Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor Thesis

Detecting Unusual Performance Behavior in Heterogeneous Environments

Dominik Hentrich

Course of Study: Softwaretechnik

Examiner: Dr.-Ing. André van Hoorn

Supervisor: Dr. Dušan Okanović

Commenced: January 3, 2018

Completed: July 3, 2018

CR-Classification: I.7.2

Abstract

Software applications provide a wide area of features and configurations, or process large datasets. Hence, certain performance problems only occur on specific constellations which are especially revealed when the application is used by customers. Application Performance Management (APM) was introduced to monitor and analyze performance data at runtime. In this thesis, an APM system was created to visualize the performance data of a Java application. It detects and alerts unusual performance behavior while the application is executed on several machines with several configurations. Therefore, a lot of the application's metadata is collected and compared.

In the field of automobile industry, Electronic Control Units (ECUs) and their software have become more and more complex. Configuration tools are used to handle this complexity and to generate the related source code. The processing of the large datasets leads to long duration. A configuration tool called DaVinci Configurator Pro of the company Vector Informatik is used for configuration, validation and generation of the necessary basic software and runtime environment of the ECUs. This tool provides a case study for the described APM system in this thesis.

The implementation of the APM system has shown that performance regressions of heterogeneous environments, which are caused by specific metadata can be detected by the reduction to a point anomaly detection problem and single comparisons for the several parameters of the metadata. The quality or reliability of the detection depends on the selection of the captured metadata. Regarding algorithms, the box plot rule-based algorithm as well as the nearest neighbor-based approach can be used for this task.

Kurzfassung

Software-Anwendungen bieten ein weites Spektum an Funktionen und Konfigurationen oder verarbeiten große Datenmengen. Dabei treten bestimmte Performanceprobleme nur bei gewissen Konstellationen auf, welche oft erst bei der späteren Nutzung zustande kommen. Application Performance Management (APM) wurde entwickelt, um Performancedaten zur Laufzeit zu analysieren. In dieser Arbeit wurede ein APM-System erstellt, welches Performance-Daten visualisiert. Dieses erkennt auffälliges Performance-Verhalten selbst und schlägt Alarm, während die Anwendung auf verschiedenen Computern mit unterschiedlichen Konfigurationen ausgeführt wird. Dafür wird ein weites Spektrum an Meta-Daten aufgezeichnet.

Im Bereich der Automobilindustrie ist die Ausstattung der Fahrzeuge durch Steuergeräte und deren Software immer komplexer geworden. Software-Konfigurationswerkzeuge werden genutzt, um Quellcode zu generieren und so die Komplexität leichter zu beherrschen. Die Verarbeitung der großen Datenmengen führt dabei zu hohen Laufzeiten. Die Konfigurationssoftware DaVinci Configurator Pro der Firma Vector Informatik wird für die Konfiguration, Validierung und Generierung benötigter Basis-Software und Laufzeitumgebung eingesetzt. Diese Software bietet eine Fallstudie für das beschriebene APM-System dieser Arbeit.

Die Implementierung des APM-Systems hat gezeigt, dass Performance-Probleme, welche im Zusammenhang mit bestimmten Meta-Daten auftreten, erkannt werden können, wenn diese auf das Problem der Punkt-Anomalie-Erkennung mit einzelnen Vergleichen für die jeweiligen Meta-Daten reduziert werden. Die Qualität oder Zuverlässigkeit der Erkennung ist von der Wahl der aufgezeichneten Meta-Daten abhängig. Was die Algorithmen betrifft, können sowohl der Box-Plot-als auch der Nearest-Neighbor-basierte Ansatz verwendet werden.

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 The Case Study . . . 2
- 1.3 Thesis Structure . . . 5

2 Foundations 7
- 2.1 The Foundations of APM . . . 7
- 2.2 Anomaly Detection . . . 9
- 2.3 Related Work . . . 16

3 The Implementation of the APM System 23
- 3.1 Fundamental Requirements and the Application Concept . . . 23
- 3.2 Instrumentation . . . 24
- 3.3 Storage of Collected Performance Data . . . 25
- 3.4 Visual Representation . . . 29
- 3.5 Anomaly Detection . . . 31

4 Evaluation 41
- 4.1 Evaluation Goals . . . 41
- 4.2 Description of Results . . . 42
- 4.3 Discussion of Results . . . 46

5 Conclusion and Future Work 49
- 5.1 Summary . . . 49
- 5.2 Future Work . . . 50

Bibliography 51

List of Figures

1.1 Draft of the APM System 2
1.2 Visualization of a project load step on the graphical user interface 3
1.3 Visualization of an evaluation step on the graphical user interface 4

2.1 The architecture of APM 8
2.2 Point Anomaly Example 11
2.3 Contextual anomaly example 12
2.4 Collective anomaly example 13
2.5 The definitions of the box plot 16
2.6 The architecture of Dynatrace AppMon 19
2.7 Selection of a whole package to observe with Dynatrace AppMon 20
2.8 Selection of the method to observe with Dynatrace AppMon 21
2.9 Dynatrace AppMon PurePaths 21

3.1 The added attributes of the implemented record types APM service . . . 25
3.2 The class diagram of the APM service 26
3.3 The database schema 28
3.4 An example for an ApmRecordSession instance 29
3.5 An example for an ApmRecord instance 29
3.6 An example for an ApmRecordProject instance 29
3.7 Showing details by the filtered measurement mode 31
3.8 The comparison of all methods by the comparison mode 32
3.9 The comparison of all versions by the comparison mode 32
3.10 Presentation problems of the time differences caused by anomalies . . . 33
3.11 Applying the logarithmic scale 33
3.12 Applying the zoom scale 34
3.13 Class diagram of the anomaly detection algorithms 37
3.14 Finding the responsible parameter for the detected anomaly 38

4.1 All measurements of use case 27 45

List of Tables

4.1 Detected anomalies which are caused by parameters 44
4.2 Responsible parameters which are detected by the second approach . . . 46

List of Acronyms

AOP Aspect Oriented Programming

APM Application Performance Management

ECU Electronic Control Unit

IQR Inter Quartile Range

RCP Rich Client Platform

SIP Software Integration Package

Chapter 1

Introduction

The introduction is split into three parts. At first, a motivation for the whole work is given. At the second part, the case study is described, which provides an application for data collection and the evaluation of the implemented prototype of the goal system. At the last part of this chapter, the general thesis structure is described.

1.1 Motivation

In the field of software development, performance regarding execution times plays a significant role [HHMO17]. Software applications often provide a wide area of features and configurations or process large datasets. Hence, certain performance regressions only occur on specific use cases, which are especially revealed when the application is used by customers. In order to speed up related software applications, decisive components or parameters must be found. Software is continuously improved and released in the form of new application versions [Ja13, Section 21.1.2]. Moreover, in some cases the software is modified to meet specific requirements of certain customers or to provide optional features for selling. That means performance regressions can occur with such packages regardless of the executed version. On the other hand, the different packages lead to several variants of the same software version. Regarding large software applications it is usually impossible to test all constellations of execution. Some performance regressions occur only by certain processes or under certain circumstances. Apart from the executed software, the executing hardware is relevant for application performance. These variations make it difficult to compare performance data and detect related performance regressions.

The goal of this thesis is to find and alert performance regressions of a desktop application when it is applied in productive use. Application Performance Management (APM) was

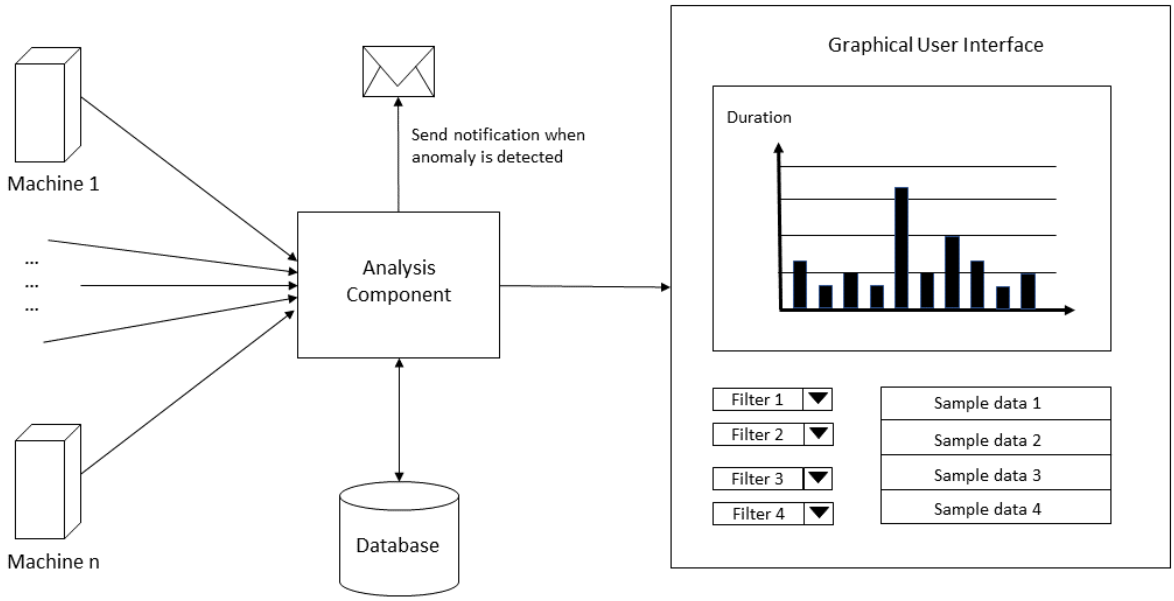


Figure 1.1: Draft of the APM System

introduced to monitor and analyze performance data at runtime. In this thesis, a system was created to detect and alert unusual performance behavior while the application is executed on several machines with several versions and several customer packages. Therefore, the application's metadata is collected and compared. An overview of this approach is given in Figure 1.1. The performance data of the machines on the left is collected and stored in the database on the bottom. The analysis component process an anomaly detection algorithm to detect performance regressions and send a notification for every anomalous measurement. The measured data is shown on a graphical user interface which is shown on the right.

1.2 The Case Study

In the automobile industry, Electronic Control Units (ECUs) and their software become more and more complex in recent time. That is why configuration tools have to process large datasets, which lead to long execution times. A configuration tool is used for configuration, validation and generation of the necessary basic software and runtime environment of the ECUs. There exists an Eclipse Rich Client Platform (RCP) application which is implemented in Java.

When the application is started, a launching window of the application is shown. Then, an existing configuration project can be opened by the selection of an existing project

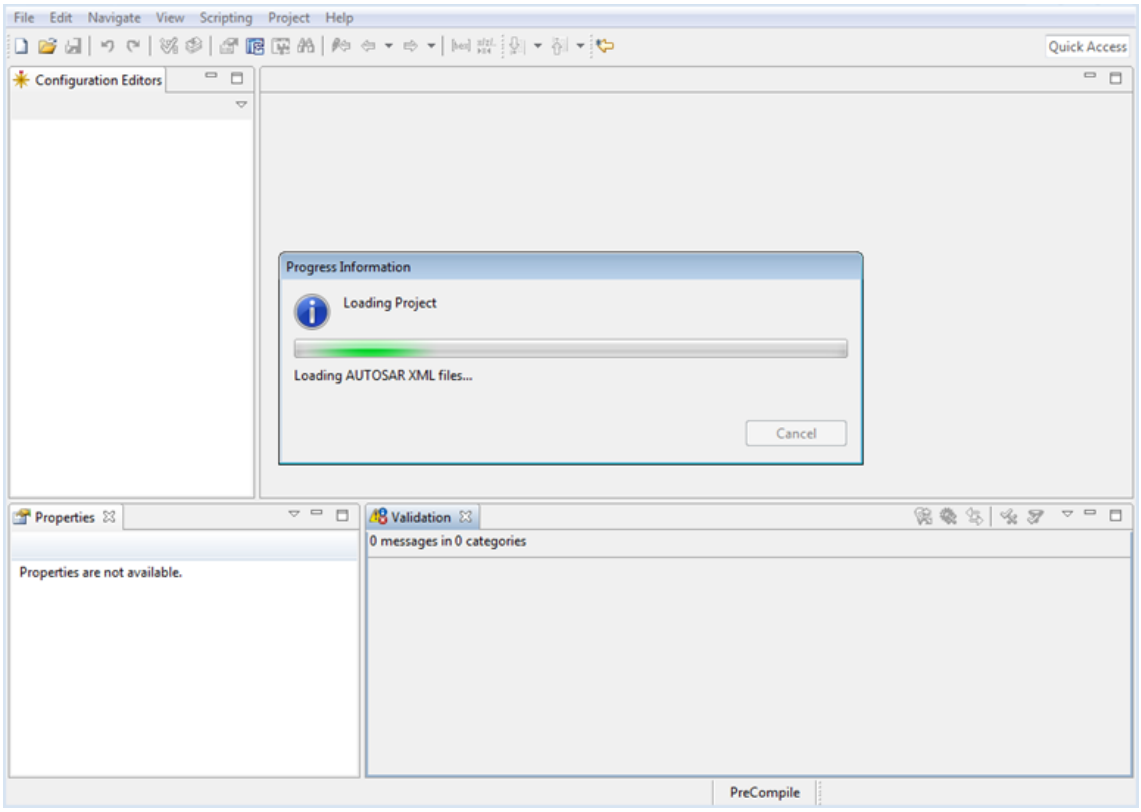


Figure 1.2: Visualization of a project load step on the graphical user interface

on the file system. When the desired project is selected, several load steps are executed. The currently processed load step is shown by the graphical user interface. An example for one load step is shown in Figure 1.2. When all load steps are successfully executed, the necessary configuration files of the existing project are available for the validation. Similar to the project load steps, several validation steps are executed to check the existing configuration for errors. The name of the recently executed validation step is shown on the right at the bottom (see Figure 1.3).

This is a very time consuming process, which must be passed before beginning work on existing configuration projects. Any kind of performance regression shall be detected and prevented if possible. These can caused by several metadata. As mentioned in Section 1.1, in some cases the software is modified to meet specific requirements of certain customers or to provide optional features for selling. These modifications are implemented by packages which are called Software Integration Packages (SIPs). Apart from that, the software is continuously improved by new application versions. The related execution times also depend on the processed ECU-project and the executing hardware. The goal of the case study is to detect and alert performance regressions and determining the related metadata. The testing of all constellations of the mentioned

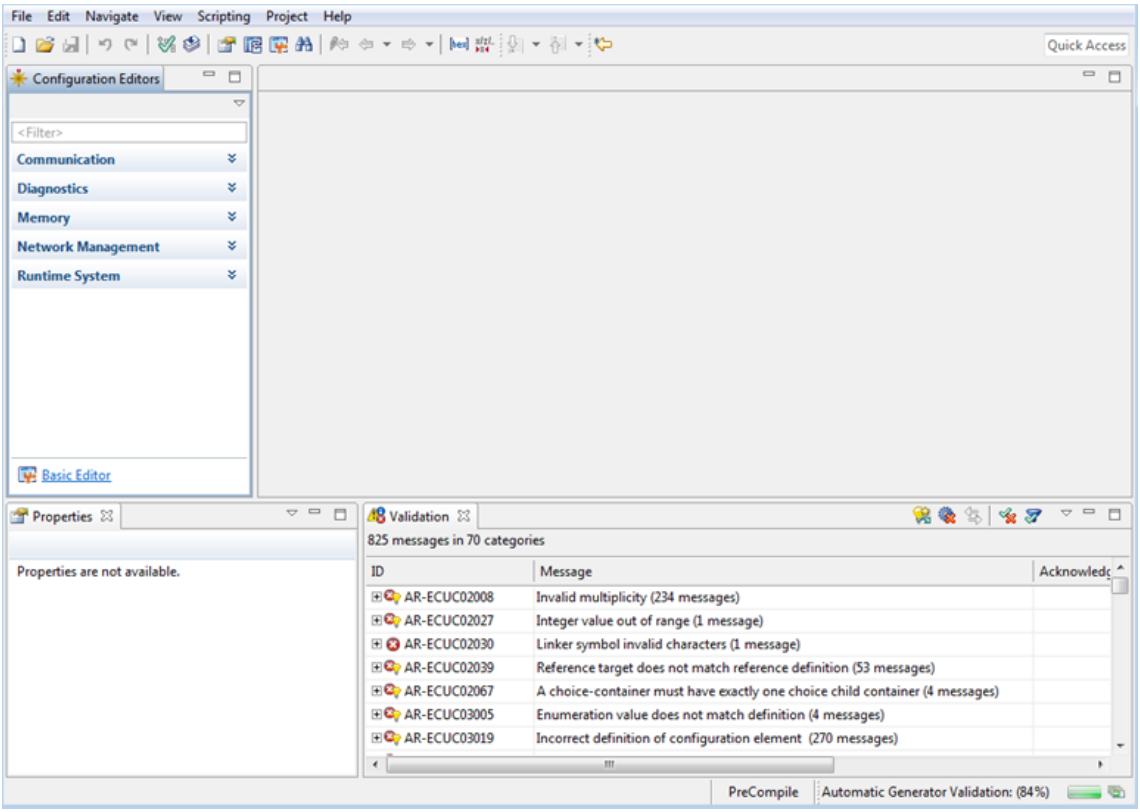


Figure 1.3: Visualization of an evaluation step on the graphical user interface

metadata is not possible. There exist a large number of projects, which is continuously increased. Hence, apart from usual regression tests which are executed before a new software version is released [ERS10], the application should be observed when it is executed in productive usage. Thus, the presented tool provides a case study for the described APM system in this thesis.

1.3 Thesis Structure

This section outlines the structure of the thesis.

Chapter 2 – Foundations: In this chapter the foundations for the thesis are described. These include the components of an APM system as well as the approaches and algorithms in the topic of anomaly detection. These foundations are needed to understand what is done and introduce the approaches, which are evaluated and used for the implementation of the final software system. The state of the art in the topic of APM is given. Therefore, three existing APM tools are introduced. Then, related work on similar challenges in this topic area is presented.

Chapter 3 – The Implementation of the APM System: This is the main part of the thesis. In this chapter the requirements for every task or component of the developed system are described. The necessary steps for data collection and the storage in the database are described. Apart from that, the provided possibilities of the implemented graphical user interface are shown. Following, the approaches which are described in Chapter 2 are compared to choose the right techniques for anomaly detection. Then, details of their implementation are presented.

Chapter 4 – Evaluation: In this chapter, the created APM system is evaluated. First, goals are formulated by research questions. Then, the used methodology and the setup of the case study are described, which were used for the measurements which were used to answer the research questions.

Chapter 5 – Conclusion and Future Work: In this section the work is summarized and additional topics for future work are described.

Chapter 2

Foundations

In this chapter the foundations for the thesis are described. These include the components of an APM system as well as the approaches and algorithms in the topic of anomaly detection. These foundations are needed to understand what is done and introduce the approaches, which are evaluated and used for the implementation of the final software system. When it comes to related work, selected works in this topic area are presented. Then, the state of the art in the topic of APM is given.

2.1 The Foundations of APM

In this work the execution of a software application is observed to improve the performance of the application. Here, the term performance is defined by the timeliness of the application's functions as applied in the paper from Woodside, Franks and Petriu [WFP07]. Other aspects like CPU utilization are not considered. In order to detect performance regressions, a system should be created which collect and store performance data and related information when the application is executed. All historical data should be shown by graphs on a graphical user interface and anomalies should be detected and alerted. These are the fields of APM. An APM tool collects, provides and analyzes performance data of an application, its services or special classes or methods. This principle is shown in Figure 2.1. In this section the foundations of APM are described by the general structure.

A collector is responsible for capturing the raw data of performance information. In order to provide time measurements of special methods when they are executed for the collector, the source code can be instrumented. The collected data is passed to the control component, which processes the performance data for storing in the database, alerting, reporting and live visualization [IBM08]. The tasks for data collection and

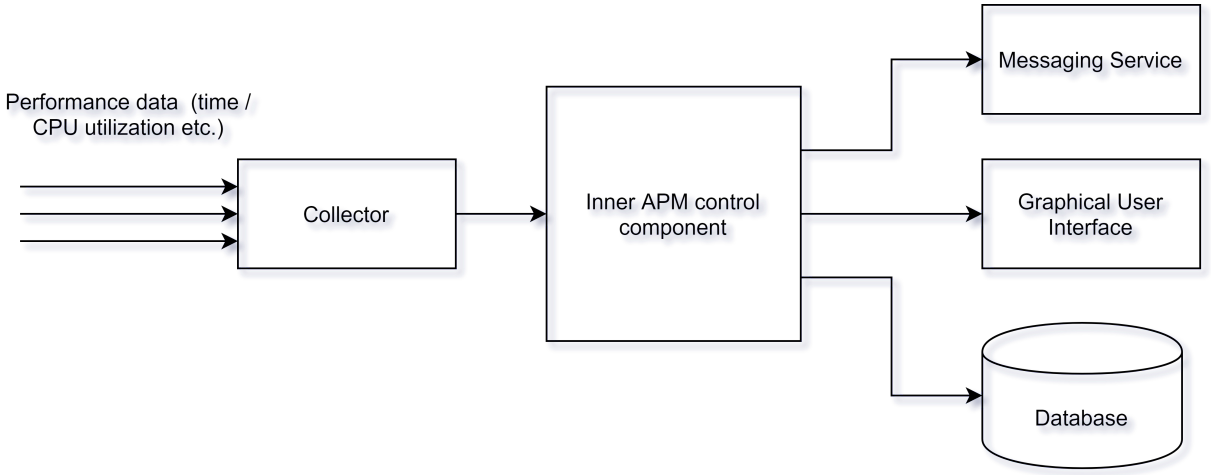


Figure 2.1: The architecture of APM

analysis are described in more detail in the following subsections. The section of data collection is divided into code instrumentation and the general approaches for the collector. Following, the several analysis approaches are described.

2.1.1 Instrumentation

In order to select the monitored sequences of the application, the program can be instrumented. The following details are described in [IBM08]. It is distinguished between source code instrumentation and byte code instrumentation. In source code instrumentation the original source code of the application is changed. Thus, the code of the application is contaminated with logic for performance management which is often not desired. On the other hand, internal parameters like attribute values are easily read. Byte code instrumentation does not require the original source code and change or create Java classes after compiling of the original Java classes of the application. This has the advantage of less performance regressions. Static byte code instrumentation creates new classes which are included in the monitored application. Dynamic byte code instrumentation changes classes at runtime during the class-loading process. The instrumented classes only exist as long as the application is executed. These are configured in an external file. A third approach called as interception does not change the original source code or runtime byte code of the target classes. Therefore, invocations are diverted through an interceptor which register the required information and pass the invocations on to the target.

In order to observe Java applications at runtime, Java virtual machines (JVMs) of the Java standard edition (Java SE) provides an interface to access monitoring data. This

interface allows to monitor the class loading system, the compilation system, the garbage collectors and memory spaces.

2.1.2 Collectors

The collector is a component of an APM system which collects all interesting performance data of the application. The following details are described in [IBM08]. There are three patterns for polling, listening and interception. In the case of polling, several measurements of the executing environment like CPU utilization of the executing environment are periodically read. This is also called sampling. When listening is applied, the collector receives data by events which send callbacks in case of execution. In this case the number of measurements depends on the individual application execution and leads to stronger loss of performance. The third approach is interception where the collector is inserted between communicating components and observes information regarding number of requests and response times.

2.1.3 Analysis

An APM tool is used for different goals. The following are described in [HHMO17]. The first is problem detection and alerting. In this case, some parts of the application are observed to detect and alert anomalies to find performance problems. This can be executed permanently to prevent performance regressions. Statistical techniques can be applied to analyze the captured performance data. The second reason to use an APM tool is the diagnosis and root cause isolation. In this case there is an existing problem an the APM tool is used to find the reason. The third goal is automatic refactoring and adaption of a running system. This approach can be used to adapt cloud solutions to current requirements by performing the related scaling tasks.

2.2 Anomaly Detection

Anomaly detection describes the problem of finding unusual values or patterns in a data set. The unusual values or patterns are called anomalies. The anomaly detection task in this work is finding performance regressions in the form of unusual execution times. In this section all foundations for the implementation of an anomaly detection algorithm are given. These include the general definition, the nature of input and output data, the types of anomalies as well as detecting techniques and existing approaches for

algorithms. These contents are based on the anomaly detection survey from Chandola, Banerjee and Kumar [VAV09].

2.2.1 The Nature of Input and Output Data

The input data of the anomaly detection algorithm is given as instances [VAV09, Ch. 2.1]. These instances can consist of only one measured value (univariate instances) or several measurements as a kind of attributes (multivariate instances). When it comes to univariate instances, the measurements can be compared by the comparison of this single value. If multivariate instances are used, the measurements includes several attributes of the same or several data types. In this case, techniques must be found to compare the measurements.

The results of an executed anomaly detection algorithm can be reported as scores or labels [VAV09, Ch. 2.4]. Scores provide values to estimate the likelihood that the checked anomaly is actually an anomaly. All measurements, anomalies as well as normal values, can be sorted in a ranking which allows a comparison of the anomalies and displays the normal and anomalous values in a smooth transition. Labeling provides a boolean value for each measurement, which shows if the checked measurement describes an anomaly or not.

2.2.2 Types of Anomalies

The anomalies in a dataset are distinguished into three different types of outliers which are decisive for the complexity of the detecting algorithm. They are presented with an example in each case.

Point Anomaly

Point anomaly detection compares every data instance with the whole dataset or statistical computations of this [VAV09, ch. 2.2.1]. The anomalies of this approach contains observations whose values are not in the region of the others. An example is illustrated in Figure 2.2. This is the simplest type of anomaly. The red colored triangle shows an anomaly when it is compared to the measurements which are shown by the blue colored points.

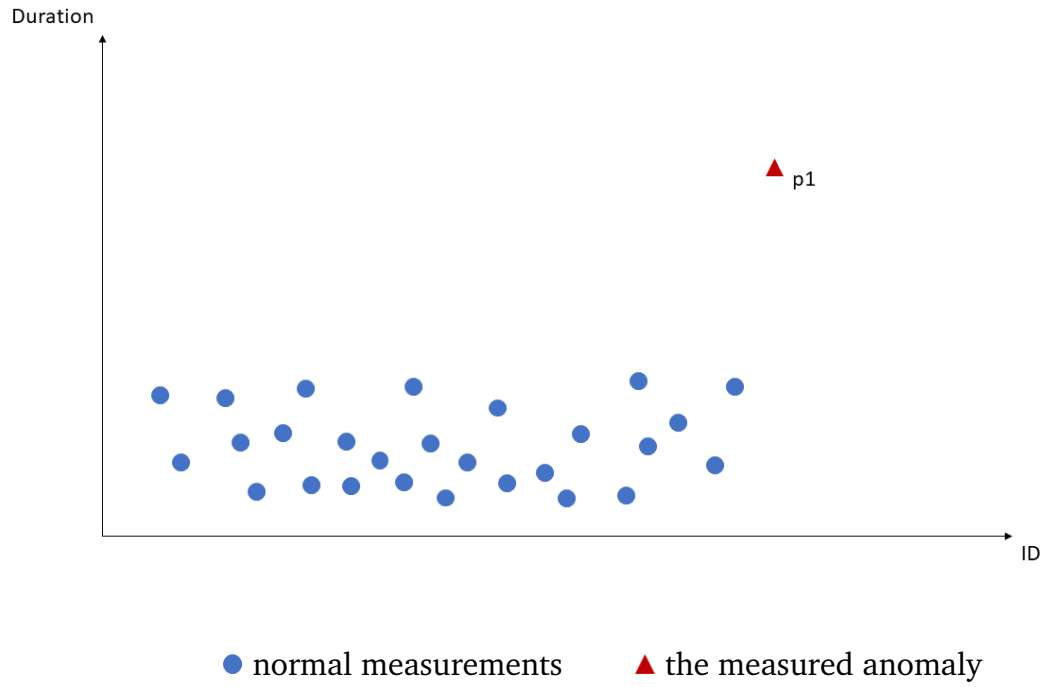


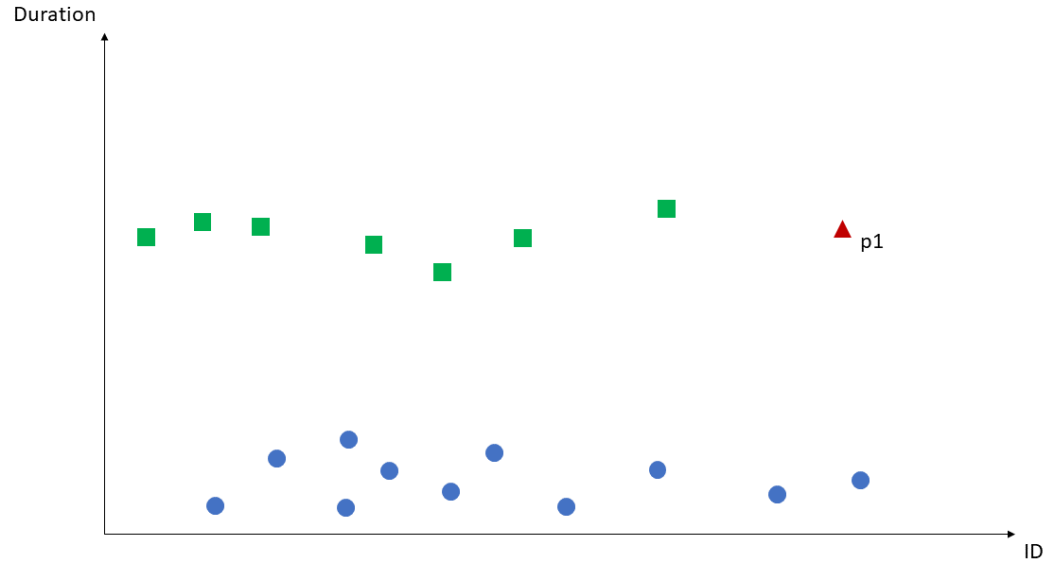
Figure 2.2: Point Anomaly Example

Contextual Anomaly

Contextual Anomalies are detected with respect to a specific context [VAV09, ch. 2.2.2]. The attributes of the observations are distinguished in contextual and behavioral attributes. The contextual attributes define the relationship between different instances or if they even should be compared regarding anomalies while the behavioral attributes are usually related to measured values. This is shown by an example in Figure 2.3. The several colors and shapes symbolize measurements of the same contextual attributes. In case of point anomaly detection, where the instance is compared to the whole dataset, the measurement p_1 is not detected as an anomaly with respect to the measurements which are shown by the green colored squares. In contextual anomaly detection, the contextual attributes are included in the detection process and p_1 is detected as an anomaly when it has the same contextual attributes like the blue colored points.

Collective Anomaly

When it comes to collective anomalies, multiple instances are anomalous together in the context of the entire data set [VAV09, ch. 2.2.3]. They are used if the collected measurements have to fit a pattern which could be given by the entire historical data set.



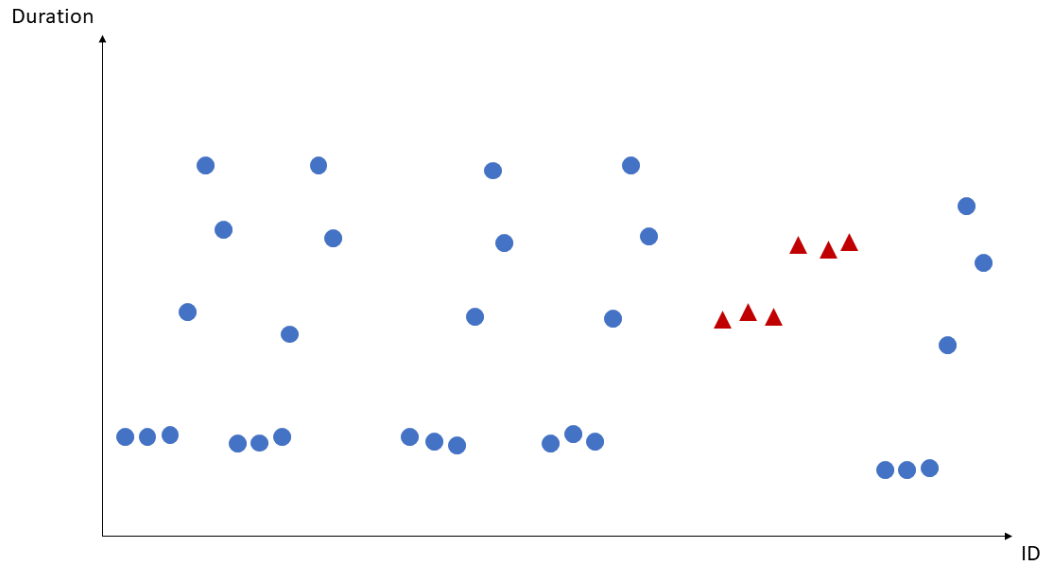
● measurements of the same contextual attribute values
■ measurements of the same contextual attribute values
▲ the anomaly detection depends on the contextual attributes of the measurement

Figure 2.3: Contextual anomaly example

Collective anomaly detection is more challenging than point and contextual anomaly detection. An example is shown in Figure 2.4. The measured values of the red colored triangles do not represent any anomalies in regard of their respective measured values. However, the red colored instances do not fit the repeating pattern, given by the other instances. Hence, they are captured as anomalies.

2.2.3 Types of Detecting / Learning

When it comes to anomaly detection it is distinguished between supervised and unsupervised anomaly detection. In supervised anomaly detection, a labeled training data set is used as learning or training data for the algorithm. In this case it is a must to define the classes of normal and anomalous instances [VAV09, ch. 2.3.1]. Unsupervised anomaly detection does not need any training data. This approach works with the assumption that anomalous instances do not occur as frequent as normal instances [VAV09, ch. 2.3.3]. Semi-supervised anomaly detection provides a middle course and requires only



● normal measurements with a repeating pattern
▲ anomalous values which do not fit the repeating pattern

Figure 2.4: Collective anomaly example

the normal instances. This approach is used if modeling anomalous instances is difficult [VAV09, ch. 2.3.2].

2.2.4 The Existing Approaches of Algorithms

Regarding anomaly detection algorithms, there exist several approaches.The details and differences are described in the following.

Straightforward and Classification-Based

In the straightforward anomaly detection approach a region for normal values is defined by experts. This is often very difficult and in future, the actual anomaly boundaries could be moved by the evolution of the application. Classification-based anomaly detection works in a supervised way. First, the classifier learns from a labeled data set to create several groups for normal and anomaly values. Then, the checked instances are assigned

to one of the created groups. [VAV09]. Both techniques requires fixed classes for normal values.

Nearest Neighbor

Nearest neighbor-based anomaly detection assumes that normal data instances occur in the same region while anomalies are widely scattered, i. e., normal values occur as nearest neighbors. If the measurements are given by multivariate instances, the distance between two data instances is defined by multiple computations of their attributes. There are two techniques to compute an anomaly score for new instances in this topic. On the one hand, techniques compute the distance of the n^{th} nearest neighbor or the distance of the n furthermost instances. On the other hand, there are techniques that count the number of instances within a certain radius to calculate the density of the neighborhood for each data instance. A low density of neighborhood suggests an anomaly.

Clustering

Clustering-based anomaly detection techniques classify data instances to group similar ones into the same cluster. This approach is primarily used without any training data. There are three techniques of clustering-based anomaly detection. In the first, the measurements are assigned to clusters by algorithms. Data instances which do not belong to any cluster are assigned as anomalies. In this case, the anomalies are detected in a labeling way. The second approach works with the distance between an instance and the closest cluster centroid. This distance determine an anomaly score. The third approach differentiates between clusters. This is useful if anomalies build own clusters. In this case it is supposed that clusters of anomalies are not as large and dense as clusters of normal data instances [VAV09]. The key difference between nearest neighbor-based and clustering-based anomaly detection is that nearest neighbor-based approaches analyze the neighborhood for every instance while clustering-based techniques compare the instances with a small number of clusters. Thus, the computational complexity of clustering-based techniques depends on the algorithms for clustering while neighbor-based complexity is mainly increased by the number of compared instances. On the other hand, clustering-based techniques was often designed for clustering and are not optimized for anomaly detection [VAV09].

Statistical Approach

Statistical techniques assume that normal data instances occur in a higher probability region of a stochastic model as anomalous ones. They are distinguished in parametric and non-parametric techniques.

- Parametric techniques require specific probability distributions which are generally given by functions. There are several probability distributions. Gaussian model-based techniques estimate the parameters with Maximum Likelihood Estimates (MLE). Therefore, the difference to the estimated mean is used for the computation of anomaly scores. The box plot rule works similar to the Gaussian model-based techniques, but there is no standard distribution required. The quartiles of the box plot are used instead of the Gaussian's estimated mean. This provides a simple alternative to the Gaussian-based approach when no standard distribution is given. The related definitions from the foundations are depicted in Figure 2.5. The maximum and minimum allowed values which define the borders for the anomaly values are calculated by the absolute difference of the identified quartiles. This difference is the Inter Quartile Range (IQR). This value is multiplied with a factor and is added to the upper quartile or subtracted from the lower quartile to get the absolute values. The proposed value of this factor is 1.5.

  Regression model-based anomaly detection is useful if the values of the examined dataset are increased by a linear trend line or fit a specific model, whereby the anomaly boundaries are modified with respect to the timestamps of measured data. Here, the influence of individual measurements to the trend line or other regression models determines their anomaly scopes. The techniques are not limited to use only one probability distribution. In some cases it can be sensible to test measured data for more than one distribution.

- An example for a non-parametric techniques is the histogram-based approach. Therefore, a histogram with several bins is built by training data, whereby a subset of the whole range is captured by this. Only normal instances can be assigned to one of the bins. Other instances are comprehended as anomalies. The width of individual bins define the boundaries between normal and anomalous values which leads to the number of found anomalies for a given data set.

Other Approaches

Information theoretic anomaly detection works with the complexity of the given data set. The complexity C for a given data set D is described by a function $C(D)$. Anomalies are captured by finding the minimal subset I which maximize $C(D) - C(D - 1)$. Spectral

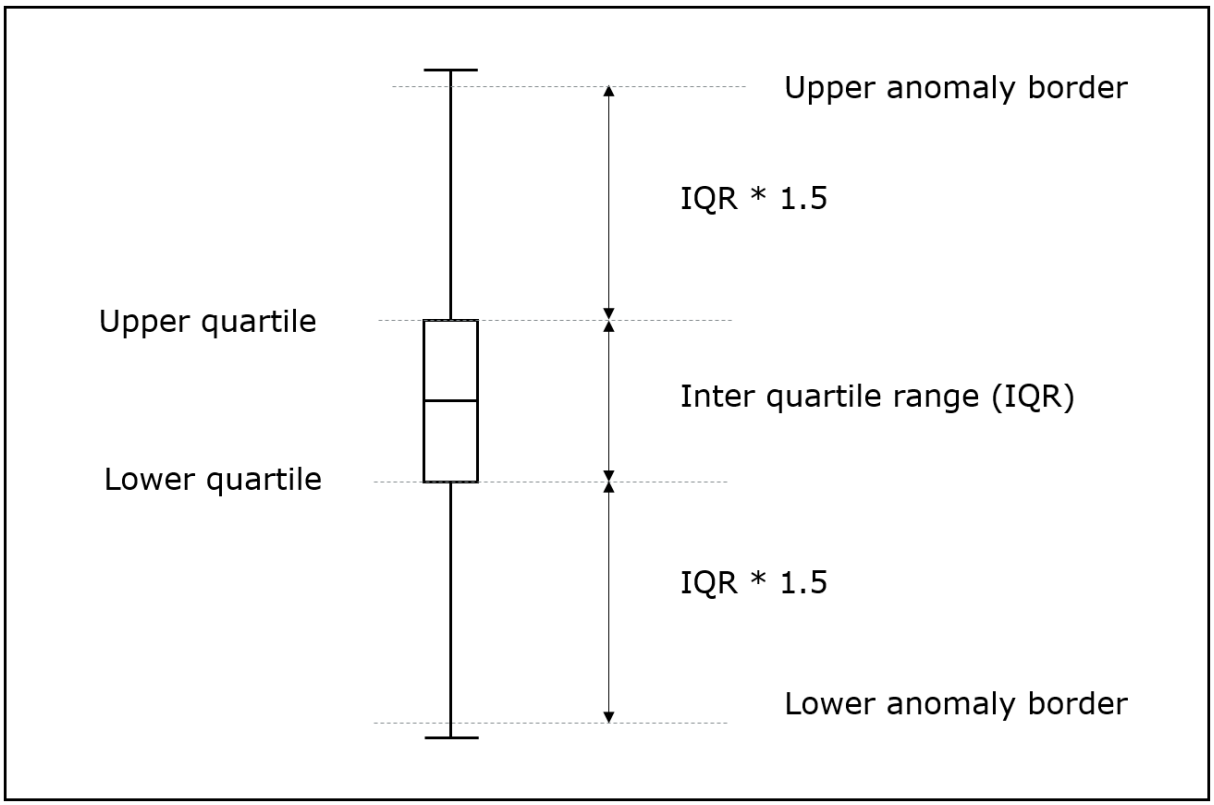


Figure 2.5: The definitions of the box plot

theoretic anomaly detection combine the instance's attributes in order to create a view on the dataset where anomalies get more different values to the normal instances [VAV09].

2.3 Related Work

Today, countless of APM tools exist [HHMO17]. Most of them are designed to observe distributed web applications which are continuously executed. Known vendors are AppDynamics, Dynatrace or New Relic. In this work, a single stand-alone application should be observed which is not permanently executed on a server. The research in the topic of APM tools is rare and there are nearly no comparing studies or surveys. Hence, the three tools Kieker [HWH12], MoSKito [Atl15d] and Dynatrace [Dyn18]'s APM tool AppMon [Dyn17a] in the field of APM are introduced in the following subsections to give an overview. Furthermore, several approaches of anomaly detection with respect to performance regressions are presented.

2.3.1 Kieker

The Kieker framework [HWH12] is distinguished in a component for monitoring and one for the analysis. The component for monitoring is responsible for instrumentation, data collection and logging of the application. The Analysis component is responsible for reading, analyzing and visualizing of the collected data. The collected data is passed with readers and writers which are responsible for serialization and deserialization. Regarding analysis, plugins can be implemented with a pipes and filters architecture [Pro17, CH. 3.3]. The architecture of the observed application can be reconstructed and visualized in sequence diagrams and dependency diagrams [Pro17, CH. 5].

In Kieker source code instrumentation can be used as well as an Aspect Oriented Programming (AOP)-based byte code instrumentation. In case of source instrumentation, time measurements are realized by explicit function calls. Thus, a code section which should be measured has to be surrounded by monitoring method calls. The AOP-based approach can be used with the AspectJ Java extension in Kieker. In this way no manual modification of the source code is necessary. If certain methods have to be observed, only their declarations are extended by annotations not their invocations. Thus, byte code instrumentation cannot be used to measure separate sequences within the methods or threads. The collected data is stored in monitoring record objects whose attributes includes the timestamps of start and end time of an execution, the name of the operation and the trace-ID which numbers the calls of one operation [RHM+08]. When source code instrumentation is used, Kieker allows to define custom monitoring record types. This makes it possible to store additional values regarding measured data. In this case the recording object is initialized in the source code of the target application. Therefore, Kieker provides the interface "IMonitoringRecord", which can be implemented to define the custom monitoring record. If an immutable type is desired, which is initialized by a constructor, the marker interface "IMonitoringRecord.Factory" can be implemented. Most of the methods must be implemented to enable serialization and deserialization. There is a getter and a setter to maintain the logging time stamp of the record object. If you do not want to implement this methods by yourself, you can additionally extend the superclass "AbstractMonitoringRecord" [Pro17, CH. 3.3]. When it comes to the transmission of the collected data, Kieker provides several reader and writer plugins. The writers, which are called by the Kieker Monitoring Controller must be set by a properties file. The SingleSocketTcpWriter can be set to send a passed record via TCP. For reading there are reader and listener interfaces which can be used. The Interface IRecordReceivedListener can be implemented to process the transmitted record objects. Regarding this, the method onRecordReceived must be implemented, whereby the received record object is passed as its method parameter. The SingleSocketRecordReader is used to receive the sent record objects. The implemented listener is passed to the constructor of the SingleSocketRecordReader when it is initialized [Pro17].

2.3.2 MoSKito

MoSKito [Atl15d] is a monitoring tool which meet the presented requirements of an APM tool, which was introduced in Section 2.1. It was designed for Java web applications. There are three MoSKito projects to cover the fields of APM: MoSKito-Essential, MoSKito-Central and MoSKito-Control. MoSKito-Essential is the core of the APM system and allows to monitor the application of a single JVM. This is the most important one for this topic. MoSKito-Essential consist of four sections: Producers, Stats, StatValues and Values, whereby every section contains or aggregates the following. The Producers are the sources of all collected data in form of source code sections. These could be methods, interfaces, threads, or whole classes. Stats represent statistics of the collected data of the producers. The StatValues could be described as a hashmap which stores the values of the stat objects with intervals as their keys. The values are the actually stored values in context of the associated interval [Atl15d]. Thresholds support the opportunity to observe a single parameter and signals when it leave the defined boundaries by a kind of traffic light system [Atl15e]. In MoSKito Central all of the collected data is saved in JSON, XML or CSV files. With configurations SQL or non-SQL options are available. Alternatively, the data can be stored in the application file server [Atl15c].

MoSKito is an interval-based APM system designed for web applications. This approach is known as polling or sampling. In this case, performance information of the system is measured at regular intervals. In order to save memory just a single value is stored for every time window at one moment.[Atl16].

Journeys show the performance data of special use cases. They illustrates the full call trees of executed methods with their parameters and return values[Atl15a]. Journeys must started and stopped explicitly by the GUI[Atl15b]. Hence Journeys cannot be used to collect performance data automatically. In order to collect performance data from producers, accumulators must be configured. [Atl16].

The application is instrumented by byte code instrumentation. Therefore, AOP is used with the open source project AspectJ. In the topic of AOP by AspectJ the classes and methods which should be monitored or should be excluded from monitoring must be tagged with annotations. In a second step an XML file have to be extended. An other approach for AOP by MoSKito allows to map the several producers on specific categories. Therefore, String values can be set in parentheses. This could be a possibility to transmit relevant metadata of the target application [Atl13].

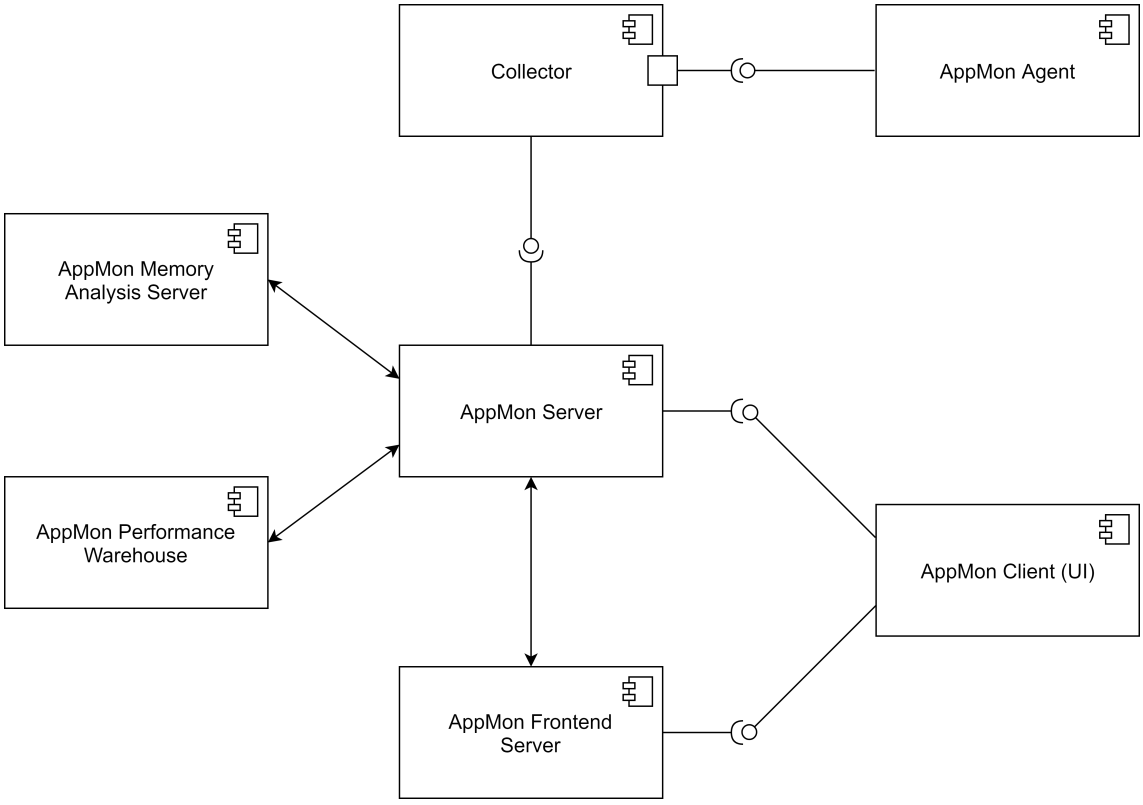


Figure 2.6: The architecture of Dynatrace AppMon

2.3.3 AppMon

The architecture of Dynatrace [Dyn18]'s APM tool AppMon [Dyn17a] consist of several components. The AppMon server is the core of the APM system. Together with the client (UI) it is responsible for configuration and administration. The performance data is captured by agents which transmits the data to the collector. Then it is sent to the AppMon Server and stored in the file system which is called as the session store. The AppMon Frontend Server is responsible for the analysis tasks. This component is executed on the same machine as the AppMon Server. The AppMon Memory Analysis Server performs memory snapshots. The AppMon collector processes, collect and pack the captured data from the agents and execute monitoring plugins which works without an agent. The AppMon Performance Warehouse stores measured time-series data. This is possible in a local database as well as in a remote database server. Call trees, thread data and memory utilization are directly stored in the session store. The AppMon Client (UI) is installed on the machine of the responsible person and represents the collected performance data visually [Dyn17b]. This architecture is shown in Figure 2.6.

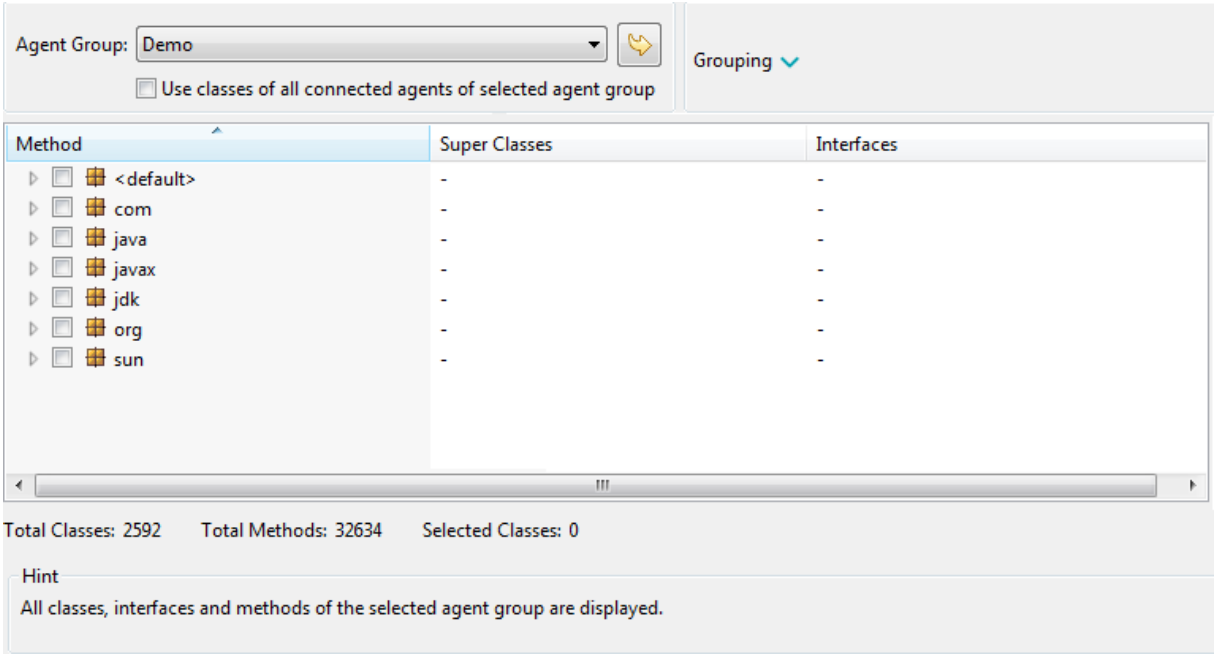


Figure 2.7: Selection of a whole package to observe with Dynatrace AppMon

AppMon supports performance analysis of AWT and swing applications. The agent is equipped with configured sensors which are able to capture the listeners, interfaces, classes and methods of the observed application. Therefore, the parts to capture are selected via check boxes. This is shown in Figure 2.8 and Figure 2.7 for or all methods of a class or a whole package. To choose a method, the classes in the view can be expanded like the classes of a package.

The timestamps and the duration of the observed method are collected as well as argument parameters and return values. Argument capturing for listener arguments like *ActionEvent e* or *ItemEvent e* provides additional information like the name of the pressed button of the application or the entered key on the keyboard. The call trees of the captured methods are completely recorded [Dyn17c] and presented in so-called Pure Paths. The Pure Paths are shown in Figure 2.9.

The configuration of monitored classes by checkboxes is very easy to use on the one hand. On the other hand, it is not possible to manipulate records in order to capture needed metadata. To distinguish several software versions, an own collector for every version could be used. However, the names of used configuration packages cannot be captured.

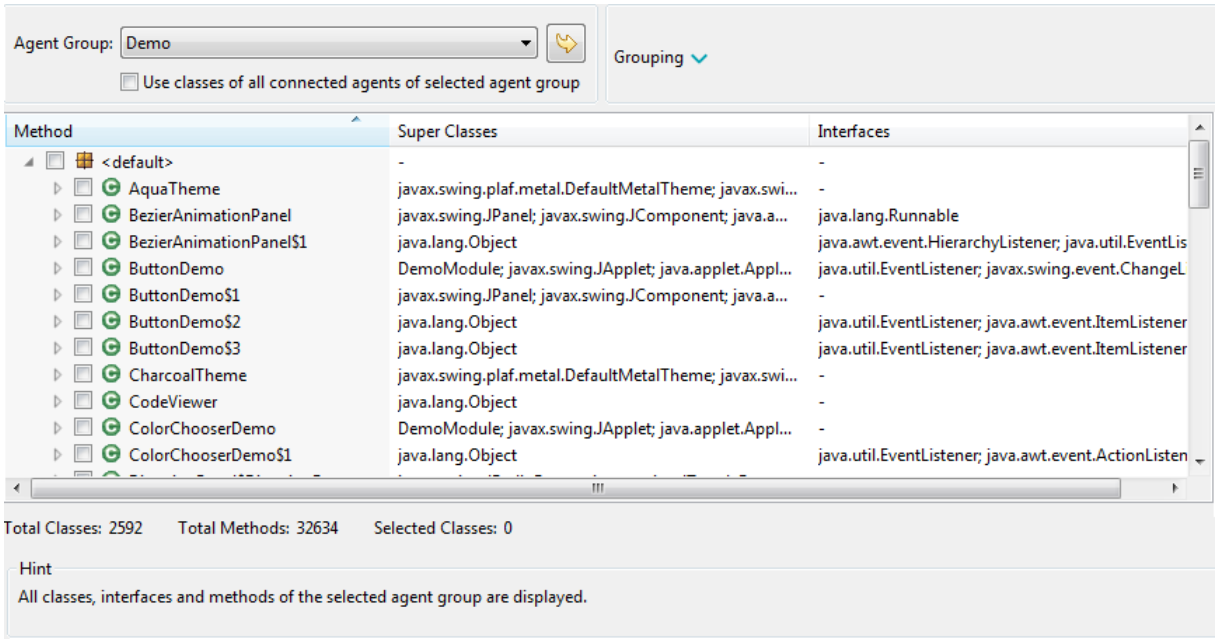


Figure 2.8: Selection of the method to observe with Dynatrace AppMon

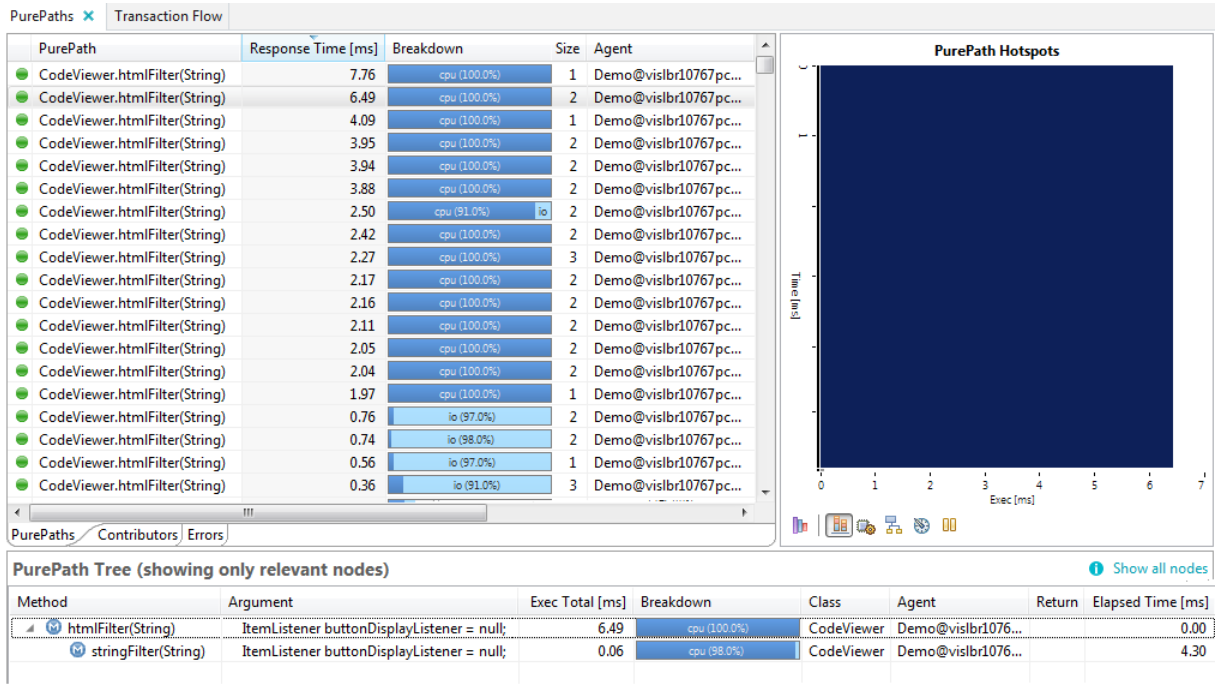


Figure 2.9: Dynatrace AppMon PurePaths

2.3.4 Related Work of Anomaly Detection

In the field of anomaly detection with respect to performance regressions in software applications, there exist a lot of related work. Two works are presented in this section.

A load-independent anomaly detection approach is described by a paper from Ghaith, Wang, Perry and Murphy [SMPJ13]. The proposed technique uses the transaction profiles of a software application. This is related to the transactions, which are executed to visit the several system resources. These includes the times of CPU and Disk I/O-tasks of the client and sever machines as well as the network transmission durations. Just the critical key transactions of the observed application are selected and applied for the analysis. The queue network model describes the waiting times in the queues until a request can actually be processed by the needed resource. The combination of the work load information, transactions profiles and the queue network model provide a model to predict resources utilization and response times. This concept shows if execution times can be improved by additional resource components. The transaction profiles for itself is just a hypothetical concept and does not represent a real working scenario. However, the comparison of the transmission profiles is used as a load-independent indicator for performance regressions. This approach fits the architecture of distributed server systems. In this work, an application is independently executed on several hardware.

Another work from Jan Waller, Nils C. Ehmke, and Wilhelm Hasselbring [WEH15] presents an approach to include performance benchmarks into continuous integration. Similar to this thesis, performance regressions should be detected by monitoring instrumentation which is integrated before the related problems occur. Therefore, a small performance overhead is desired. This was achieved by the presented Kieker framework (see Section 2.3.1). Several performance measurements were executed to compare the several software versions to detect performance regressions as early as possible. The performance regressions should have been found when several versions were compared. Hence, the next step was determining the responsible components of the application to find the responsible parts of source code. In this case, it is known that the changes of the new version are responsible for the performance regressions. In the topic of this thesis, there are other parameters like the project, which is processed by the observed application. In addition, the observed application is executed on several machines of several hardware. When the observed application is executed, these parameter are combined in various constellations. Several comparisons are necessary to check performance regressions for every parameter. When a performance regression is found, further comparisons are necessary to find the changed parameters, which led to the detected anomaly. That is why the introduced approach cannot been applied in this case.

Chapter 3

The Implementation of the APM System

This is the main part of the thesis. In this chapter the requirements for every task or component of the developed system are described. The necessary steps for data collection and the storage in the database are described. Apart from that, the provided possibilities of the implemented graphical user interface are shown. Following, the approaches which are described in Chapter 2 are compared to choose the right techniques for anomaly detection. Then, details of their implementation are presented.

3.1 Fundamental Requirements and the Application Concept

The main topic of this Bachelor thesis is to create an APM system which detects performance regressions of selected use cases in order to point out potentials of performance improvement. This draft is shown in Figure 1.1. In relation to this work the selected use cases are shown by the presented example in Section 1.2. The duration of the related methods have to be measured. Since there are several steps when the application is executed, it could be sensible to group the observed methods by custom types for load steps etc. of the case study. In this way, similar methods can be simply compared on the graphical user interface. Therefore, filters are set to compare the methods of the same types. These custom types are only used for future analysis by the related generated graphs. Hence, they are selected by the related company and are not explained in more detail. One passed method parameter can be interesting for analysis and should be captured in an optional way. As mentioned in Section 1.1, there is several metadata of the application which influences the execution time. In order to compare performance data of several versions [Ja13, Section 21.1.2] of the application, their version names are captured. Moreover, the target application is configured in a software integration

package which must be delivered as its name, too. The execution time is also influenced by the project, which is opened and processed by the observed application. The directory path to the project on the executing machine is read and stored for the related measurements. In order to detect anomalies which are caused by the executing hardware, the host name of the executing machine is captured.

The target application to observe in this work is not continuously executed on a central server. It is an application which can be executed by several users, on several hardware, every time. The performance data should be collected without any additional actions of the user. Hence, the recording functionality must be integrated into the application, i.e. the application must send the performance data on its own when it is executed.

The graphical user interface should give an overview of all measured data and the related meta information. The Provision of filters should help to select and compare specific performance data. Another component which is a part of the analysis component in Figure 1.1, should analyze incoming measurements and alert if anomalies are detected.

3.2 Instrumentation

In the field of byte code instrumentation which is used by all introduced APM tools (see Section 2.1.1), there is no possibility to assign all of the metadata of Section 3.1 to the measured execution times of the methods. Only standardized method information like passed parameters or the calling class can be captured in this way (see Section 2.3). When it comes to source code instrumentation, the custom monitoring record types of Kieker, which are introduced in Section 2.3.1 allow to save any desired values for every monitoring record. This is also true for the desired metadata.

Several of the captured metadata is the same for multiple measurements. In order to minimize the additional execution time, which is caused by the records, all of the metadata should only be read when it actually changes, i.e. several information is delivered one time and cached until it is actually modified. The host name, the version and the related software integration package do not change, while the application is executed. In order to reduce the data traffic, this information is delivered as a dataset when the application is started. The local path of the opened project is not read when the first method is executed. Hence, this information must be delivered in a separated way. Thus, there are three record types which must be implemented. Figure 3.1 shows a kind of class diagram which shows the attributes of the implemented custom monitoring record types. The type "ApmRecord", for the measured time stamps (tin, tout) and the observed method, which includes its name (method), its type (see Section 3.1), and a passed parameter (param). The type ApmRecordSession is initialized when the

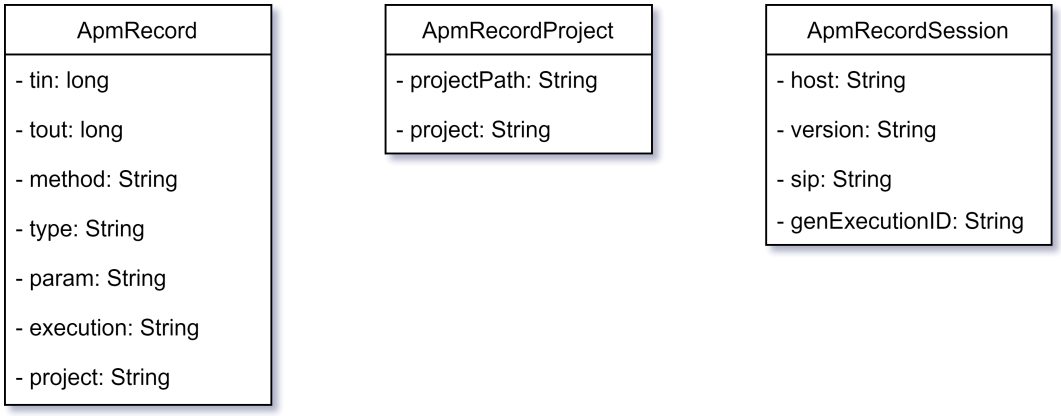


Figure 3.1: The added attributes of the implemented record types APM service

application is started and includes the host name of the executing machine (host), the version of the executed, observed application and the software integration package of the application (sip). The type ApmRecordProject saves the local path of the opened project (see Section 3.1). In order to assign the project path and the session information to the related measurements, generated unique IDs are set (ApmRecord: execution, project; ApmRecordProject: project; ApmRecordSession: getExecutionID).

Since every record should be delivered without modifications and should be set with minimum additional code, it is sensible to use the immutable approach of custom monitoring record, where the values are initialized by the constructor. The logging timestamp of the records is not needed in this thesis. Hence, its implementation of the related methods is applied by the AbstractMonitoringRecord (see Section 2.3.1).

3.3 Storage of Collected Performance Data

The captured performance data of all executing machines in the local area network must be available for the graphical user interface of the APM tool as well as for the part which executes the anomaly detection algorithms. Therefore, all historical data must be accessible with fast filtering possibilities, and calculations like average values. All of the collected data must be securely transmitted and stored permanently. Thus, the transmission via TCP and the storage in a database are sensible. The transmission via TCP is provided by implemented reader and writer plugins of the Kieker framework (see Section 2.3.1).

There is several performance data and metadata which must be stored in the database (see Section 3.1). They are temporarily stored and transmitted by the record types which are introduced in Section 3.2.

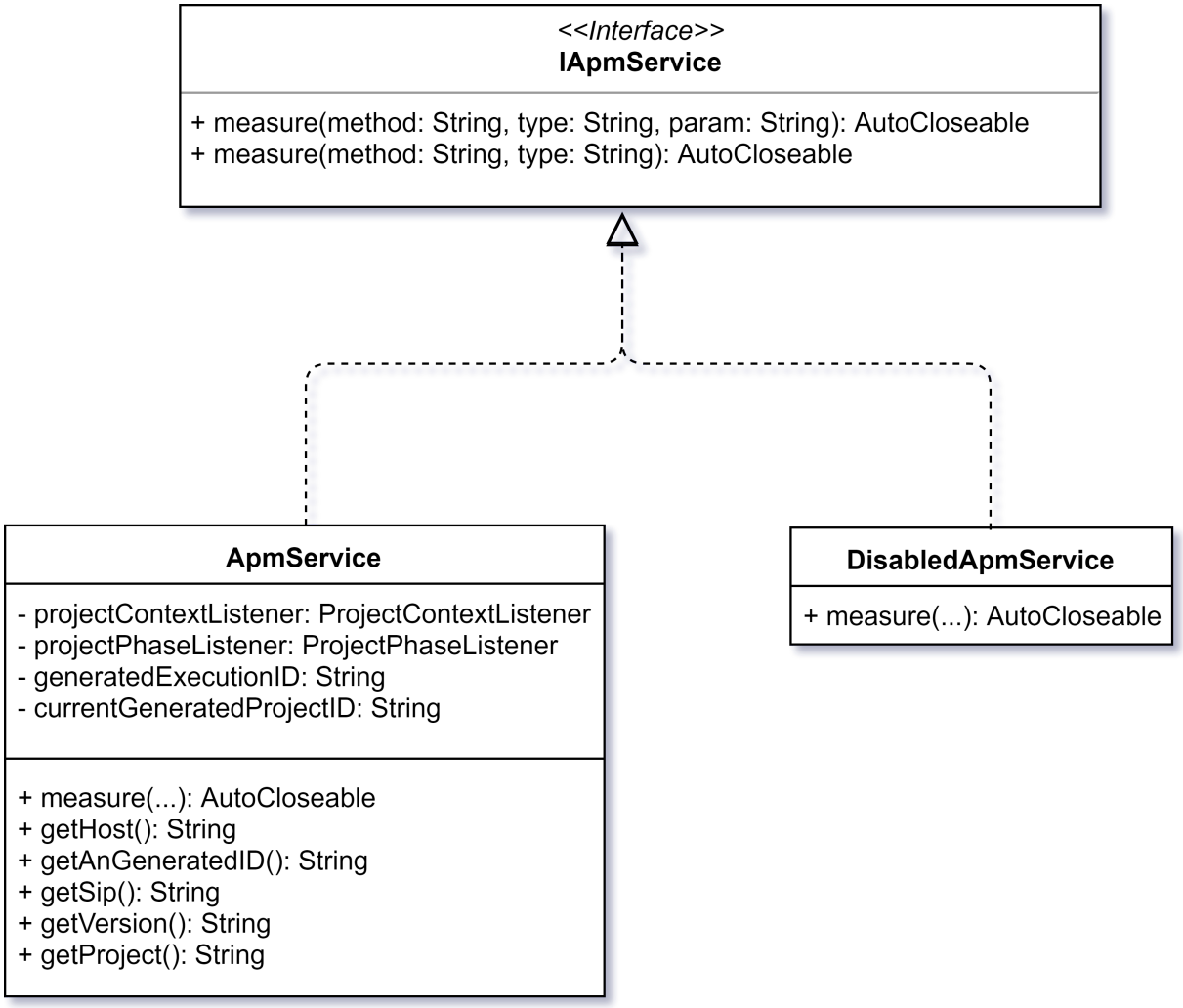


Figure 3.2: The class diagram of the APM service

The saved set of record-related information consist of the following items:

- The time stamp of the measurement
- The duration
- The measured method or use case
- The category of the method
- An optional related parameter
- The processed project
- The executing host computer

- The version of the observed application
- The related package for software configurations

When it comes to the creation of a database schema, dependencies are resolved to prevent redundancies. In order to speed up database queries, all tables are connected via unique IDs. In this case, the number of String comparisons is reduced.

The methods of the application and the variations of their attributes are limited. Hence, it is sensible to have an own table for methods which includes its name, the group of methods and the passed parameters, whereby the variations of passed values for the parameter is limited.

When the software is executed and recording of performance measurements starts some of the related metadata is not available from the beginning. Regarding the fact that all measurements of one execution will get the same meta information like the executing host or the executed software application version. In order to avoid second or more accesses on the record instances for the completion of meta information, all measurements get a unique ID for its related execution. That brings the advantage that the related metadata is stored only one time. The unique ID consists of the host name, the current time as the number of nanoseconds elapsed since the first January of 1970, 00:00 UTC and a random number. The resulting database schema is shown in Figure 3.3.

The described concept of the storage of the collected performance data is illustrated by the following example.

When the application is started, all constant metadata of the application is read and transmitted by an ApmRecordSession instance. An example is shown in Figure 3.4. Then, a new execution instance is created in the database. The attribute genExecutionID corresponds to the execution_info of the execution instance in the database. The related IDs of the host, the version and the sip are read from the database or new instances are created if they have not been created yet.

The next incoming dataset which is delivered via TCP contains the measurements and related information of a observed method. It is delivered by an instance of the ApmRecord. An example is shown in Figure 3.5. Now, the measured duration is calculated by the time stamp attributes tin and tout and a new record instance is created in the database. In the next step, a new project instance is created in the database to link the record to its session data. Since no ApmRecordProject has been delivered yet, the project instance in the database is created without the project_path attribute. This is set when the ApmRecordProject instance is delivered. An example for an ApmRecordProject instance is presented in Figure 3.6.

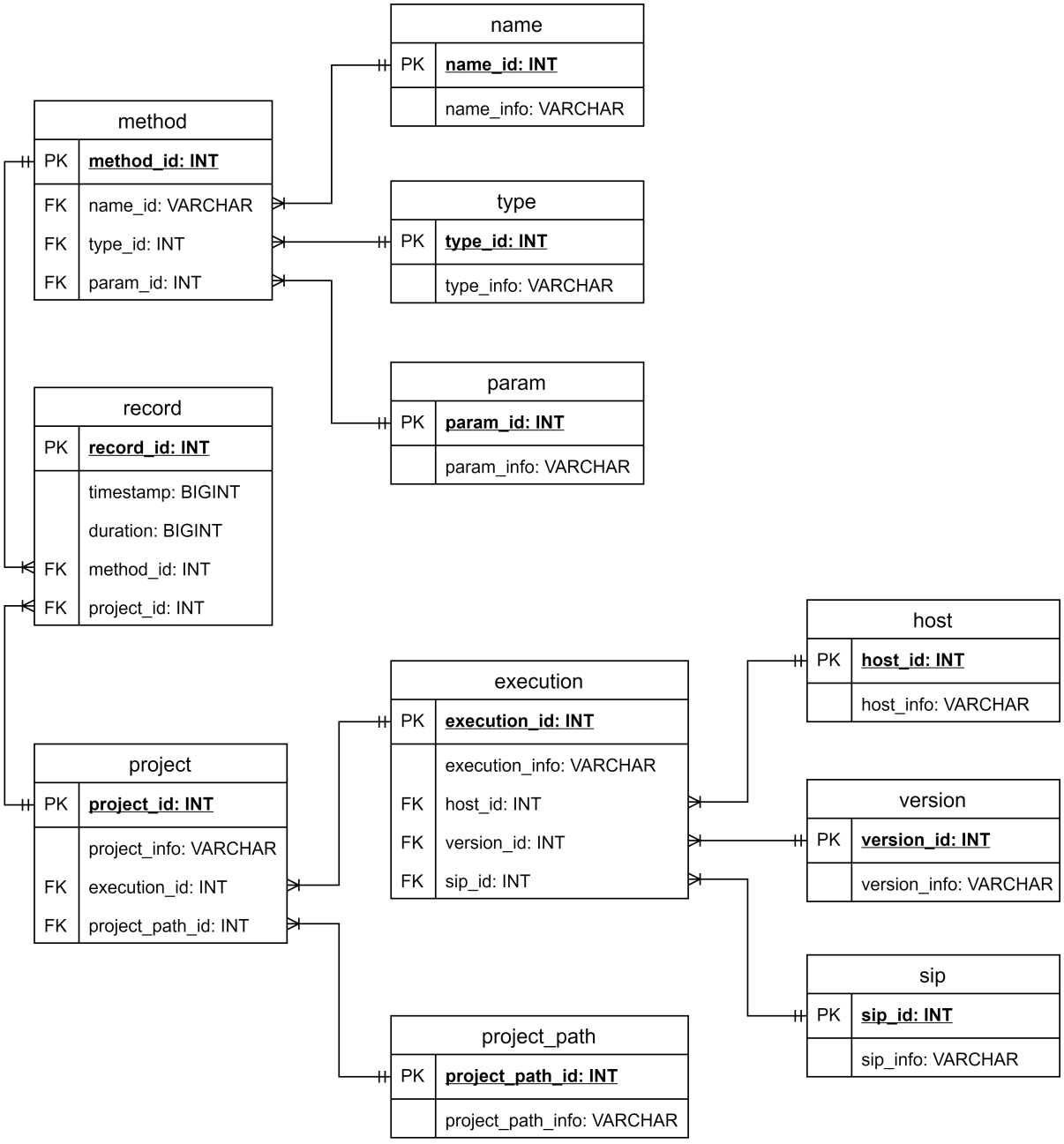


Figure 3.3: The database schema

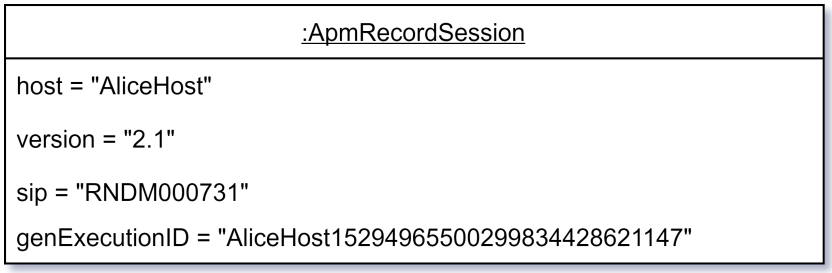


Figure 3.4: An example for an ApmRecordSession instance

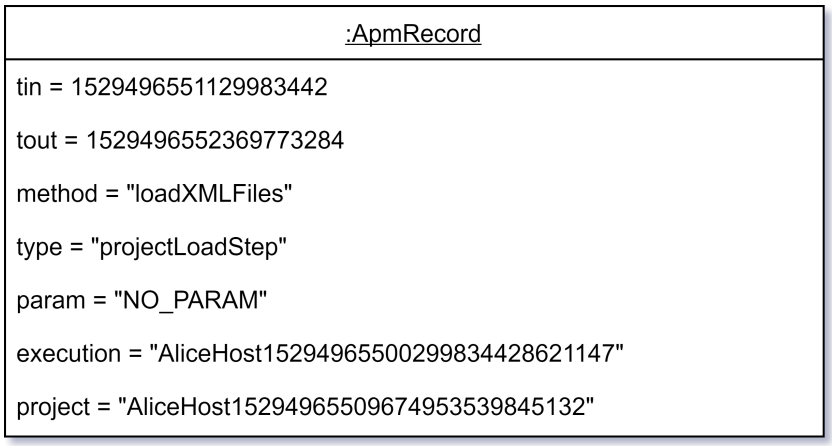


Figure 3.5: An example for an ApmRecord instance

3.4 Visual Representation

For the visual representation, a graphical user interface is implemented using Java. Therefore, JavaFX is used, since the available chart framework [Ora] fits the requirements (see Section 3.1) of the application. The measurements are chronologically list on the x-axis while the measured duration of the measurement is shown on the y-axis. The filters for the meta information are enabled by given JavaFX combo boxes.

There are two modes for the presentation of the measurements. The first, which is set at the beginning, is the filtered measurement mode. After start, all measurements are

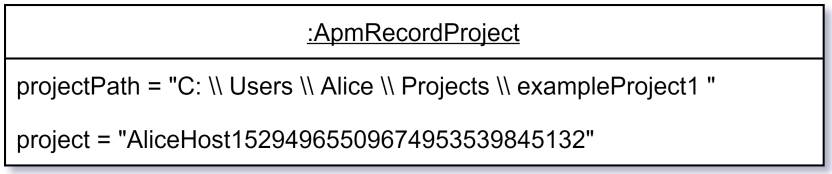


Figure 3.6: An example for an ApmRecordProject instance

shown in the diagram. An example is shown in Figure 3.7. While hovering over a single measurement point, an exact duration of the measurement is shown. Furthermore, when the point is clicked, the captured meta information is shown on the text fields below the diagram. The second is the compare mode. This allows the comparison of all measured methods with respect to the related captured meta information. This is shown in Figure 3.8. In this case, the x-axis shows methods, instead of single measurements. Therefore, the minimum, maximum and average execution time values of each method are calculated and listed on the x-axis of the diagram. The method names could not be displayed in a sensible way in the diagram. The related information appears as a tool tip in case of hovering or when clicking at the text fields below. The other three buttons of the radio button group allows to show changes of single methods related to the captured meta information. Figure 3.9 shows the measurements of a single method (named 12) related to the four captured versions which are assigned in the legend of the diagram. The additional filters can help to find the responsible factors. The same procedure is possible to compare the software integration packages or the handled projects. If more than one method is shown, the measurements of the same method are listed side by side.

3.4.1 Adaption of the Y-Axis Scaling

Large deviations of anomalies lead to problems when all measurements are shown. This situation is shown by an example of a real measurement in Figure 3.10. In this case, a logarithmic scale on the y-axis can be used to show the differences between lower measured durations in more detail. This is not possible in the recent version of JavaFX. A possibility to get a logarithmic scale is to implement an own axis type from scratch. In order to reduce the effort, the logarithmic axis is simulated by conversions of the shown values and calculations while formating the displayed numbers on the y-axis. This brings the advantage of automatic adaption to the shown measurements like maximum values and tick distances to end with a round number as well as animations at changes. Furthermore, the whole logarithmic scale can be dynamically switched without building the diagram from scratch with new axis. Therefore, a check box enables the logarithmic scale. The result is shown in Figure 3.11. This result shows another problem. If the number axis is automatically scaled, the axis usually starts by zero. In this case, the large area up to the smallest measured value is not used and small differences between the upper measurements are often not as easy to discover as it would be if the whole area of the diagram is used. Hence, the possibility to cut off the area below the minimum value which is not used for measurements is provided. This possibility is offered by the zoom scale. It is also switched by a check box. The result is shown in Figure 3.12.

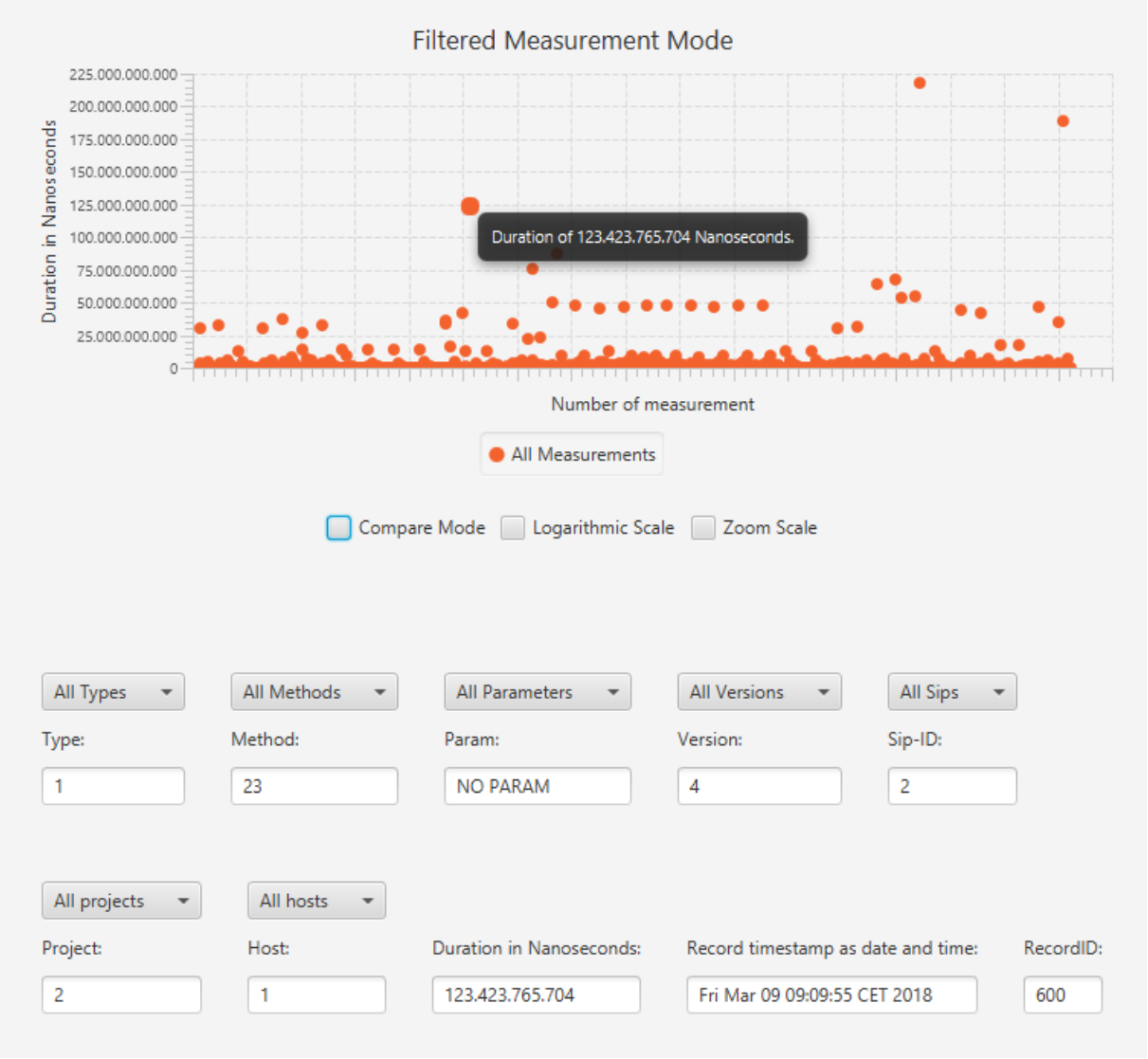


Figure 3.7: Showing details by the filtered measurement mode

3.5 Anomaly Detection

In this section the requirements of the anomaly detection algorithms are discussed and the existing approaches and algorithms are evaluated. Following details of the implementation are described.

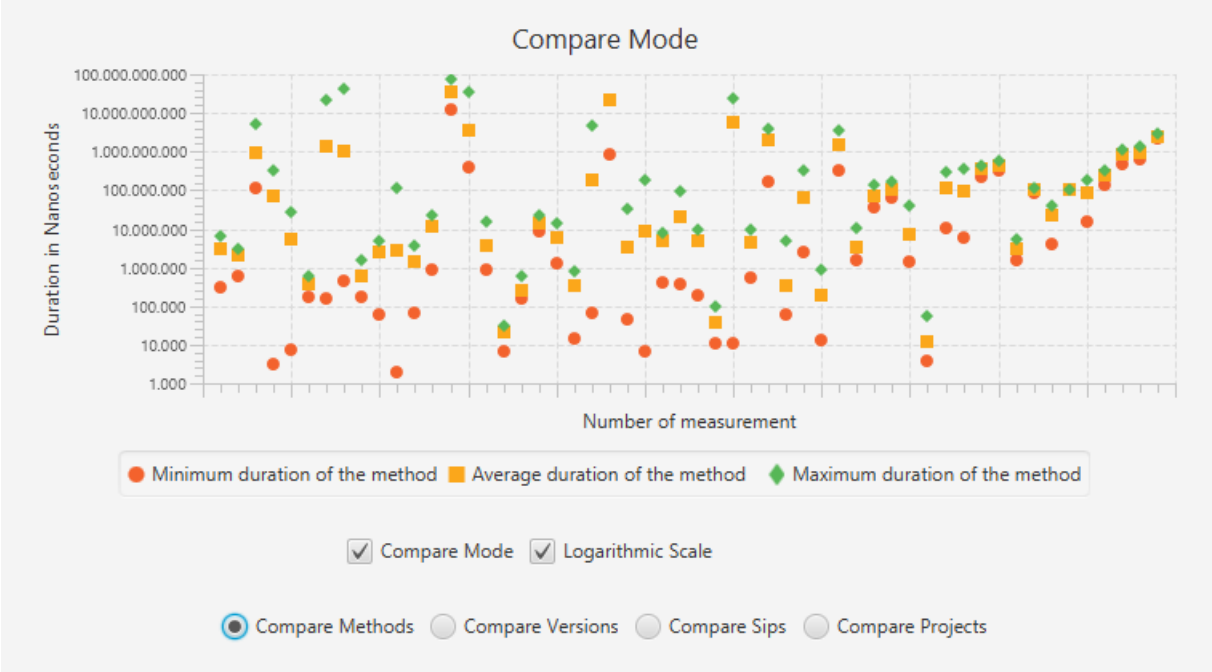


Figure 3.8: The comparison of all methods by the comparison mode

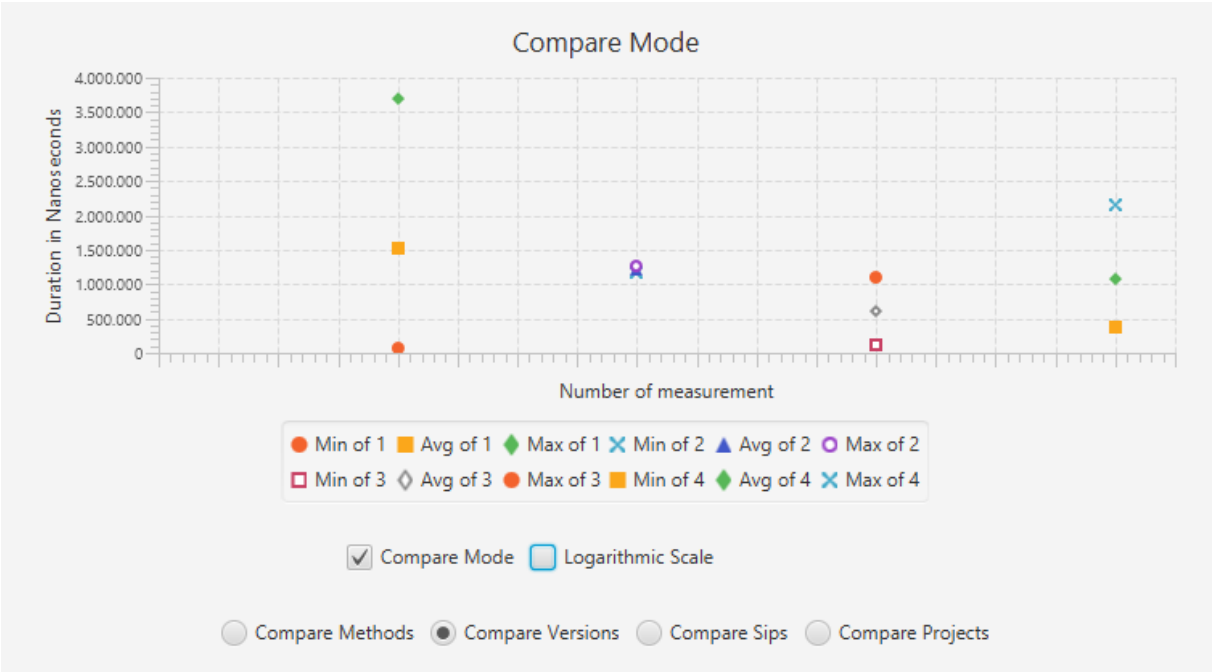


Figure 3.9: The comparison of all versions by the comparison mode

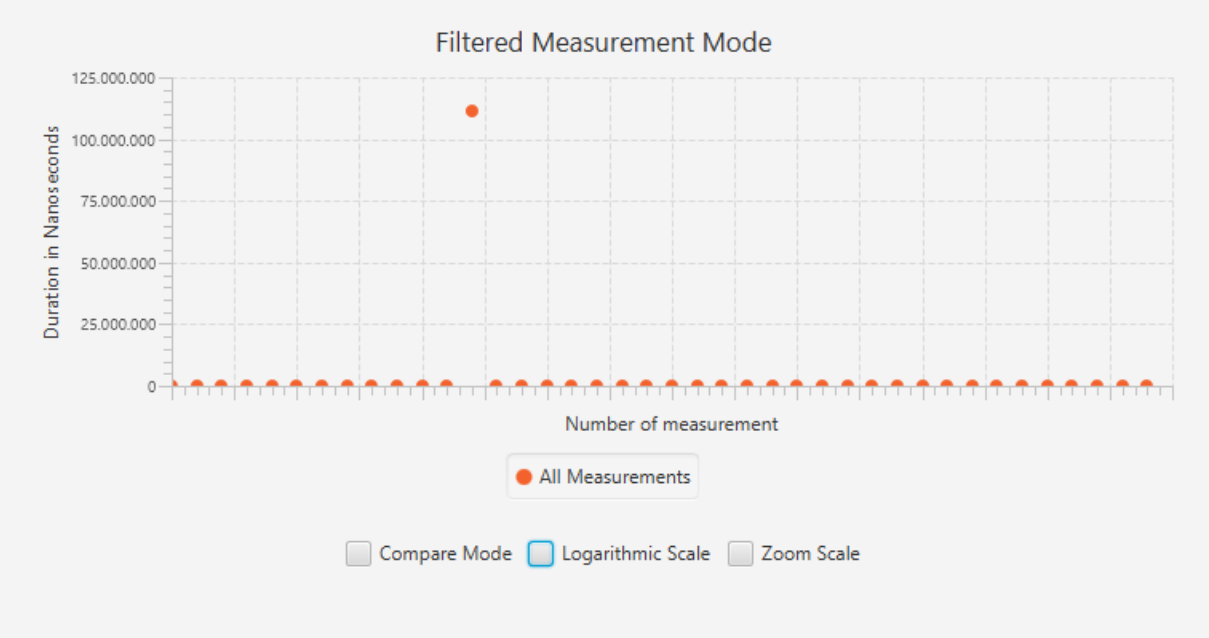


Figure 3.10: Presentation problems of the time differences caused by anomalies

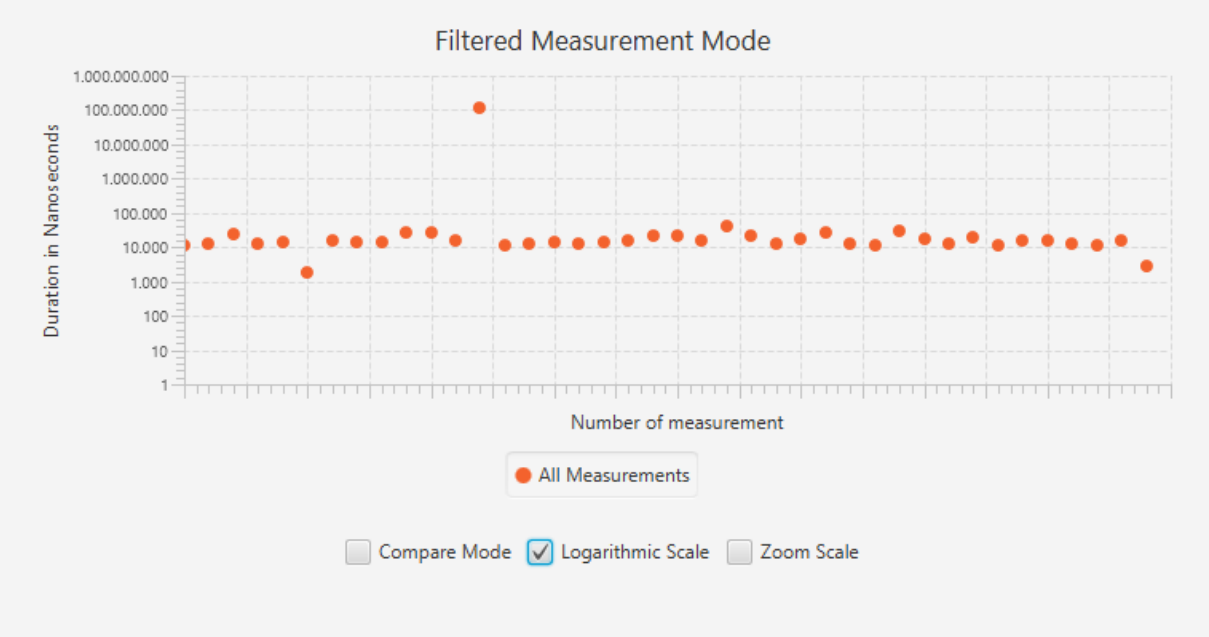


Figure 3.11: Applying the logarithmic scale

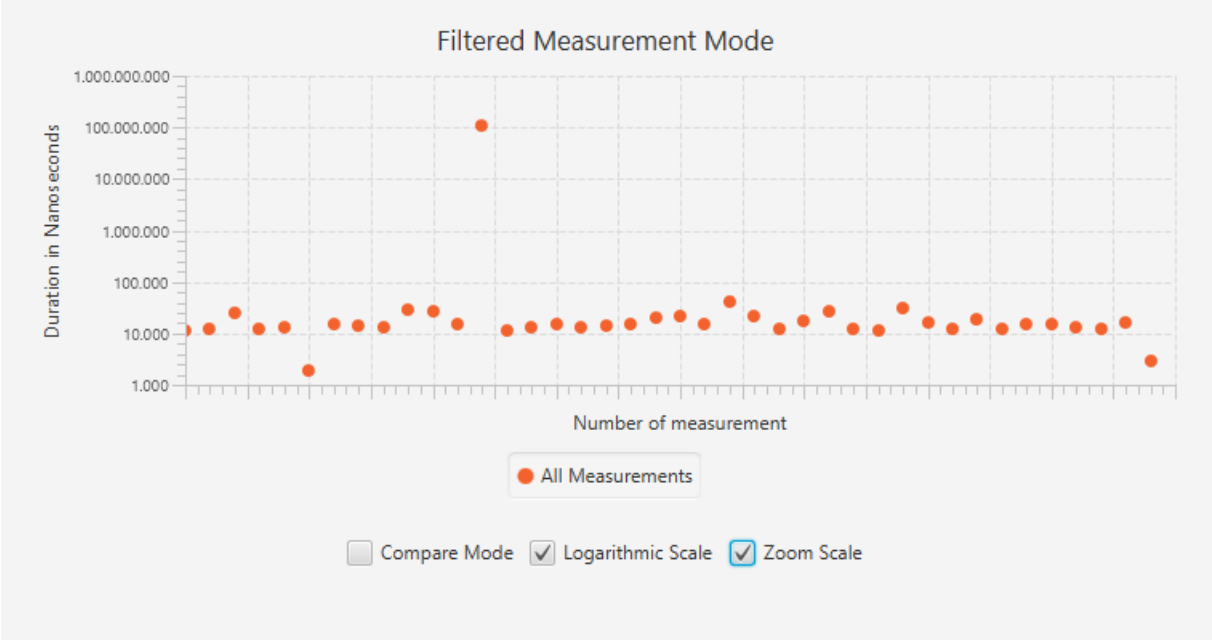


Figure 3.12: Applying the zoom scale

3.5.1 The Nature of Input and Output Data

In general, the recorded data sets of the case study in this thesis consist of several attributes which contain information of the recorded function, the duration as well as metadata like the version of the executed observed application or the processed project. All attributes are related to a specific measurement. Regarding this, the input data of this use case presents multivariate instances. On the other hand, the measured duration is the only relevant attribute with a continuous value type which provides an unlimited range of values. Thus, it is the only one which could be unique for every data instance. It follows from this that the captured attributes of the metadata could be seen as unique enumeration types which can take just on a limited number of values. These enumeration types could be used as filters for separate anomaly detection tasks which process only univariate instances.

3.5.2 The Type of Anomaly to Detect

As mentioned in Section 2.2.2, there exist the three types of anomalies: Point anomalies, contextual anomalies as well as collective anomalies. To solve the anomaly detection problem of contextual or even collective anomalies, there exist two approaches.

A model in the structure of the dataset must be found to estimate probable values for the next incoming measurements or the problem must be reduced to a point anomaly detection problem [VAV09, ch. 10].

Regarding collective anomaly detection, the iterating pattern of the measurements of the case study in this thesis could be given by single recorded executions of the observed application. Every measurement of the pattern could be related to a specific executed function. If these attributes are used as contextual attributes, the anomaly detection problem could be reduced to contextual anomaly detection. In contrast to timestamps, the observed functions and their metadata provide single, unique labels for contextual attributes. Hence, they can be used as filters, to compare measurements with similar metadata. That means all measurements of a single executed function could be used as input data for a point anomaly detection algorithm.

3.5.3 The Type of Detecting / Learning

To define a labeled training data set it is necessary to know which values are normal and where are boundaries to anomalous values. Furthermore, the training data in a supervised mode defines fixed classes which are not modified by new incoming data (see Section 2.2.3). However, caused by the evolution of new application versions, anomalous values which fire alerts in the past could be normal values in the future. In the case of fixed classes, the same problem leads to new alerts which is not desired. The values to detect are infrequent measured values which occurs in just a few types of constellations. Thus, an unsupervised anomaly detection technique which operates with unlabeled data and adjust the notifying classes by new incoming data provides the right technique to get a longterm solution. In this case, historical data must be compared to the new measured data to decide if an anomaly is measured or not.

3.5.4 The Anomaly Detection Algorithm

Caused by the described univariate point anomaly approach, there is just a single value of the checked measurement instances which must be compared to detect outliers. Hence, the complexity of the anomaly detection is reduced.

As described in Section 3.5.3, an unsupervised anomaly detection technique is the right approach for the anomaly detection task in this thesis. Straightforward and classification based algorithms work in a supervised or at least semi-supervised way (see Section 2.2.4). Hence, they were not used.

Clustering brings the advantage that anomalies, normal values, or both of them are divided into groups (see Section 2.2.4). This is sensible if the measurement instances actually build several clusters when the attributes of a single measurement instance are compared or processed by an algorithm. As described in Section 3.5.2 the compared measurement instances are just point anomalies. In the case of this thesis, outliers are only characterized by high execution times. Anomalies are detected if a certain limit value is exceeded. There are not any groups which allows longer execution times or specific constant lower values which should be detected as an anomaly or not. Hence, clustering is not the right technique for the task of this thesis.

Histogram based approaches are realized with training data to build fixed histogram bins (see Section 2.2.4). As described in Section 3.5.3 such a supervised approach is not sensible for this task of anomaly detection. Furthermore, there are not any groups which allows longer execution times or specific constant lower values which should be detected as an anomaly or not.

Parametric approaches needs a specific probability distribution function, which describes the distribution of all measurements. Regarding performance analysis of software applications, there is no Gaussian or other probability distribution function given. Thus, most of parametric techniques cannot be applied in this case. However, the box plot rule works with every distribution of measurements. Apart from that, there are no fixed anomaly classes built.

The nearest neighbor approach also meets the requirements of finding outliers far away from the most measured values. Apart from that, outliers which are measured multiple times and leads to nearly same values with same constellations of metadata, are alerted just for the first few times. This property is very useful in this use case, because it would be not sensible to receive multiple messages for the same problem.

3.5.5 The Implementation

In this subsection, the part of the application which is responsible for the anomaly detection is implemented. Therefore, two anomaly detection algorithms are implemented. The first approach uses the box plot rule to find anomalies. The second algorithm uses the nearest neighbor approach.

The anomaly detection is implemented as a separate application. The main class sets the attributes and calls the anomaly detection algorithms in a loop. In order to choose and configure the used algorithms and options like the frequency of the detecting loop the class AnomalyDetectionProperties is created. This reads a properties file when the application is stated and provides them.

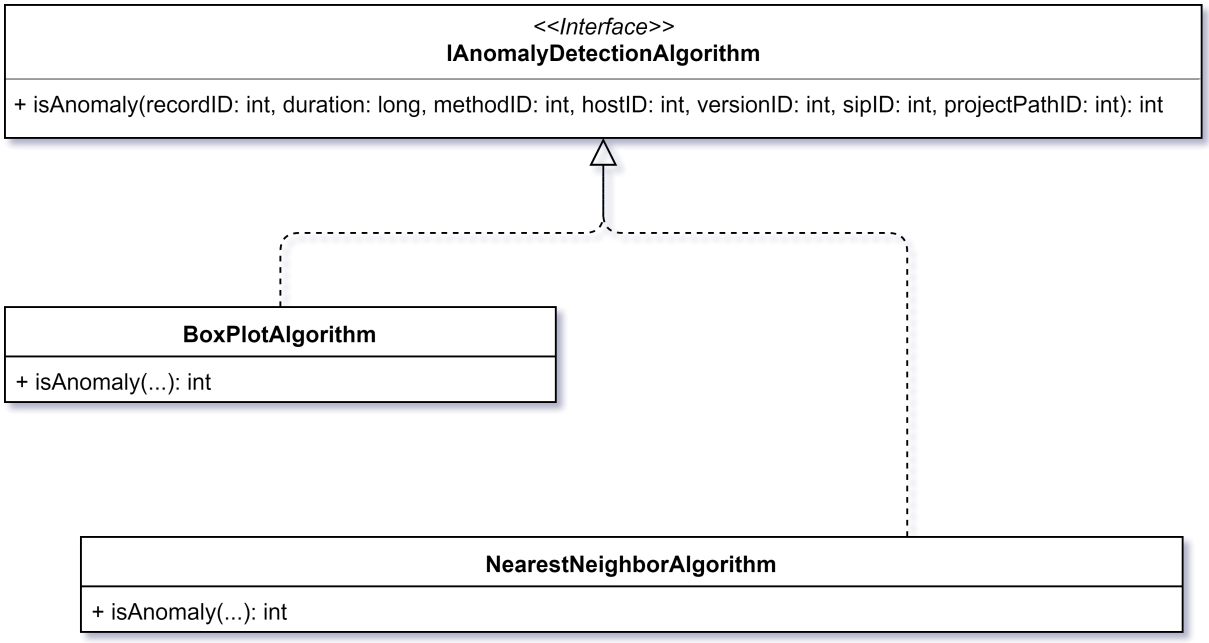


Figure 3.13: Class diagram of the anomaly detection algorithms

To exchange the used algorithms of the anomaly detection or to use multiple of them, they are implemented by the strategy pattern [Sar15]. The related class diagram is shown in Figure 3.13. As shown in the diagram, the related parameters are used as filters and are passed when the method of the interface is called. The implementation of the algorithm checks if the passed duration presents an anomaly at the resulting dataset. Another type AnomalyDetection calls the initialized algorithm to check if an anomaly occurs and to find out which parameters of the metadata are responsible. Therefore, several datasets are built by several constellations of the passed parameters. The anomaly detection algorithms return integer values. They return 1 if an anomaly is detected, -1 if the checked measurement depict no anomaly and 0 if there are not enough values at the filtered dataset to executed the algorithm.

Selection of the Datasets

As mentioned before the anomaly detection algorithms are executed for several datasets which results from the selected parameter filters. At first, the measured value is compared to all measurements of the same method. That means apart from the selected method, there are no more filters set. If the measurement is detected as an anomaly, the responsible parameter must be found.

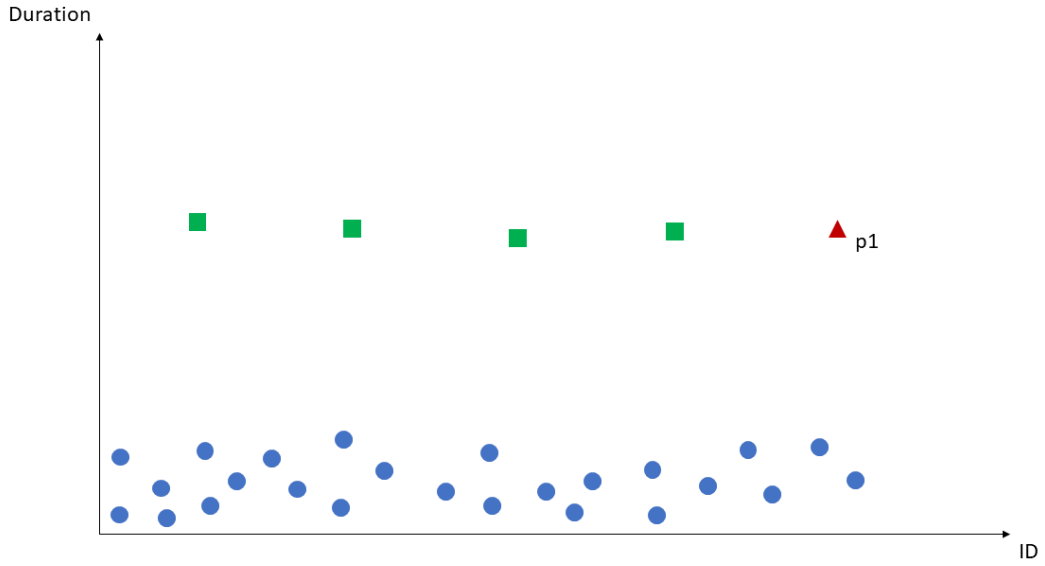


Figure 3.14: Finding the responsible parameter for the detected anomaly

If an anomaly is caused by a special parameter of the captured metadata (see Section 3.1), most of the measurements for this parameter are anomalous in the whole dataset of the checked method. On the other hand, the measured duration presents no anomaly when it is compared to other measurements of this version. Hence, if you want to find out the responsible parameter for the detected anomaly, you have to find the filtered dataset where the checked measurement instance does not depict an anomaly. This idea is shown in Figure 3.14. If the detected anomaly (the red colored measurement p1) were caused by its version, the measurements of the same version (the green colored measurements) were at the same area as the detected anomaly. In this case, the read measurement point were not an anomaly when it is compared to the green points. This approach works as long as there are enough measurements to execute an anomaly detection algorithm.

In order to identify the responsible parameter at the first occurrence, another approach was chosen. Instead of checking the measurement in the environment of a filtered dataset, the median value of the filtered dataset is compared to the whole measurements of the related method. If the median duration is also detected as an anomaly, the related parameter is detected as the responsible. This procedure is repeated for all parameters. This brings the advantage that the responsible parameter is detected even if there are no more measurements apart from p1. Other parameter values which occur at the first time in measurement p1 are also detected as responsible factors as potential false positive

results. But this is the case for both approaches. In order to avoid that random outliers distort the measurements for the anomaly detection the median value is used instead of the average duration.

If there is no anomaly found when the measurement is compared to all measurements of the same method without filters, the whole described process is executed with parameter filters. Even if a measurement seams to be a normal value when it is compared to all measurements of the same method, it could be an anomaly when it is compared to the measurements of the same project, for example. Hence, the measured duration is compared to the measurements of according parameter values.

Box Plot Rule Based Anomaly Detection Algorithm

The box plot rule based anomaly detection algorithm works with the relation of the measurement instance to statistical calculations of the given dataset (see Section 2.2.4. At the first step of the algorithm, the quartiles of the filtered dataset are read from the database. Therefore, the related numbers of the values in the sorted dataset are used to find the measurements of the inner quarters. If the filtered dataset contains less than four values, the four quartiles for the anomaly detection cannot be built. In this case, the algorithm returns 0 to signal that there are not enough values.

Nearest Neighbor Based Anomaly Detection Algorithm

The nearest neighbor based approach assumes that normal values occur in a high density, while anomalies occur scattered on a wide area. This assumption does not fit the topic of software applications in each case, however it works in many cases and for the first measurements of a group of anomalous measurements what is desired at this topic.

For the implementation of this algorithm the distances to the measurements next to the checked measurement instance are compared to the distances of the related dataset. Therefore, the average distance to a fixed number of next neighbors are calculated and compared to the average distance of 90 percent of the nearest neighbors of the whole filtered dataset. The usage of the nearest 90 percent eliminates outliers which could distort the average calculation. Similar to the factor of the inter quartile range regarding the box plot rule based algorithm, the calculated average are multiplied by a fixed value. If the average distance of the nearest neighbor distances of the checked measurement is higher, the measurement is detected as an anomaly.

Chapter 4

Evaluation

In this section, the created APM system is evaluated. Research questions are formulated and the chosen methodology to answer them is presented. Therefore, the described case study (see Section 1.2) is used for the evaluation with several experiments. The results are presented and discussed.

4.1 Evaluation Goals

The goals of the evaluation are to check if the anomaly detection of the APM system works as desired and to determine the quality of detection. Therefore, regressions regarding execution times should be detected by the anomaly detection algorithms. Furthermore, the responsible parameters of the captured metadata should be detected and shown in the output. These goals are formulated by research questions.

RQ1 Can the developed system be used to alert performance regressions?

This question is mainly related to the general anomaly detection algorithms. Therefore, it must be examined the detected anomalies. The detection of the responsible parameters are not relevant for this question.

RQ2 Can the developed system establish the cause of the performance regressions?

The evaluation shall show if the reasons or rather responsible parameters in the metadata of the detected anomalies can be determined. This is only possible, if the detected anomalies are actually caused by the captured metadata. Hence, the related anomalies must be found in a first step. These anomalies are analyzed in a second step. Thus, there are two metrics related to RQ2. The first is the ratio between the detected anomalies, which are actually caused by the parameters of the captured metadata and the anomalies which are caused by other factors, which are not captured by the

developed APM-system. The second metric is related to the relative number of anomalies, which are actually caused by the metadata. In this way, the evaluation has checked whether the captured parameters were selected wisely as well as the correctness of the implemented algorithms.

Apart from the main task of the whole system, the implemented algorithms and approaches in the part of the anomaly detection are compared in the mentioned topics.

4.1.1 Evaluation Methodology and the case study setting

In order to answer the research questions, several experiments were executed for the case study (see Section 1.2) with the implemented prototype. There were no historical performance regressions which could be used to test and evaluate the implemented APM-system. The simulation of performance regressions for the implemented system would have been a very time-consuming process, since performance regressions in this work depend on the combination of the related metadata of the measured durations, i.e., the simulation of one anomaly would have been require specific constellations of multiple variables, for every compared measurement. Hence, the evaluation is executed with real measurements of the case study. The performance data is collected and stored by the created APM system, while the application is executed 48 times with several metadata. Only the executing hardware has not been changed. The configured anomaly detection algorithms were executed for the collected performance data. The evaluation of detected anomalies would have been as complicated as its simulation. Every detected anomaly should have been compared to all measurements of the same version, the same project etc. separately for every parameter of the captured metadata. That is why the graphical user interface were used to check the results of the anomaly detection by the available filters.

In the standard use of the developed system, the anomaly detection is executed as a loop to check the latest performance data. For the evaluation, the anomaly detection was executed in a single run, whereby the performance data of the 48 observed executions was analyzed.

4.2 Description of Results

Regarding the box plot rule based algorithm, the proposed value for the IQR factor is 1.5 (see Section 2.2.4). This value let to more than 100 detected anomalies when the application was executed. This is not a realistic number regarding anomalies and is not helpful for an alerting system and the evaluation. Hence, the anomaly detection

algorithms were configured that they lead to 30 detected anomalies for each algorithm. This was a manageable number, which can be evaluated. The IQR factor was set to 48 to achieve these results. Regarding the implementation of nearest neighbor, the related factor was set to 520 to achieve the 30 detected anomalies.

Some measurements are no anomalies when they are compared to all measurements, but detected as anomalies when they are compared to the measurements with a single common parameter of the captured metadata. Since there are several parameters, one measurement can lead to several detected anomalies. Hence, the 30 detected anomalies do not mean that there were 30 several measurements detected as anomalies.

To answer research question RQ2.1, it was checked if the detected anomalies are actually caused by the captured metadata (see Section 3.3). Therefore, they were compared to the other measurements of the same method. If the other measurements with same parameters have led to the same anomalous values, the related parameters were determined as responsible. Therefore, every parameter was separately checked by the filters of the graphical user interface.

Table 4.1 shows all of the detected anomalies. Therefore, the related record-ID and the use case ID, which was stored as the method name in the database are shown on the left. The two columns in the middle shows by which algorithm the anomaly was detected. On the right, the column "caused by metadata" shows whether the detected performance regressions are actually caused by one or multiple parameters of the captured metadata (see Section 3.3). The transition from normal to anomalous values is smooth. In most cases, detected anomalies are related to unusual long execution times which could be seen as true positive detected anomalies. Sometimes there are measurements which are detected as anomalies by only one of the implemented algorithms. Moreover, the measurement with the Record-ID 605 of the use case 27 was detected as an anomaly although it is obviously not an anomaly. These events are shown by Figure 4.1. Measurement *p1* is the false positive detected measurement. It was detected as an anomaly by the nearest neighbor-based algorithm, since there are no near neighbors. The green triangles show the four measurements which were not been detected as anomalies by nearest neighbor. The three measurements of them, close to 80.000.000 nanoseconds are as close to each other that they pull down the average value for the nearest neighbor duration. Hence, they were not detected as anomalies.

Regarding the box plot based algorithm, 18 of 29 (62%) of the detected anomalies are actually caused by the captured parameters. When it comes to the nearest neighbor based algorithm, 15 of 30 (50%) of the detected anomalies are actually caused by the captured parameters.

To answer research question RQ2.2, it was checked, if the responsible parameters were found. Therefore, the anomalies of Table 4.1, which are actually caused by parameters

Identifying Numbers		Detected by algorithm		Caused by metadata
Record-ID	Use case	Box plot rule	Nearest neighbor	
99	27	✓	✓	✓
212	15	✓	×	×
296	27	✓	×	✓
344	27	✓	×	✓
392	27	✓	×	✓
398	33	✓	✓	×
435	22	×	✓	×
440	27	✓	×	✓
448	35	✓	×	×
520	23	✓	✓	×
548	15	✓	✓	×
557	24	✓	✓	✓
585	8	✓	✓	×
588	11	✓	✓	×
600	23	✓	✓	×
601	24	✓	✓	✓
605	27	×	False Positive	×
622	4	×	✓	×
642	24	✓	✓	✓
646	27	✓	✓	✓
708	7	×	✓	✓
709	8	✓	✓	✓
723	22	✓	✓	✓
761	23	✓	✓	×
877	7	×	✓	×
921	7	×	✓	×
1197	27	✓	✓	✓
1245	27	✓	✓	✓
1374	22	✓	✓	×
1395	7	×	✓	×
1447	23	✓	✓	×
1526	27	✓	✓	✓
1569	22	✓	✓	✓
1605	22	✓	✓	✓
1651	25	✓	✓	✓
1687	25	✓	✓	✓

Table 4.1: Detected anomalies which are caused by parameters

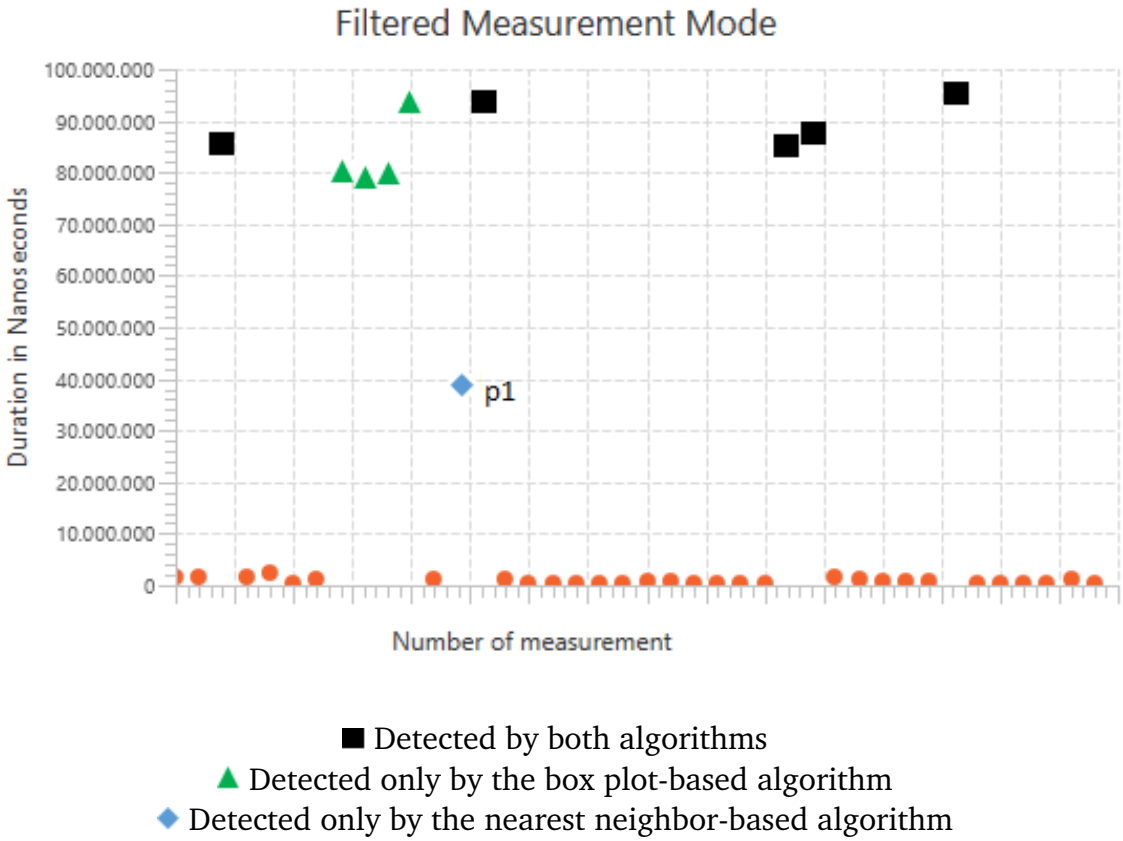


Figure 4.1: All measurements of use case 27

were evaluated. Table 4.2 shows in which cases the responsible parameters were found. The results are separately presented for the implemented algorithms and the several metadata. These includes the related host name (H), the version name (V), the SIP name (S) and the local path of the processed project (P). The responsible parameters can be detected as false negative, false positive, true negative or true positive. True negative detected results are related to parameters, which are not responsible for the performance regressions and not detected as responsible. To improve the clarity of the table, these results are presented by empty fields. The table evaluates the approach for the detection of the responsible parameter. This approach is not used when no anomaly is detected. Hence, if no anomaly was detected by the algorithm, the related fields are filled by crosses.

When the detected performance regression was caused by one or multiple parameters of the captured metadata, it was found in each case.

Identifying Numbers		Box Plot				Nearest Neighbor			
Record-ID	Use case	H	V	S	P	H	V	S	P
99	27			TP				TP	
296	27		TP		TP	×			
344	27		TP		TP	×			
392	27		TP		TP	×			
440	27		TP		TP	×			
557	24		TP				TP		
601	24		TP				TP		
642	24		TP				TP		
646	27				TP				TP
708	7	×						TP	TP
709	8			TP	TP			TP	TP
723	22			TP	TP			TP	TP
1197	27				TP				TP
1245	27				TP				TP
1526	27				TP				TP
1569	22			TP	TP			TP	TP
1605	22			TP	TP			TP	TP
1651	25			TP	TP			TP	TP
1687	25			TP	TP			TP	TP

TP = True Positive FN = False Negative FP = False Positive

× = No anomaly detected

Table 4.2: Responsible parameters which are detected by the second approach

4.3 Discussion of Results

Regarding Table 4.2 and the related research question (RQ2), it can be said that if an anomaly is caused by one of the captured parameters of the metadata, the developed approach works correctly and detect the responsible parameters. Regarding Table 4.1, nearly the half of the detected anomalies are not caused by the captured metadata. That means that there are other factors, which influence the measured performance time. The garbage collector could be a reason for these high variations.

RQ1 is concerned with the alerting property for anomalies. With respect to the evaluation and the number of detected anomalies, an alerting possibility has not yet been implemented. For practical usage, the company of the case study has to analyze the detected anomalies to adapt the parameters of the detecting algorithms (see Section 4.2) to find the desired numbers of anomalies. Regarding the anomaly detection algorithms in general, Table 4.1 shows that some anomalies are detected by only one of the algorithms. There are six anomalies which were detected by the box plot rule based algorithm, but not by the nearest neighbor based approach. When it comes to Figure 4.1 it was shown, that measurements which present average values can be detected as anomalies. This is also caused by the evaluation technique. Since all performance data was checked by a single run, some anomalous values were measured multiple times. These anomalies occurs with nearly same durations. The nearest neighbor approach detect only the first few measurements of these anomalies (see Section 2.2.4 and Section 3.5.4), i.e. the nearest neighbor approach is not sensible when there are too many values in a single anomaly detection run.

4.3.1 External validity

The external validity is related to generalizing the results of the work. The results of the conducted experiments in regard to anomaly detection depend on the observed application. To detect the relevant anomalies, the related parameters must be adapted until the requested performance regressions and if possible only these are detected. Therefore, the detected anomalies must be analyzed by experts of the observed application. The relevance of the captured metadata depends on the implementation of the observed application.

4.3.2 Internal validity

The presented APM system of this work was implemented to find performance regressions. Several metadata (see Section 3.3) was captured to compare the collected performance data and to find the responsible factors. Regarding Table 4.2 all of the responsible parameters where found. However, when two parameters of the captured metadata are determined as responsible, there are often no more measurements for both values. Hence, it is extremely likely that only one of the detected parameters is responsible. This parameter must be found in further analysis by human experts.

All of the experiment results were evaluated by the graphical user interface. That means, the responsibility of the detected parameters was not checked by further performance tests and random anomalies could led to these results.

Chapter 5

Conclusion and Future Work

In this section the whole work and its results are summarized. The following section shows additional topics for future work.

5.1 Summary

The topic of this Bachelor thesis was the creation of an APM system which detects performance regressions of selected use cases in heterogeneous environments. The challenges were the automated collection of the performance measurements and the related metadata when the application is executed as well as the anomaly detection with respect to the several metadata.

At the beginning of this work, the tasks of APM and its relation to this work were presented. Several approaches of anomalies and anomaly detection were described and compared. The collection and the transmission of the performance data could be realized by the Kieker framework. The possibilities of JavaFX provided all of the necessary features to fulfill the requirements with regard to the graphical user interface. The complexity of the anomaly detection problem could be reduced to a point anomaly detection problem, which could be solved by relatively simple algorithms.

The evaluation has shown that the implemented APM system cannot determine a responsible parameter of the captured metadata for the detected anomaly in nearly 50 percent of cases. In some cases random deviations could be induced by the several times of execution of the garbage collection. For alerting, the parameter values of the implemented anomaly detection algorithms must be adapted by human experts. Then, an automatic notification system which sends an e-mail for example must be implemented.

In summary, it can be said that performance regressions of heterogeneous environments, which are caused by specific metadata can be detected by the reduction to a point anomaly detection problem and single comparisons for the several parameters of the metadata. The quality of the detection depends on the selection of the captured metadata. Regarding algorithms, the box plot rule-based algorithm as well as the nearest neighbor-based approach can be used for this task while the second one provides a kind of learning function for similar anomalous values.

5.2 Future Work

Regarding the case study, the metadata which are collected in this work are responsible for approximately the half of the detected anomalies. In the future work, other factors like the execution times of the garbage collection could be examined to find an explanation for the other detected anomalies.

In the current state of the implemented APM system, there is no data collection of several machines at the same time possible. Additional implementations or the integration of future updates of the Kieker framework could provide the necessary properties with respect to the TCP transmission.

Appendix

Bibliography

[Atl13] Atlassian. *Working with Annotations*. 2013. URL: https://confluence.opensource.anotheria.net/display/MSK/Working+with+Annotations (cit. on p. 18).

[Atl15a] Atlassian. *Journeys*. 2015. URL: https://confluence.opensource.anotheria.net/display/MSK/5.+Journeys (cit. on p. 18).

[Atl15b] Atlassian. *MoSKito Demo, Journeys*. 2015. URL: http://burgershop-hamburg.demo.moskito.org/burgershop/moskito-inspect/mskShowJourneys (cit. on p. 18).

[Atl15c] Atlassian. *MoSKito-Central Overview*. 2015. URL: https://confluence.opensource.anotheria.net/display/MSK/MoSKito-Central+Overview (cit. on p. 18).

[Atl15d] Atlassian. *MoSKito-Essential Overview*. 2015. URL: https://confluence.opensource.anotheria.net/display/MSK/MoSKito-Essential+Overview (cit. on pp. 16, 18).

[Atl15e] Atlassian. *Thresholds*. 2015. URL: https://confluence.opensource.anotheria.net/display/MSK/6.+Thresholds (cit. on p. 18).

[Atl16] Atlassian. *Accumulators*. 2016. URL: https://confluence.opensource.anotheria.net/display/MSK/7.+Accumulators (cit. on p. 18).

[Dyn17a] Dynatrace. *AppMon Documentation*. 2017. URL: https://www.dynatrace.com/support/doc/appmon/ (cit. on pp. 16, 19).

[Dyn17b] Dynatrace. *Architecture*. 2017. URL: https://www.dynatrace.com/support/doc/appmon/appmon-reference/architecture/ (cit. on p. 19).

[Dyn17c] Dynatrace. *Diagnose AWT and swing applications*. 2017. URL: https://www.dynatrace.com/support/doc/appmon/application-monitoring/monitor-specific-applications/diagnose-awt-and-swing-applications/ (cit. on p. 20).

[Dyn18] Dynatrace. *Dynatrace Homepage*. 2018. URL: https://www.dynatrace.de/ (cit. on pp. 16, 19).

[ERS10] E. Engström, P. Runeson, M. Skoglund. *A systematic review on regression test selection techniques*. 2010 (cit. on p. 4).

[HHMO17] Heger, van Hoorn, Mann, Okanovic. "Application Performance Management: State of the Art and Challenges for the future." In: (2017) (cit. on pp. 1, 9, 16).

[HWH12] A. van Hoorn, J. Waller, W. Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis." In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE '12. New York, NY, USA: ACM, 2012, pp. 247–248 (cit. on pp. 16, 17).

[IBM08] IBM. *Run-time performance and availability monitoring for Java systems*. IBM, 2008. URL: https://www.ibm.com/developerworks/library/j-rtm1/index.html (cit. on pp. 7–9).

[Ja13] J.Ludewig, H. and. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag., 2013 (cit. on pp. 1, 23).

[Ora] Oracle. *Package javafx.scene.chart*. URL: https://docs.oracle.com/javafx/2/api/javafx/scene/chart/package-summary.html (cit. on p. 29).

[Pro17] K. Project. *Kieker User Guide*. Forschungsbericht. Oct. 2017. URL: http://eprints.uni-kiel.de/16537/ (cit. on p. 17).

[RHM+08] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, W. Hasselbring. "Kieker: Continuous Monitoring and on demand Visualization of Java Software Behavior." In: *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE'08)*. ACTA Press, Feb. 2008, pp. 80–85 (cit. on p. 17).

[Sar15] V. Sarcar. *Java Design Patterns. A Tour with 23 Gang of Four Design Patterns in Java*. Apress, 2015, p. 171 (cit. on p. 37).

[SMPJ13] G. Shadi, W. Miao, P. Philip, M. John. *Profile-based, Load-Independent Anomaly Detection and Analysis in Performance Regression Testing of Software Systems*. 2013 (cit. on p. 22).

[VAV09] C. Varun, B. Arindam, K. Vipin. *Anomaly Detection: A Survey*. Tech. rep. University of Minnesota, Sept. 2009 (cit. on pp. 10–14, 16, 35).

[WEH15] J. Waller, N. Ehmke, W. Hasselbring. *Including Performance Benchmarks into Continuous Integration to Enable DevOps*. 2015 (cit. on p. 22).

[WFP07] M. Woodside, G. Franks, D. C. Petriu. *The Future of Software Performance Engineering*. 2007 (cit. on p. 7).

All links were last followed on June 24, 2018.

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