            responseTimesArray[i] = dataPoints.get(i).getResponseTime();

      double lowerQuartile = new Percentile()
                  .evaluate(responseTimesArray, 25);

      double upperQuartile = new Percentile()
                  .evaluate(responseTimesArray, 75);

      double interquartileRange = upperQuartile - lowerQuartile;

      double upperWhiskerThreshold = upperQuartile + interquartileRange * 1.5;

      // Outlier detection
      for (int i = 0; i < dataPoints.size(); i++) {
            if (dataPoints.get(i).getResponseTime() > upperWhiskerThreshold) {
                  problematicDataPoints.add(dataPoints.get(i));
            }
      }
      // return Application Hiccups
      return problematicDataPoints;
      }
}
```

Listing A.9 Alternative Application Hiccups rule

```
@Rule(name = "ApplicationHiccupsRule")
public class ApplicationHiccupsRule {

	// Crossing threshold
	private static final double CROSSING_THRESHOLD = 50000;

	@TagValue(type = Root_Tag)
	// Connection to the InfluxDB
	private InfluxDBConnector connector;

	@Action(resultTag = ApplicationHiccups_Tag)
	public List<DataPoint> action() {

	// Fetches data points from InfluxDB
	List<DataPoint> dataPoints = new LinkedList<DataPoint>();
	dataPoints = getDataPoints();

	// List for saving later data points, that are above the CROSSING_THRESHOLD
	List<DataPoint> problematicDataPoints = new LinkedList<DataPoint>();

	// The number of crossings of the CROSSING_THRESHOLD
	long numberOfCrossings = 0l;

	// Check, if first response time is already above the CROSSING_THRESHOLD
	boolean isLastTimeAboveThreshold = (dataPoints.get(0).getResponseTime() >
		CROSSING_THRESHOLD);

	for (int i = 1; i < dataPoints.size(); i++) {
		double currentTime = dataPoints.get(i).getResponseTime();

		if (currentTime > CROSSING_THRESHOLD) {
			problematicDataPoints.add(dataPoints.get(i));
		}

		// Algorithm to calculate the amount of crossings
		if ((isLastTimeAboveThreshold && currentTime < CROSSING_THRESHOLD)
				|| (!isLastTimeAboveThreshold && currentTime >
					CROSSING_THRESHOLD)) {
			numberOfCrossings++;
			isLastTimeAboveThreshold = !isLastTimeAboveThreshold;
		}
	}
	// Application Hiccups:
	long actualHiccups = numberOfCrossings / 2;
	...
	// Do something with actualHiccups
	...
	return problematicDataPoints;
	}
}
```

Listing A.10 Garbage Collection Hiccups rule

```
@Rule(name = "GarbageCollectionHiccupsRule")
public class GarbageCollectionHiccupsRule {

    private static final double PERCENT = 0.5;

    @TagValue(type = Root_Tag)
    // Connection to the InfluxDB
    private InfluxDBConnector connector;

    // Requesting results from Application Hiccups rule
    @TagValue(type = ApplicationHiccups_Tag)
    private List<DataPoint> appHiccupsDataPoints;

    @Action(resultTag = GarbageCollectionHiccups_Tag)
    public List<DataPoint> action() {

        // When there are no Application Hiccups, there cannot be GC Hiccups
        if (appHiccupsDataPoints.size() < 1) {
            return null;
        }

        List<DataPoint> outliersCausedByGarbageCollector = new
            LinkedList<DataPoint>();

        // Iterate through Application Hiccups and check, which one is caused
        // by the GC
        for (int i = 0; i < appHiccupsDataPoints.size(); i++) {

            double currentResponseTime = appHiccupsDataPoints.get(i)
                    .getResponseTime();
            double currentGCTime = appHiccupsDataPoints.get(i)
                    .getGCTime();

            if (currentGCTime > currentResponseTime * PERCENT) {
                outliersCausedByGarbageCollector.add(appHiccupsDataPoints.get(i));
            }
        }
        return outliersCausedByGarbageCollector;
    }
}
```

Appendix

Bibliography

[ap00] C. U. Smith, L. G. Williams. "Software performance anti-patterns." In: *In Proc. 2nd Int. Workshop on Software and Performance (WOSP '00)* (2000), pp. 127–136 (cit. on pp. iii, v, 1, 7, 9, 16, 17, 20, 21).

[Bur06] I. Burnstein. *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2006 (cit. on p. 65).

[CA16] CA Technologies. *CA Application Performance Management*. 2016. URL: http://www.ca.com/de/products/ca-application-performance-management.html (cit. on p. 58).

[CI11] B. Craven, S. M. Islam. *Ordinary least-squares regression*. SAGE Publications, 2011 (cit. on p. 44).

[Clo15] CloudScale. *One-Lane Bridge*. 2015. URL: http://wiki.cloudscale-project.eu/index.php/One-Lane_Bridge (cit. on p. 20).

[DAKW03] B. Dudney, S. Asbury, J. K. Krozak, K. Wittkopf. *J2EE antipatterns*. John Wiley & Sons, 2003 (cit. on p. 17).

[DGS02] R. F. Dugan Jr, E. P. Glinert, A. Shokoufandeh. "The Sisyphus database retrieval software performance antipattern." In: *Proceedings of the 3rd international workshop on Software and performance*. ACM. 2002, pp. 10–16 (cit. on p. 43).

[dIT15] NovaTec Consulting GmbH and University of Stuttgart (Reliable Software Systems group). *diagnoseIT Research Project*. 2015. URL: https://diagnoseit.github.io/ (cit. on pp. iii, v, 2, 11).

[Dyn16] Dynatrace. *Application monitoring*. 2016. URL: https://www.dynatrace.com/solutions/application-monitoring/ (cit. on p. 58).

[Eve98] B. Everitt. *Cambridge dictionary of statistics*. Cambridge University Press, 1998 (cit. on p. 45).

[Gra10] A. Grabner. *Top 10 Performance Problems taken from Zappos, Monster, Thomson and Co*. 2010. URL: http://apmblog.dynatrace.com/2010/06/15/top-10-performance-problems-taken-from-zappos-monster-and-co/ (cit. on pp. 20, 55).

[HHO+15] C. Heger, A. van Hoorn, D. Okanović, S. Siegl, A. Wert. "Expert-Guided Automatic Diagnosis of Performance Problems in Enterprise Applications." In: *EDCC Paper* (2015) (cit. on pp. iii, v, 2, 7, 10, 11, 32).

[IHE15] O. Ibidunmoye, F. Hernández-Rodriguez, E. Elmroth. "Performance anomaly detection and bottleneck identification." In: *ACM Computing Surveys (CSUR)* 48.1 (2015), p. 4 (cit. on p. 8).

[inf16a] influxdata. *InfluxDB*. 2016. URL: https://www.influxdata.com/time-series-platform/influxdb/ (cit. on pp. 5, 14, 42).

[inf16b] influxdata. *InfluxDB Documentation*. 2016. URL: https://docs.influxdata.com/influxdb/v1.1/ (cit. on pp. 14, 42).

[Jav16] Java™ Platform Standard Ed. 7. *Class ThreadLocalRandom*. 2016. URL: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadLocalRandom.html (cit. on p. 70).

[JSB97] D. F. Jerding, J. T. Stasko, T. Ball. "Visualizing interactions in program executions." In: *Proceedings of the 19th international conference on Software engineering*. ACM. 1997, pp. 360–370 (cit. on p. 8).

[KK54] J. F. Kenney, E. S. Keeping. "Mathematics of statistics-part one." In: (1954) (cit. on p. 45).

[Kop11] M. Kopp. *The Top Java Memory Problems – Part 2*. 2011. URL: http://apmblog.dynatrace.com/2011/12/15/the-top-java-memory-problems-part-2/ (cit. on pp. 19, 55).

[Mit10] T. Mitsa. *Temporal data mining*. CRC Press, 2010 (cit. on p. 13).

[MTL78] R. McGill, J. W. Tukey, W. A. Larsen. "Variations of box plots." In: *The American Statistician* 32.1 (1978), pp. 12–16 (cit. on pp. 51, 52).

[Nov] NovaTec Consulting GmbH. *loadIT*. URL: http://www.loadit.de/home/ (cit. on p. 59).

[Nov08] M. Novakovic. "Performance Anti-Patterns." In: *Java Magazin* Sonderdruck der Firma codecentric (2008) (cit. on pp. 18, 38).

[Nov11] NovaTec Consulting GmbH. *loadIT - DVD Store Demo*. 2011. URL: https://documentation.novatec-gmbh.de/display/LOADITDOC/loadIT+-+DVD+Store+Demo (cit. on p. 8).

[Nov16] NovaTec Consulting GmbH and University of Stuttgart (Reliable Software Systems group). *diagnoseIT Rule Engine*. 2016. URL: https://github.com/diagnoseIT/diagnoseIT (cit. on pp. 11, 21).

[NR09] M. Novakovic, A. Reitbauer. “O/R Mapping Anti-Patterns-Flush und Clear.” In: *JAVA Magazin* 1 (2009), p. 70 (cit. on p. 37).

[NTC16a] NovaTec Consulting GmbH. *inspectIT*. 2016. URL: http://www.inspectit.eu/ (cit. on p. 64).

[NTC16b] NovaTec Consulting GmbH. *NovaTec*. 2016. URL: http://www.novatec-gmbh.de/ (cit. on pp. iii, v).

[OHH+16] D. Okanović, A. van Hoorn, C. Heger, A. Wert, S. Siegl. “Towards Performance Tooling Interoperability: An Open Format for Representing Execution Traces.” In: *European Workshop on Performance Engineering*. Springer. 2016, pp. 94–108 (cit. on pp. 3, 8, 12, 58, 64).

[ÖR15] T. Ödegaard, Raintank Inc. *Grafana*. 2015. URL: http://grafana.org/ (cit. on p. 65).

[Ora16] Oracle Corporation. *Java*. 2016. URL: https://www.java.com (cit. on p. 5).

[PH14] M. Peiris, J. H. Hill. “Towards detecting software performance anti-patterns using classification techniques.” In: *ACM SIGSOFT Software Engineering Notes* 39.1 (2014), pp. 1–4 (cit. on p. 10).

[Pha] Phabricator. *Performance: N+1 Query Problem*. URL: https://secure.phabricator.com/book/phabcontrib/article/n_plus_one/ (cit. on pp. 31, 32).

[PM08] T. Parsons, J. Murphy. “Detecting Performance Antipatterns in Component Based Enterprise Systems.” In: *Journal of Object Technology 7, 3*. 2008, pp. 55–91 (cit. on p. 10).

[RM07] D. Rayside, L. Mendel. “Object ownership profiling: a technique for finding and fixing memory leaks.” In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM. 2007, pp. 194–203 (cit. on p. 19).

[SW] C. Smith, L. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. 2002* (cit. on p. 37).

[SW02a] C. U. Smith, L. G. Williams. “New software performance antipatterns: More ways to shoot yourself in the foot.” In: *Int. CMG Conference*. 2002, pp. 667–674 (cit. on pp. 10, 18, 43, 45).

[SW02b] C. U. Smith, L. G. Williams. “Software Performance AntiPatterns; Common Performance Problems and their Solutions.” In: *-CMG-CONFERENCE-*. Vol. 2. Citeseer. 2002, pp. 797–806 (cit. on p. 18).

[SW03] C. U. Smith, L. G. Williams. "More new software performance antipatterns: Even more ways to shoot yourself in the foot." In: *Computer Measurement Group Conference*. Citeseer. 2003, pp. 717–725 (cit. on pp. 17, 21, 37).

[Ten14] G. Tene. *Understanding Application Hiccups: An Introduction to the Open Source jHiccup Tool*. 2014. URL: http://www.azulsystems.com/webinar/understanding-application-hiccups-on-demand (cit. on pp. 18, 19).

[The16] The Apache Software Foundation. *The Apache Commons Mathematics Library*. 2016. URL: http://commons.apache.org/proper/commons-math/ (cit. on pp. 44, 52).

[Tru11] C. Trubiani. "Automated generation of architectural feedback from software performance analysis results." PhD dissertation. Università di L'Aquila, 2011 (cit. on pp. 2, 9, 47).

[Węg13] P. Węgrzynowicz. "Performance antipatterns of one to many association in hibernate." In: *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*. IEEE. 2013, pp. 1475–1481 (cit. on pp. 9, 85).

[Wer15] A. Wert. "Performance Problem Diagnostics by Systematic Experimentation." PhD dissertation. Karlsruher Institut für Technologie, 2015 (cit. on pp. 1, 2, 9, 18–21, 36, 37, 43, 55).

[Wik10] Wikimedia Commons. *Linear regression.svg*. 2010. URL: https://commons.wikimedia.org/wiki/File%3ALinear_regression.svg (cit. on p. 44).

[Wor16] World Wide Web Consortium (W3C). *Extensible Markup Language (XML)*. 2016. URL: https://www.w3.org/XML/ (cit. on p. 62).

[ZSNS09] W. Zucchini, A. Schlegel, O. Nenadic, S. Sperlich. *Statistik für Bachelor-und Masterstudenten: eine Einführung für Wirtschafts-und Sozialwissenschaftler*. Springer-Verlag, 2009 (cit. on p. 43).

All links were last followed on December 7, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature