Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master’s Thesis No. 21

Specification and Runtime Extraction of Enterprise Application Architectures for Expert-Guided Performance Problem Diagnosis

Claudio Waldvogel

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Lars Grunske
Supervisor:	Dr.-Ing. André van Hoorn
Commenced:	2015/01/14
Completed:	2015/07/16
CR-Classification:	H.3.4

Abstract

The non-functional requirements (NFRs) of enterprise application systems (EASs) have a significant impact on the Key Performance Indicators (KPIs) of companies. Among NFRs like accessibility, security, and reusability is performance considered as one of the top most important. Performance quantifies the degree to which an application meets the requirements, with respect to response times and resource utilization. To enable early performance problem detection, so-called Application Performance Management (APM) tools are integrated in an EAS life cycle.

Due to the high initial and ongoing configuration effort of APM tools, they have hardly been accepted in the industry. This results in time-consuming and error-prone manual performance problem diagnosis. These vulnerabilities of APM tools are addressed by the *diagnoseIT* research project. The main objective of the project is to enrich existing APM processes with automated configuration of instrumentations as well as automated performance problem detection and diagnosis. Since there is already a wide variety of APM tools, *diagnoseIT* does not implement a new tool to measure performance metrics. Instead, already existing APM tools provide their monitoring data to *diagnoseIT*. As part of this research project arose this work and contributed three components to the *diagnoseIT* framework.

As a basis for performance problem diagnosis, *diagnoseIT* needs to know a variety of information (e.g., system architecture, execution environment, and dynamic runtime data) about the monitored EAS. Therefore, an Enterprise Performance Model (EPM) was designed and implemented in the first part of this thesis. The second part of the work was to provide a maintenance service for the EPM and an associated integration interface for third-party APM tools. The implemented components were assembled to a prototypical implementation of the *diagnoseIT* framework. The final evaluation of the implemented solution has shown that we are able to maintain the EPM by connecting the Kieker application performance monitoring tool to *diagnoseIT*. The evaluation results of extensive load tests showed, however, that the processable amount of data is limited by the current implementation of the persistence unit.

Zusammenfassung

Die nicht-funktionalen Qualitätseigenschaften (NFRs) von Enterprise-Anwendungen (EASs) haben eine signifikante Auswirkung auf die Leistungskennzahlen von Firmen. Unter den NFRs wie Zugänglichkeit, Sicherheit und Wiederverwendbarkeit gilt die Performance als eine der allerwichtigsten. Die Performance entscheidet darüber, in welchem Ausmaß eine Anwendung in punkto Antwortzeit und Nutzung der Ressourcen den Anforderungen entspricht. Um das frühe Erkennen von Performance-Problemen zu ermöglichen, werden sogenannte Application Performance Management (APM)-Tools in den EAS-Lebenszyklus integriert.

Wegen des hohen anfänglichen, aber auch anhaltenden Pflegeaufwands von APM-Tools werden sie in der Industrie kaum eingesetzt. Das alles läuft auf eine zeitraubende und fehlerbehaftete manuelle Problemdiagnose der Performance heraus. Diese Unzulänglichkeit der APM-Tools wird durch das *diagnoseIT* -Forschungsprojekt angegangen. Das Hauptziel des Projektes ist es, schon bestehende APM-Prozesse sowohl mit automatisierter Konfiguration von Instrumentierungen anzureichern als auch mit automatisierter Entdeckung und Diagnose von Performance-Problemen. Da es bereits eine große Auswahl von APM-Tools gibt, führt *diagnoseIT* kein neues Tool ein, um die Performance-Metrik zu messen. Stattdessen stellen bereits bestehende APM-Tools ihrer Daten für *diagnoseIT* zur Verfügung. Aus diesem Forschungsprojekt heraus entstand die vorliegende Arbeit.

Das Ziel dieser Arbeit war es, drei Komponenten für *diagnoseIT* bereitzustellen. Ein sogenanntes Enterprise Performance Model (EPM) wurde definiert und implementiert, um einlaufende monitoring data zu speichern und zu verarbeiten. Um das EPM zugänglich zu machen, wurde ein System Model Maintenance Interface (SMMI) implementiert. Die Tool-Integration für bereits existierende APM-Lösungen wurde als Adapter realisiert. Alle Komponenten sind zusammengefügt als prototypische Implementierung des *diagnoseIT*-Frameworks. Die abschließende Evaluation der implementierten Lösung hat gezeigt, dass wir das EPM befüllen und auswerten können, indem wir das Kieker APM-tool mit *diagnoseIT* verbinden. Die Resultate der Evaluation von ausführlichen Belastungstests zeigten jedoch, dass die verarbeitbare Datenmenge durch die aktuelle Implementierung der Persistenzschicht beschränkt ist.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 diagnoseIT and Scope of this Thesis . . . 1
- 1.3 Document Structure . . . 3

2 Research Methodology **5**
- 2.1 G1: Enterprise Performance Model and Repository . . . 6
- 2.2 G2: System Model Maintenance Interface (SMMI) and Prototype . . . 7
- 2.3 G3: Evaluation . . . 9

3 Foundations and Technologies **11**
- 3.1 Model-Driven Software Development (MDSD) . . . 11
- 3.2 Eclipse Modeling Project (EMP) . . . 12
- 3.3 Not only SQL (NoSQL) . . . 16
- 3.4 Architecture of Enterprise Applications . . . 16
- 3.5 Architecture Modeling Languages . . . 18
- 3.6 Model Extraction . . . 19
- 3.7 Kieker Framework . . . 19

4 The Enterprise Performance Model (EPM) **23**
- 4.1 Requirements . . . 23
- 4.2 EPM Specification . . . 25
- 4.3 EPM Implementation . . . 30

5 The *diagnoseIT* Framework **35**
- 5.1 Requirements . . . 35
- 5.2 Supporting Software and Libraries . . . 37
- 5.3 Framework Implementation . . . 39
- 5.4 *diagnoseIT* Framework in Action . . . 49

6 Evaluation **53**
- 6.1 Evaluation Goals . . . 53
- 6.2 Proof-of-Concept Evaluation . . . 54
- 6.3 Lab Experiment . . . 61

7 Related Work **75**

8 Conclusions and Future Work **77**
8.1 Summary 77
8.2 Discussion 78
8.3 Future Work 79

Bibliography **81**

List of Figures

1.1 Overview of the *diagnoseIT* approach 2

2.1 Overview what parts of the *diagnoseIT* approach are covered by this thesis. . 5

3.1 Unifying Java, XML, and UML 12
3.2 Minimal subset of Ecore model 13
3.3 Connected Data Objects (CDO) architecture overview 14
3.4 CDO client architecture 15
3.5 CDO server architecture 15
3.6 Layered Architecture Example 17
3.7 Kieker framework overview 20
3.8 Kieker trace meta-model 21
3.9 Subset of Kieker meta-model to represent software systems 22

4.1 Conceptual overview of the EPM 25
4.2 Meta-classes for software modules. 26
4.3 Software module meta-model instance of the JPetStore application 27
4.4 Meta-classes for resource environments 28
4.5 Exemplary instance of a resource environment meta-model. 28
4.6 Meta-classes for allocations and exemplary instance. 29
4.7 Meta-classes for software traces. 30
4.8 Example transformation from a method call to a trace meta-model instance. 31
4.9 Ecore modeling in Eclipse integrated development environment (IDE) 32

5.1 High-level *diagnoseIT* architecture overview. 40
5.2 Core classes of the *Application* component. 41
5.3 Core classes of the CDO server and client components 43
5.4 Core classes of the model repository component 45
5.5 Core classes of the adapter component. 47
5.6 Control flow to register an *Adapter* at a *diagnoseIt* application. 50
5.7 Control flow to update the EPM model repository. 52

6.1 Technical infrastructure of the proof-of-concept evaluation 56
6.2 Kieker pipe-and-filter structure of the DiagnoseITAnalysisTool 57
6.3 Content of the *TraceRepository* within Mongo Management Studio (MMS). . . 58
6.4 Repository overview within Eclipse IDE. 59

6.5 Technical infrastructure of the lab experiment. 62
6.6 Activity diagram to visualize the functioning of the *ExperimentRunner*. 64
6.7 Kieker pipe-and-filter structure used in lab experiments 66
6.8 Example of a Box-and-Whisker Plot. 68
6.9 Response time measurements for Scenarios 1 and 2a. 68
6.10 Performance measurements for Scenario 1 70
6.11 Response time measurements for Scenario 2b,c 71
6.12 Performance measurements for Scenario 2a 71
6.13 Performance measurements for Scenario 2b 72
6.14 Performance measurements for Scenario 2c 72

List of Tables

4.1 Complementing sub meta-models of the EPM . . . 23
4.2 Enterprise Performance Model requirements . . . 24
4.3 Excerpt of Ecore model configuration properties. . . . 32
4.4 Generation Model Properties . . . 33

5.1 Extractable EPM Model Entities . . . 48
5.2 Exemplary content of the EPM in accordance with the described update process. . . . 51

6.1 EG1: Assessing the functionality of the *diagnoseIT* prototype, from a user's viewpoint. . . . 54
6.2 EG2: Assessing the scalability of the System Model Maintenance Interface (SMMI) in lab experiments. . . . 55
6.3 Technical details of the employed physical machines. . . . 57
6.4 Summary which parts of the Enterprise Performance Model are resolveable with Kieker. . . . 60
6.5 The experiment plan. . . . 67
6.6 Experiment results summary . . . 74

Listings

5.1 Example application configuration 41
5.2 Application start up 42
5.3 CDO Server Configuration 42
5.4 CDO Environment Usage 44
5.5 Excerpt of *SystemModelMaintenanceResource* 46
5.6 Example Runtime Configuration (adapter-runtime-configuration.yml) 47
5.7 Simple example how *Adapter* is used to transmit a single *Trace*. 49
5.8 Example *InaccessibleInformationResolver* implementation 50
6.1 Exemplary experiment plan. 64

List of Acronyms

ACID Atomicity, Consistency, Isolation, and Durability

ADL Architecture Description Language

ADMTF Architecture-Driven Modernization Task Force

AIM Adaptable Instrumentation and Monitoring

API Application Programming Interface

APM Application Performance Management

CBSA component-based software architecture

CDO Connected Data Objects

CI confidence interval

CIM Common Information Model

DSL Domain Specific Language

DTO Data Transfer Object

EAS enterprise application system

EG evaluation goal

EMF Eclipse Modeling Framework

EMP Eclipse Modeling Project

EPM Enterprise Performance Model

FR functional requirements

GPML General Purpose Modeling Language

GQM Goal Question Metric

HTML Hypertext Markup Language

IaM instrumentation and monitoring

IDE integrated development environment

Listings

IDM Instrumentation Description Model

IQR inter quartile range

JAR Java Archive

JAX-RS Java API for RESTful Web Services

JMS Java Message Service

JMX Java Management Extensions

JSON JavaScript Object Notation

JVM Java Virtual Machine

JVM TI JVM Tool Interface

KPI Key Performance Indicator

M2M model-2-model

M2T model-2-text

MDSD Model-Driven Software Development

MIB Management Information Base

MMS Mongo Management Studio

MOM Message-Oriented Middleware

NFR non-functional requirement

NoSQL Not only SQL

OMG Object Management Group

OSGI Open Services Gateway Initiative

POJO Plain Old Java Object

QoS Quality of Service

REST Representational State Transfer

SD standard deviation

SQL Structured Query Language

SMM Structured Metrics Meta-Model

SMMI System Model Maintenance Interface

SNMP Simple Network Management Protocol

SQL Structured Query Language

SUA system under analysis

T2M text-2-model

UML Unified Modeling Language

URI Uniform Resource Identifier

WAR Web application Archive

XMI XML Metadata Interchange

XML Extensible Markup Language

YAML YAML Ain't Markup Language

Chapter 1

Introduction

1.1 Motivation

The non-functional requirements (NFRs) of enterprise application systems (EASs) have a significant impact on the Key Performance Indicators (KPIs) of companies. Performance is considered as one of the top most important NFRs. Performance quantifies the degree to which an application meets the requirements, with respect to response times and resource utilization. Integrating Application Performance Management (APM) tools in the EAS life cycle enables early performance problem detection and troubleshooting. APM tools use continuous monitoring to detect symptoms of performance problems. In addition, the APM tools also run diagnostics to determine the cause of the problem. A crucial aspect for monitoring and diagnostics, hence the possibility to detect performance problems, is the quality of the measurement data. How often and what data is collected directly influences the quality of the measurement data [diagnoseIT, 2015].

To date, APM tools have not found wide acceptance in industry. This is due to high initial and ongoing configuration efforts of those tools. This results in manual performance diagnosis. The research project *diagnoseIT* addresses the vulnerabilities of state-of-the-art APM tools. The main objective of the project is to enrich existing APM processes with automated configuration of instrumentations as well as automated performance problem detection and diagnosis. The automation is based on formalized expert knowledge. *diagnoseIT* can control the instrumentation independently and can thus achieve a good balance between detail and overhead of the current instrumentation. Based on expert knowledge, *diagnoseIT* instruments the monitored EAS with a default configuration. During runtime the configuration is continuously refined to meet the EAS needs. Due to the expert knowledge, *diagnoseIT* is aware of well-known error patterns and is capable to perform an automated problem diagnosis [diagnoseIT, 2015].

1.2 diagnoseIT and Scope of this Thesis

The core components of the *diagnoseIT* approach are depicted in Figure 1.1. Each component is identified by a work package. Information about the monitored EAS, such as system architecture, current instrumentation, and Quality of Service (QoS) attributes, are stored in the Enterprise Performance Model (EPM) (WP1). The EPM forms the basis for the APM

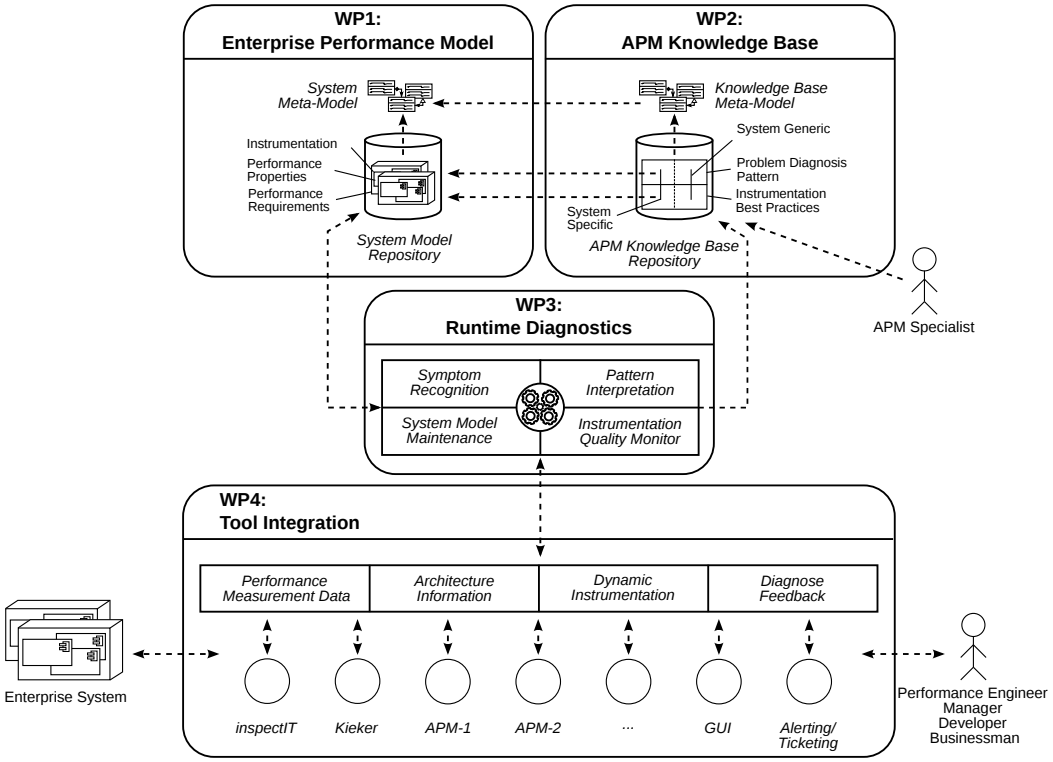


Figure 1.1. Overview of the *diagnoseIT* approach

knowledge base (WP2). System-dependent as well as independent expert knowledge is needed by *diagnoseIT* to facilitate automated instrumentation and problem diagnosis. The automated *diagnoseIT* runtime diagnostics (WP3) relies on the combination of the EPM and the APM knowledge base. *diagnoseIT* provides four runtime diagnostic components: (i) system model maintenance, (ii) symptom recognition, (iii) pattern interpretation, and (iv) instrumentation quality monitor. Independent APM tools are integrated in the *diagnoseIT* architecture via an additional APM tool integration interface. This interface also serves as entry point for data querying and modifications (e.g., querying architectural information or updating the current instrumentation).

The scope of this thesis focuses on three components of the *diganoseIT* approach. First, the EPM (WP1). The goals, including work packages, of the EPM are presented in Section 2.1. Second, the system model maintenance component of the runtime diagnostics module (WP3). In the following this component will be called System Model Maintenance Interface (SMMI). The goals of the SMMI are presented in Section 2.2. Third, an APM tool integration interface to connect existing APM tools to the *diagnoseIT* application (WP4). The relating goals are presented in Section 2.2.

1.3 Document Structure

The remainder of this document is structured as follows:

- Chapter 2 provides a detailed description of the goals, research questions, and work packages for this thesis.
- Chapter 3 comprises the foundations of this thesis.
- Chapter 4 gives a detailed overview of the developed Enterprise Performance Model (EPM).
- Chapter 5 gives a detailed overview of the implemented *diagnoseIT* framework.
- Chapter 6 comprises the evaluation of the implemented *diagnoseIT* framework.
- Chapter 7 gives an overview of related work.
- Chapter 8 summarizes the results and outlines outstanding issues for possible future work.

Chapter 2

Research Methodology

This chapter provides a detailed description of the goals (G) to be addressed by this thesis and how they fit into the context of the *diagnoseIT* project. In addition, corresponding work packages (WPs), deliverables, and used technologies are defined for each goal. The overlapping parts of this thesis and the *diagnoseIT* project are depicted in Figure 2.1.

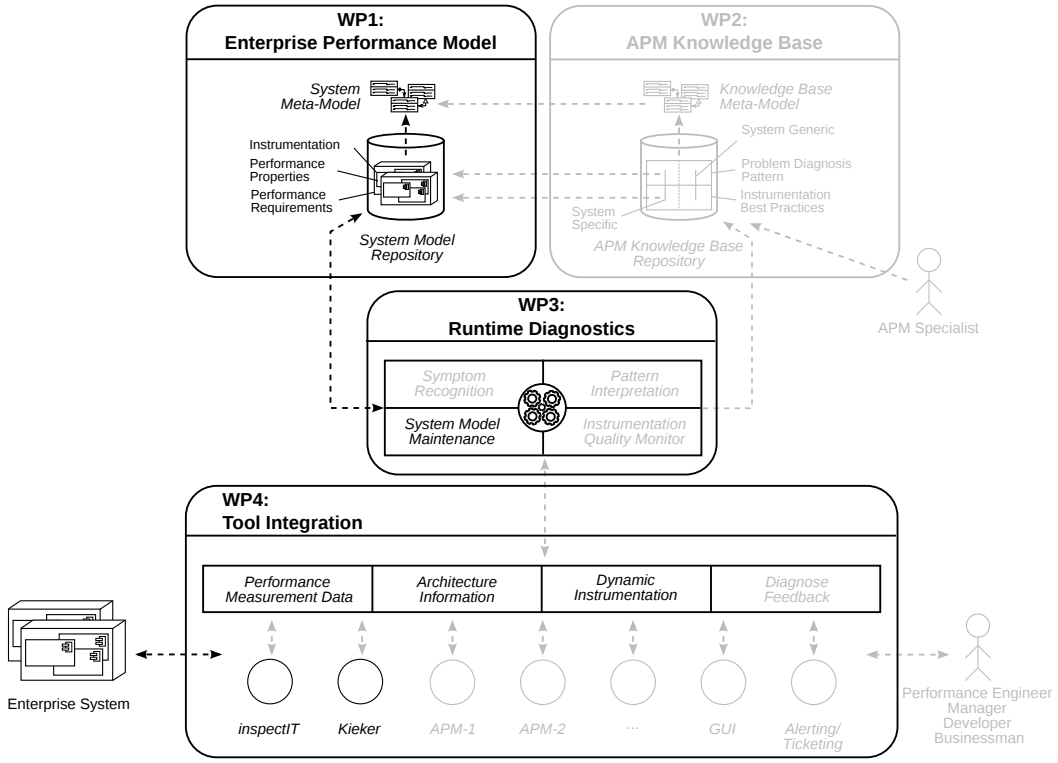


Figure 2.1. Overview what parts of the *diagnoseIT* approach are covered by this thesis.

2.1 G1: Enterprise Performance Model and Repository

The first goal of this thesis is the Enterprise Performance Model (EPM) in combination with the related model repository. As depicted in Figure 2.1, WP1 is congruent with WP1 of the *diagnoseIT* approach (see Figure 1.1).

The Enterprise Performance Model (EPM) serves to keep track of all relevant information about a monitored system. The meta-model must be able to cover information on different layers of abstraction. With respect to the monitored systems, it has to cover all relevant information regarding the architecture. More specifically, packages, classes, methods and the utilized middleware as well as information about the runtime environment like servers and executing Java Virtual Machines (JVMs). In addition, the meta-model will be able to store information about the current instrumentation of the system.

To complete the EPM, a model repository is developed in addition. The purpose of the model repository is to provide a mechanism which allows to query and to modify the underlying meta-model. The main challenge here is to decide which parts of the meta-model should be published and which parts should be protected from external access.

The first part goal is to define a well thought through meta-model for the EPM. The second is to specify a corresponding model repository for the *diagnoseIT* approach.

RQ1: Which properties have to be covered by the EPM?
RQ2: How can the different parts of the model be combined into one meta-model?

2.1.1 WP1.1: Requirements Analysis

The first and very crucial task is to determine what data can and especially must be stored in the meta-model. To ensure that the final design of the meta-model satisfies all needs for the *diagnoseIT* application, several steps can be performed. Together with the other participants of the *diagnoseIT* project, a brainstorming workshop will ensure that the final meta-model meets the expectations. As additional step, already existing application from the *diagnoseIT* participants can be examined to locate hotspots for performance problems. The deliverable of WP1.1 is a clearly defined list of properties and data which has to be covered by the EPM. The deliverable provides an answer to RQ1.

2.1.2 WP1.2: Review Existing Meta-Models and Check Applicability

After all requirements are defined, the next important task is to check if already existing solutions can be reused. Therefore existing meta-models with respect to architecture, execution traces, and JVM parameters must be searched and assessed for reuse. The assessment includes testing for re-usability, as well as testing for partial reusability. A starting point for the assessment of possible reusable meta-models are presented in Chapter 7.

2.1.3 WP1.3: Specification and Implementation of the *diagnoseIT* Meta-Model

The third subtask of WP1 utilizes the results of WP1.1 and WP1.2 to specify and implement the *diagnoseIT* meta-model. The main drivers of the final design of the meta-model are the gathered requirements. Because the meta-model has to cover different aspects of the system under analysis (SUA), e.g., architecture, execution traces, resource environments, and JVM parameters, the output artifact will be a composite meta-model which contains further meta-models. Nevertheless, all meta-models share a common goal, so they need a connection. The implementation of the meta-model will be based on EMF's meta-meta-model Ecore. The deliverables of WP1.3 are, first, a clear specification of the EPM and secondly a Java source-code artifact which contains the implementation. The deliverables provide an answer to RQ2.

2.1.4 WP1.4: Specification and Implementation of the Model Repository

This task targets the specification and implementation of the model repository. A first overview of the model repository was already given within the scope of G1 (Section 2.1). The purpose of the model repository is two-fold. On the one hand, it serves as an interface to retrieve and manipulate the data. On the other hand, the model repository is responsible to persist the data. The persistence functionality is realized by utilizing the Connected Data Objects (CDO) project of the Eclipse Modeling Project (EMP). The deliverables of WP1.4 are, first, a clear specification of the model repository and secondly a Java source-code artifact with the implementation. The deliverables provide an answer to RQ3.

2.2 G2: System Model Maintenance Interface (SMMI) and Prototype

As depicted in Figure 2.1, the SMMI is part of WP3, the prototypical implementation is the union of WP1, WP3, and WP4 of the *diagnoseIT* approach (see Figure 1.1).

The purpose of the System Model Maintenance Interface (SMMI) is to provide a clear interface which enables the maintenance of the model repository (Section 2.1). In this context maintenance means to keep the model repository in sync with the monitored system. This requires the functionality to create, update, and delete entities within the model repository, respectively within the EPM. In addition, the SMMI has to provide a functionality to query the repository model. This goal has two crucial aspects. First, the interface must be defined very carefully to ensure that neither too much nor too little information of the underlying meta-model is provided. Second, it has to be decided how the SMMI is made available and which transport protocols are used.

The second goal is divided into two parts. The first part is to define and implement the SMMI that it meets the preceding functional description. The second part is a prototypi-

cal implementation of the entire *diagnoseIT* system architecture. The architecture design includes a solution how existing Application Performance Management (APM) tools can be integrated into the *diagnoseIT* landscape and thus have access to the SMMI interface. In summary, the prototype includes (i) the EPM, (ii) the model repository, (iii) the SMMI, and (iv) an APM tool integration interface.

RQ3: What is an appropriate technology stack for the *diagnoseIT* prototype?

2.2.1 WP2.1: Requirements Analysis

The first and very crucial task is the preceding requirement analysis for all further work packages of G2. This includes requirements for the SMMI, the APM integration interface, and the overall system's architecture. To enable a proper maintenance of the EPM, it must be specified what functionality the SMMI has to provide. Since the SMMI serves as interface to the EPM, the main task is to insert, update, and delete entities within the repository. Furthermore, it will provide a functionality to query for EPM entities. At this point it is not clear if the SMMI will directly deal with entities of the EPM. To hide the internal structure of the EPM, a kind of transport object could be taken into account. Another part of the requirement analysis is to define the technologies to be used. These considerations include transport protocols, messaging strategies, middleware, web technologies, etc. A large part of the requirements can be set by a workshop with the other participants of the *diagnoseIT* project, as already proposed in WP1.1 (Section 2.1.1). The deliverables of WP2.1 are clear specifications for (i) the SMMI, (ii) the APM tool integration interface, and (iii) the *diagnoseIT* system architecture. The deliverables provide answers to RQ4 and RQ5.

2.2.2 WP2.2: Specification and Implementation of the SMMI

This task targets the specification and implementation of the SMMI. The main drivers of the final design of the SMMI are the deliverables of WP1.3 (Section 2.1.3) and WP2.1. The specification phase involves several steps. First of all, the provided operations have to be defined. Second, for each operation the input and output objects must be set. In summary, the deliverables of WP2.2 are (i) the Application Programming Interface (API), including input and output parameters, and (ii) a Java source-code artifact with the implementation.

2.2.3 WP2.3: Specification and Implementation of the APM Integration Interface

This task targets the specification and implementation of the APM integration interface. Generally one can say that this is the manner how APM tools gain access to the SMMI, thus to the EPM. For this reason, the API is mainly influenced by the results of WP2.2. Furthermore,

the chosen technologies of WP2.1 are the main aspects how the APM integration is conducted. The deliverables of WP2.3 are a clear specification of the APM integration interface and a Java source-code artifact with the implementation.

2.2.4 WP2.4: Specification and Implementation of the *diagnoseIT* Architecture

This task targets the entire system architecture of the *diagnoseIT* approach. As depicted in Figure 2.1, the approach consists of three main constituent parts. The EPM, the SMMI, and the APM tool integration. First, a connection between the implementations of the SMMI (WP2.2) and the APM integration interface (WP2.3) is established. The realization of the connection is based on the chosen technologies (WP2.1). Second, the Kieker Framework [van Hoorn et al., 2012] is adapted to conform to the APM integration interface. The deliverables of this task are the specification how the components are interconnected and how the interaction is realized as well as Java source-code artifact with the implementations.

2.3 G3: Evaluation

Evaluating the implemented solution is a crucial part of the thesis. To ensure a meaningful evaluation, the approach has to be evaluated from different viewpoints. At the beginning the proof-of-concept implementation is evaluated for applicability. Subsequently, lab experiments are carried out to examine how the system behaves in certain situations. The results collected are used to evaluate whether the implemented solution, especially the EPM and the SMMI, meets all requirements. The desired requirements are collected in preceding requirement analysis within the scope of G1 (2.1) and G2 (2.2). A further part of the evaluation is to assess the APM tool integration interface for applicability and ease of use.

RQ4: Does the approach meet all previously specified requirements?
RQ5: Is the approach applicable in real world scenarios?

2.3.1 WP3.1: Experiment Design

This purpose of this task is to determine the experiment design. In this thesis the Goal Question Metric (GQM) approach, proposed by Caldeira and Rombach [1994], will be used as evaluation mechanism . As proposed by Caldiera and Rombach, for each question (RQ4 and RQ5) one or more metrics will be defined. Furthermore, each metric is assigned an evaluation method (proof-of-concept, lab experiment) and it is specified if it is a qualitative or quantitative metric.

In addition to the GQM, it is specified how the experiments are conducted. This specification describes how different situations will be simulated, e.g, increasing the load or

simulating a distributed system. So it will be possible to measure the impact of increasing the load, to the exemplary metric M1 (Responsiveness of SMMI) and how they correlate. The deliverable of this work package is a clear specification how the experiments will be conducted and what will be measured.

2.3.2 WP3.2: Setup Experiment Environment

This package aims to set up the entire experiment infrastructure. This comprises the SUA, the *diagnoseIT* application, and the utilized APM tool. All applications have to be deployed and properly configured to ensure a controlled experiment environment. All data, concerning hardware and software configurations have to be retained. This is necessary to enable a sound argumentation as well as the possibility to repeat the experiment. The deliverable of this task is a clear description of the infrastructure, the software configuration, and the hardware specifications.

2.3.3 WP3.3: Experiments

This task target the actual experiment execution. The SUA is instrumented by an existing APM tool. The APM tool utilizes the SMMI (Section 2.1) to create a representation of the monitored system within the model repository. During the benchmark, the situations defined in WP3.1 are simulated.

While each test run, all data which is necessary for subsequent analysis is captured. The data required are those that are directly related to the metrics defined in WP3.1. The deliverable of this tasks is a dataset to be used in the final analysis.

2.3.4 WP3.4: Analyse Experiment Results

This tasks targets the final analysis of the data captured during experiments. The deliverables of this package are answers to the questions RQ4, RQ5. The analysis will draw conclusions how certain metrics are correlated to certain manipulations of the SUA. The conclusions are expressed in the form (qualitative/quantitative) which was previously assigned to this metric.

Chapter 3

Foundations and Technologies

This chapter gives an overview of the relevant technologies for this thesis. It starts with an definition and description of Model-Driven Software Development (MDSD), followed by a introduction to the Eclipse Modeling Project (EMP) in Section 3.2. It should be noted that this section is inspired by the dissertation of André van Hoorn van Hoorn [2014]. Section 3.4 outlines architectures of enterprise applications. Following this, Section 3.5 introduces the purpose of architecture modeling languages. In Section 3.6 different model extraction techniques are presented. The chapter closes with a brief introduction to the Kieker framework in Section 3.7.1

3.1 Model-Driven Software Development (MDSD)

Using models to describe complex systems has a long tradition in engineering disciplines. Models are used to describe complex problems more abstract, so they are better understandable. A model supports three features: mapping feature (an original object is mapped to the model), reduction feature (only the, depending on the current viewpoint, important properties of the original object are mapped to the model), and pragmatics feature (the model must be usable to replace the original) [Brambilla et al., 2012; Ludewig, 2003].

Brambilla et al. [2012] defines three key concepts for MDSD. The first key concept is abstraction from specific realization technologies. More specifically, this means that modeling languages are needed which do not depend on a certain technology. This contributes to better portability as well as interoperability. The second key concept is automated code generation, which means for instance the generation of API's, XML-Schemas and Plain Old Java Objects (POJOs). This concept supports productivity and efficiency. The third key concept is separating development of application and infrastructure. The concept enables the partitioned reusability of application-code and infrastructure-code. Definition 1 refers to the definition of software in the context of MDSD stated by Brambilla et al. [2012].

Definition 1 ***Model + Transformation = Software***

The equation requires two technologies. First, modeling languages and secondly model transformation technologies. A modeling language covers definitions of abstract syntax, concrete syntax, and semantics. While the abstract syntax defines the set of modeling elements and how those can be combined, the concrete syntax describes how it is actually

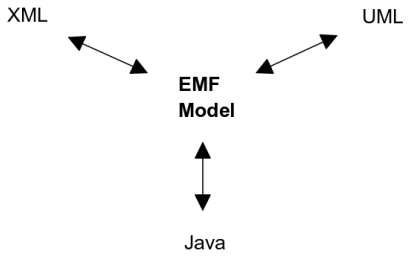


Figure 3.1. Unifying Java, XML, and UML [Steinberg et al., 2008]

represented and the semantics defines the meaning of a model [van Hoorn, 2014]. Modeling languages can be divided in General Purpose Modeling Languages (GPMLs) and Domain Specific Languages (DSLs). The first are languages which are designed to apply to any software domain. Prominent examples of GPMLs are the Unified Modeling Language (UML) and state machines. The latter are used to express a specific domain in a software system. Well-known examples are the Hypertext Markup Language (HTML) and the Structured Query Language (SQL) [Brambilla et al., 2012].

The second required technology is model transformation. A model transformation is the process of transforming a source model into a target model. It is possible to distinguish between different transformation strategies. (i) the model-2-text (M2T) and text-2-model (T2M) strategies are transformations which transfer models into a textual representation and vice versa, e.g., source code. (ii) The model-2-model (M2M) approach transforms the source model into the representation of the target model [Czarnecki and Helsen, 2006].

3.2 Eclipse Modeling Project (EMP)

The EMP is project, under the patronage of the Eclipse Foundation, which was created to collect different MDSD technology at one place. The different technologies are organized as subprojects and can be separated in support for abstract and concrete syntax development, model transformation and implementations of industry-standard meta-models [Eclipse Foundation, 2014a]. In this thesis the two subprojects Eclipse Modeling Framework (EMF) and Connected Data Objects (CDO) will heavily be used. In the following those two projects will be introduced more detailed.

3.2.1 Eclipse Modeling Framework (EMF)

The purpose of the Eclipse Modeling Framework (EMF) project is split into two. On the one hand it is a modeling framework and on the other hand it is a code generation tool. EMF is used to build tools and applications, which are based on a structured data model. EMF

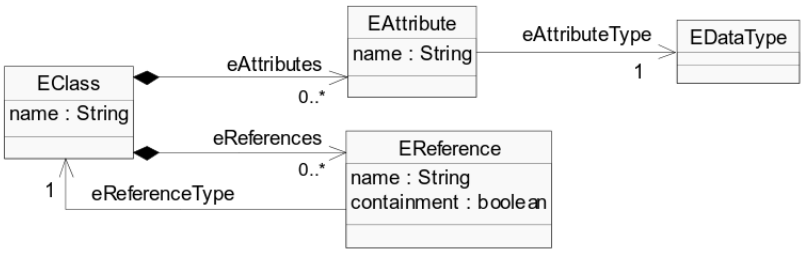


Figure 3.2. Minimal subset of Ecore model [Steinberg et al., 2008]

provides tools and runtime support for models which are defined in the XML Metadata Interchange (XMI) format [Eclipse Foundation, 2014a]. Thus, it can be said that EMF is a framework which lets you create a model in different forms and generate the others. As depicted in Figure 3.1, EMF unifies Java, Extensible Markup Language (XML), and UML. It does not matter which one is used, the resulting EMF model links them together [Steinberg et al., 2008].

Models in EMF are represented as Ecore models. Ecore itself is a EMF model, consequently it is it's own meta-model. If the model of a model is a meta-model, but the model itself is a meta-model. Then, the meta-model is a meta-meta-model [Steinberg et al., 2008].

As can be seen from Figure 3.2, Ecore provides four base classes to create an EMF model. *EClass*, *EAttribute*, *EReference*, and *EDataType*. *EClass* is used to model classes. Each *EClass* can have zero or more attributes of type *EAttribute* and zero or more references of type *EReference*. *EAttribute* is used to model the simple data types of objects. As depicted in Figure 3.2, the type of an attribute is defined by an *EDataType*. *EDataTypes* represent simple types which are not modeled as *EClass*, instead they are associated with plain Java primitives or object types. Associations between classes are modeled as *EReference*. Contrary to *EAttribute*, the type of an *EReference* is an *EClass* (see Figure 3.2). *EReference* provides the *containment* property to define a whole-part relationship. Thus, it is possible to couple the lifetime of an *EReference* to the associated container [Steinberg et al., 2008].

3.2.2 Connected Data Objects (CDO)

In addition to the previously described features, EMF provides a persistence framework as well. Since EMF models can be represented as XMI files, the persistence framework serializes model instance to XML files and deserializes from XML files. According to Stepper [2008], this approach has several drawbacks. An excerpt of drawbacks are limited resource size, no lazy loading, no concurrent modification, and no transactions. To circumvent these shortcomings, the Connected Data Objects (CDO) model repository was developed.

CDO is a pure Java model repository for EMF models and meta-models. Figure 3.3 depicts a rough overview of the general CDO architecture. As can be seen CDO has a 3-tier

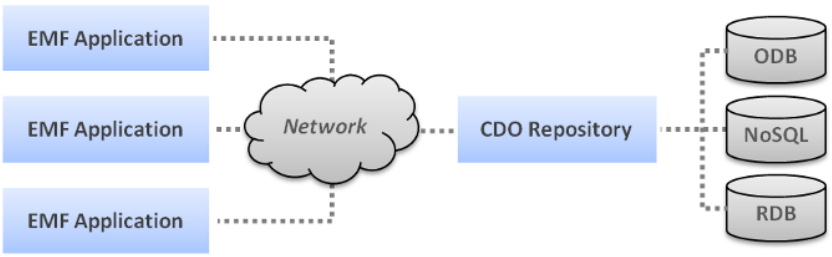


Figure 3.3. CDO architecture overview [Stepper, 2015]

architecture. *EMF Application* (*CDO client*), *CDO Repository* (*CDO server*), and *Database*. The purpose of CDO is to provide a plurality of client application access to a shared model repository via the Internet. The model repository itself is considered as *CDO server* and can connect to different types of data storage back-ends, e.g., relational databases or NoSQL [Stepper, 2015]. Provided features of CDO are: Persistence, Multi User Access, Transactional Access, Transparent Temporality, Parallel Evolution, Scalability, Thread Safety, Collaboration, Data Integrity, Fault Tolerance, and Offline Work. For this thesis the important features are persistence and transaction access. A detailed description of all features is available at [Stepper, 2015].

The persistence feature allows to store EMF models in all kind of databases, what results in a vendor-specific free application code base. Accessing EMF model objects in a transactional manner is supported by optimistic and/or pessimistic locking. The CDO transaction support the Atomicity, Consistency, Isolation, and Durability (ACID) properties.

CDO Client/Server Architecture

To ease the usage of CDO, all components are implemented as *OSGI* bundles. The *OSGI* specification is maintained by the *OSGI Alliance*[1] and describes a dynamic module system for Java. Prominent reference implementations are Equinox[2] and ApacheFelix[3]. Equinox is moreover heavily used within the Eclipse IDE. However, neither CDO client nor CDO server do necessarily require a running *OSGI* environment and can perfectly operate stand-alone. But, a missing *OSGI* environment increases the manual configuration effort [Stepper, 2015].

The internal architecture of a CDO client application is illustrated in Figure 3.4. The core components can be distinguished in three groups. *Models* and *EMF* provide support to integrate and interact with EMF models. *Protocol*, *Transport*, and *Net4j Core* enable the data transport via the Internet. *CDO Client* interleaves all components and grants the *Application* access to the CDO infrastructure [Stepper, 2015].

[1] http://www.osgi.org
[2] http://www.eclipse.org/equinox/
[3] http://felix.apache.org/

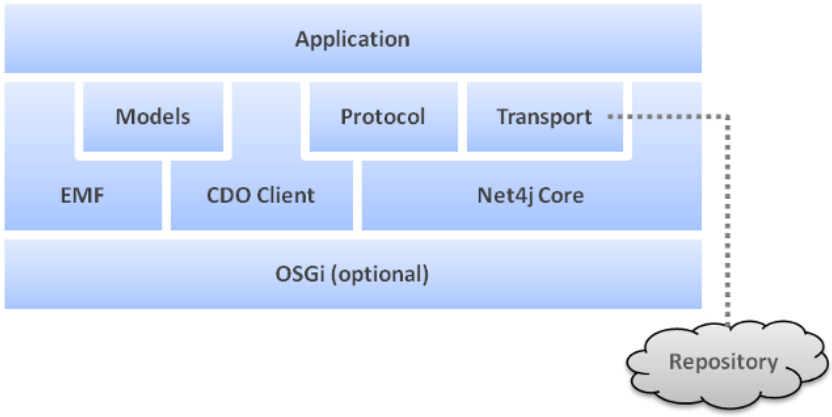


Figure 3.4. CDO client architecture [Stepper, 2015]

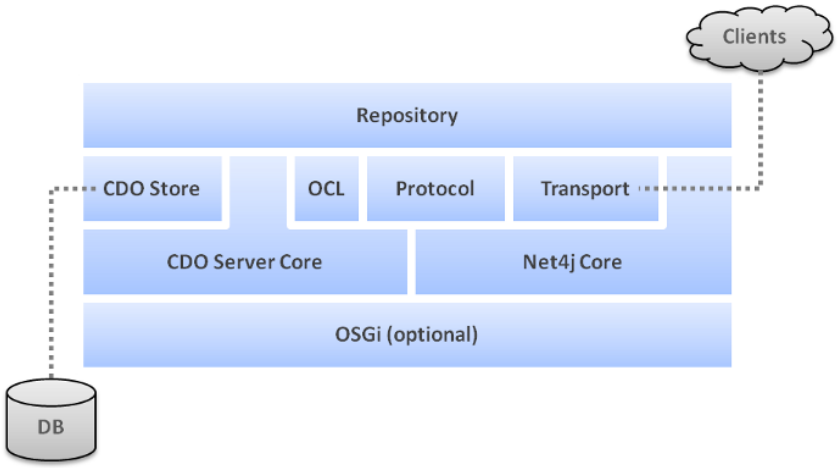


Figure 3.5. CDO server architecture [Stepper, 2015]

Figure 3.5 depicts the core components of a CDO server, or rather a CDO model repository. The core of each model repository is the *CDO Server Core* component. This component comprises all functionality which is later provided by the repository, e.g., caching, lock manger, session manager, and query handling. *CDO Store* is the component that acts as database backend and decouples the repository of proprietary persistence technologies like JDBC databases or Hibernate. *Protocol*, *Net4j*[4] *Core* and *Transport* enable the client/server communication [Stepper, 2015].

[4]https://wiki.eclipse.org/Net4j

3.3 Not only SQL (NoSQL)

Not only SQL (NoSQL) is the generic term for databases with aim to circumvent the functionalities of SQL databases. Sometimes NoSQL is also referred to *No SQl*, but that is not completely true. Indeed, there are databases that dispense SQL entirely, but also some that have encapsulated it [Weber, 2010]. However, no NoSQL database relies on a relational database model. Consequently, relations (represented as tables) are missing completely. According to Weber [2010], NoSQL databases are divided in four core categories.

- **Key/Value (K/V) stores**
  Datasets in K/V stores always consists of a unique key and a corresponding value. The value can be of any kind. This might be simple types (e.g., integer, byte arrays, etc.) or binary objects (e.g., images or text files). An example of a K/V store is *redis*[5].

- **Document store**
  The approach of document stores is to serialize data to human readable data files like, JavaScript Object Notation (JSON) or YAML Ain't Markup Language (YAML) and store this representation as entire document. Contrary to K/V stores, document based stores need to know what kind of data is stored. An example of a document store is MongoDB[6].

- **Graph databases**
  In Graph databases, data records are implemented as directed/undirected graph. Due to the fact that graphs consist of vertices and edges, the data records are interrelated. Neo4J[7] is an example of a graph database.

- **Column-oriented databases**
  Column-oriented databases switch from a row oriented (SQL like) to a column oriented database layout. This means, a row contains an arbitrary amount of values of the same attribute. In turn, that means that one data record corresponds to exactly one column. An example of a column-oriented database is Apache HBase[8].

3.4 Architecture of Enterprise Applications

Since this thesis focuses on the domain of enterprise application systems (EASs), this section will give a brief overview of common architectural styles related to EASs. In a common sense, architecture of software systems consists of two parts. The high-level breakdown into parts and early, hard to change, design decisions [Fowler, 2002]. In the context of this

[5] http://redis.io/
[6] https://www.mongodb.org/
[7] http://neo4j.com/
[8] http://hbase.apache.org/

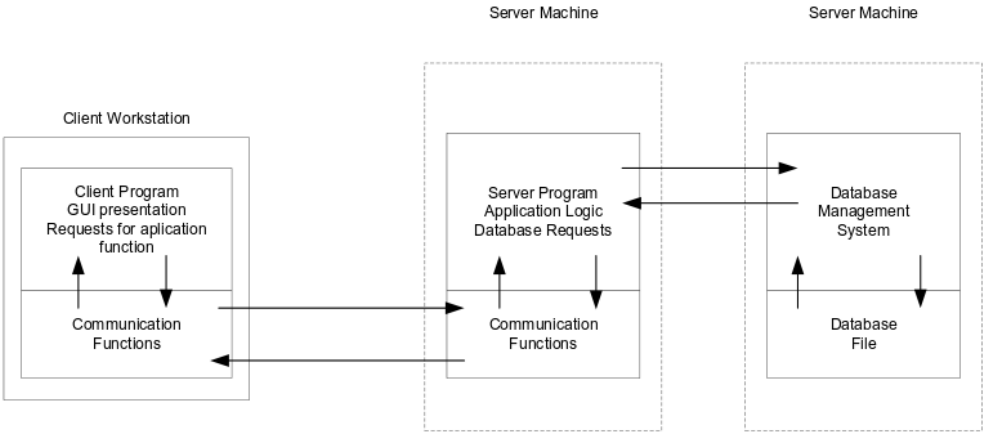


Figure 3.6. Layered architecture example [Kambalyal, 2010]

thesis EASs are considered as distributed Java business applications, e.g., payroll, patient records, and credit scoring.

According to Fowler [2002], layering is the most important technique to break a system apart. Layering brings many benefits: (i) single layers are better understandable and can be considered as a whole, (ii) layers might be replaced with an alternative implementation, and (iii) dependencies between layers are reduced. Modern EASs contain at least a presentation, a domain, and a data source layer. A three-layered architecture is shown in Figure 3.6. The first layer is responsible to display information and enable user input, the second layer contains the logic of the system and the latter establishes connections to any kind of data source, e.g., databases and message systems. As depicted in Figure 3.6, the presentation layer is located on the client machine, while business and presentation layers are located on the server machine. Depending on the application, the presentation layer might also be available at the server machine to prepare the data for subsequent display on the client machine.

Modern EASs rely on specific technologies, the most prominent are presented in the following.

- **Load-Balancer** A load-balancer redirects incoming requests to several instances of the requested server-side functionality.
- **Transaction Manager** A transaction manager handles a set of system transactions. If one transaction fails, the transaction manager is responsible to roll back all previous, successfully executed transactions. Transaction managers are needed to keep the EAS in a consistent state.
- **Web Clients** An increasing number of EASs is based on web technologies and is therefore

shipped with a web-frontend. A web-frontend is considered as thin-client, this means that all processor-intensive tasks are outsourced to the server.

- **Messaging** A Message-Oriented Middleware (MOM) enables a loose couple communication between several clients within a distributed environment.

- **Web Services** Web services are a further technology to enable a loose couple communication between several clients within a distributed environment. Furthermore, composing web services enable the possibility to create whole new applications.

3.5 Architecture Modeling Languages

A software architecture exposes the coarse structure of a software system in form of various interacting components. A well-defined architecture is a key success factor for complex software system, because a software architect is able to reason about system properties on an higher level of abstraction. Bass et al. [2012] define software architecture as stated in Definition 2.

Definition 2 *The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.*

In order to utilize the benefits of good software architectures and thus to avoid informal and ad-hoc approaches, several formal notations, so-called Architecture Description Languages (ADLs), to present system architectures were proposed [Garlan et al., 2010]. According to Matevska [2010] a software architecture must be viewed from various viewpoints. ADLs share a common intersection of viewpoints – static viewpoint, dynamic viewpoint, and deployment viewpoint. The static viewpoint considers the decomposition of the system in form of components and connectors. The dynamic viewpoint considers the runtime interaction among the components. The allocation of software components on hardware components is considered by the deployment viewpoint.

In research there is still a disunity to what degree an ADL should support the developers. On the one hand the major role of an ADL is to provide assistance by understanding a software systems. Such ADLs need a simple and graphical syntax and do not necessarily have a formally defined semantic. On the other hand the trend arose to provide formal definitions and semantics. This trend focuses on enabling powerful tools like model checkers, compilers, and parsers [Medvidovic and Taylor, 2000].

Since most ADLs are proprietary languages, w.r.t. syntax, formalism, and semantics, this thesis focuses on the existing ADL UML 2. To support modeling large-scaled component-based software architectures (CBSAs), the Object Management Group (OMG) introduced new diagram types in the UML 2 specification. Diagram types for timing, interaction overview, composite structures, components, and packages were added [Hamilton and Miles, 2006].

3.6 Model Extraction

This section introduces different techniques, how models can be extracted from the system under analysis (SUA). The introduced extraction techniques are based on reverse engineering and in the scope of this thesis considered to gain information about the system's architecture, including interfaces and components, the component's behaviour, how components are assembled, the resource environment, how components are allocated in the environment and how the system is used. Reverse engineering techniques are separated into two main approaches: *Static analysis* and *dynamic analysis* [Chikofsky et al., 1990]. However, both approaches have advantages and disadvantages that is why both can be combined to a hybrid approach.

Static analysis is applied to source code or binaries, without actually executing the code.This enables extracting interface and components. Also, the internal behaviour of components and interconnections to other components are discoverable. The process is split in three steps: (i) parsing the source code, (ii) generating the component model, (iii) computing complexity metrics [Krogmann, 2012]. A drawback of *static analysis* is that no information about the resource environment is available [Krogmann, 2012].

Dynamic analysis techniques execute the available code and monitor the SUA. To enable *dynamic analysis*, either the source/byte code has to be instrumented, or the execution environment already provides functionalities to monitor the SUA [Krogmann, 2012]. Due to the fact, that a running system is examined, it is additionally possible to gain insight in the resource demands of the components. A further benefit of *dynamic analysis* is, that also distributed systems can be analysed and a complete environment model can be extracted [Krogmann, 2012]. To ease the instrumentation of SUAs, there are several monitoring frameworks available, e.g., the Kieker framework [van Hoorn et al., 2012].

The combination of *static analysis* and *dynamic analysis* is called a hybrid approach. Hybrid approaches make it possible to exploit the advantages of both analysis approaches and so compensate the respective disadvantages [Chikofsky et al., 1990].

Due to the fact that *dynamic analysis* approaches rely on representative test cases or actual SUAs, *static analysis* approaches are considered to be more precise because those are capable to uncover all branches within a software system. While most *static analysis* approaches are limited with respect to complexity and size of the analysed system, *dynamic analysis* approaches can handle and analyse large and distributed systems. For example runtime bindings are already available for *dynamic analysis* [Krogmann, 2012].

3.7 Kieker Framework

Kieker is an extensible framework for analysing the runtime behaviour of software systems. In the scope of this thesis, especially the feature of control flow trace reconstruction is important. As depicted in Figure 3.7, the framework provides two main features. Monitoring (*Kieker.Monitoring*) and analysis (*Kieker.Anaylsis*). *Kieker.Monitoring* comprises

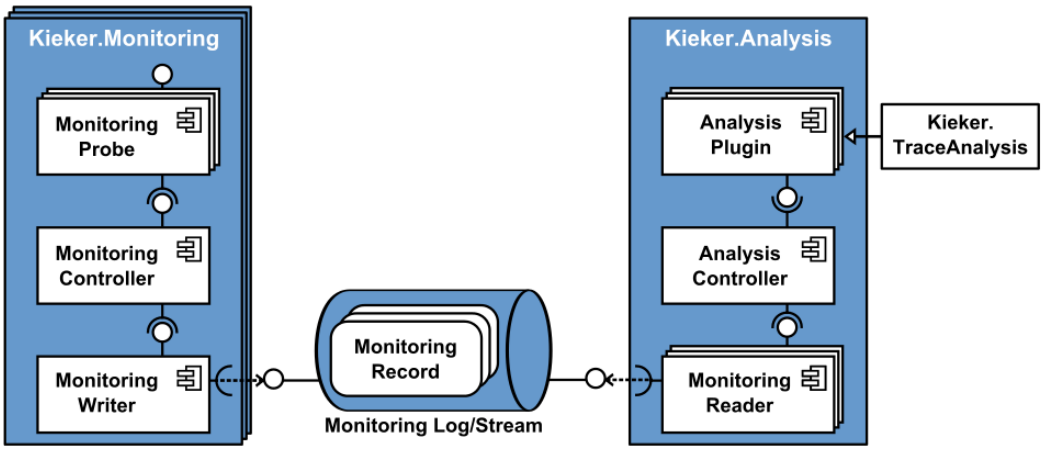


Figure 3.7. Kieker framework overview [Kieker Project]

functionality for program instrumentation, data collection, and logging. While monitoring a SUA, Kieker is able to provide various kinds of monitoring data (*Monitoring Record*), but all data records share a common data structure to assure consecutive interoperability among all features. Examples of *Monitoring Record*s are operation executions, CPU utilization, and memory consumption. To represent executed software operations Kieker provides so-called *OperationExecutionRecords*. Those records encompass information about the host on which the operation was performed, the signature of the executed operation, a timestamp when the record was logged, further timestamps for start and end of the operation, and properties to trace back the control flow.

As shown in Figure 3.7, data collection is performed by so-called probes (*Monitoring Probe*). All *Monitoring Records* are written to an internal, but extensible, writer/reader framework. Within the writer/reader framework exist always pairs of one writer (*Monitoring Writer*) and one reader (*Monitoring Reader*). Writer/reader pairs are provided for file system, database, Java Message Service (JMS), and Plain Old Java Objects (POJOs). While the *Monitoring Writers* are always utilized by *Kieker.Monitoring*, *Kieker.Anaylsis* reads the incoming *Monitoring Records* with the *Monitoring Readers*. The writer/reader framework thus bridges the gap between monitoring and analysis [van Hoorn et al., 2012]. This thesis mainly uses *Kieker.Analysis*, hence this will be presented more detailed in the following.

3.7.1 Kieker Analysis

Kieker.Analysis is responsible to read, analyse, and visualize the *Monitoring Records* collected by the *Monitoring Probes*. The core components of *Kieker.Analysis* are *Monitoring Readers*, the *Analysis Controller*, and *Analysis Plugins* (Figure 3.7). The *Analysis Controller* is used

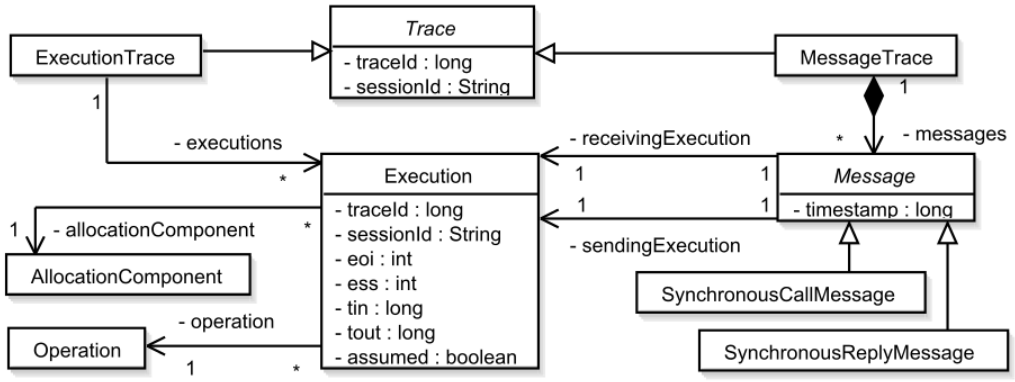


Figure 3.8. Kieker trace meta-model [van Hoorn, 2014]

to create new instances of *Kieker.Analyis*. As depicted in Figure 3.7, each *Kieker.Analysis* instance consists of several *Analysis Plugins*. All *Analysis Plugins* are lined up in a pipe and filter architectural style. To enable a pipe and filter structure each *Analysis Plugin* provides zero or more input ports and several output ports. Kieker distinguishes between two kinds of *Analyis Plugins*, readers and filters. Consequently, *Monitoring Readers* are *Analysis Plugins* itself. The first *Analysis Plugin* in the pipe and filter chain is a *Monitoring Reader*, followed by an unlimited list of filters (*Analysis Plugin*).

The *Analysis Controller* is used to create the pipe and filter chain. The final configuration how filters are assembled is either done programmatically or is provided as configuration file. It should be noted that *Analysis Plugins* must never emanating from preceding or trailing *Analysis Plugins*. This restriction facilitates unlimited usage of *Analysis Plugins*. A predefined pipe and filter composition of *Analysis Plugins* is provided as *Kiker.TraceAnalysis* tool. This is a stand-alone application to analyse already available monitoring data. One feature of *Kieker.TraceAnalysis* is to reconstruct executed control flow traces. This is an interesting feature, because as already described in Section 2.1 the Enterprise Performance Model (EPM) has to reflect control flow traces as well.

3.7.2 Kieker Trace Meta-Model

Within the Kieker framework execution traces are available in two kind of representations. *Execution Trace* and *Message Trace*. As shown in Figure 3.8, both share a common base entity: *Trace*. As depicted, the representations differ how the actual *executions* are represented. As *Execution Trace* the trace is represented as plain list of *executions*. In contrast, a *Message Trace* is an aggregation of *messages*. Internally, an *Execution Trace* is the starting point and gets transformed into a *Message Trace*. How the trace reconstruction takes place is out of scope of this thesis. Further information is available in [van Hoorn, 2014; van Hoorn

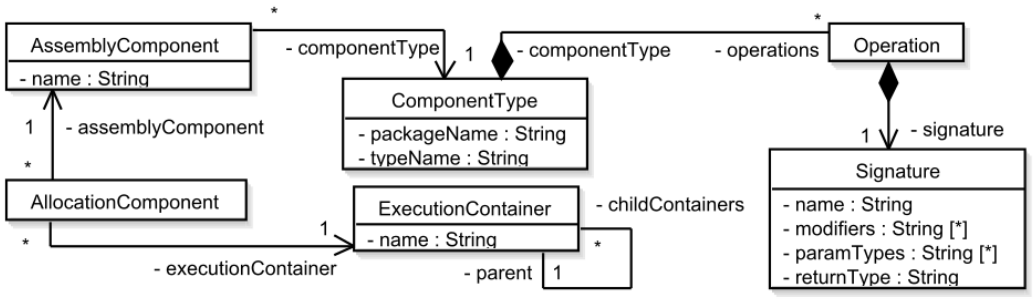


Figure 3.9. Subset of Kieker meta-model to represent software systems [van Hoorn, 2014]

et al., 2009]. A *message* is the logical grouping of a receiving, and a sending *execution*. Furthermore, a *message* adopts a certain role: *Synchronous Call Message* or *Synchronous Reply Message*. This distinction clearly defines, if a certain *message* starts an *execution*, or if it is returning *execution*. Additionally, each *execution* has attached one *Allocation Component* and one *Operation*. *Allocation Component* and *Operation* are entities of the internal Kieker meta-model to represent software systems (see Figure 3.9). As depicted, an *Operation* is part of a *Component Type*. In this scope a *Component Type* represents, simply put, an available class within the system. From this it follows that an *Operation* reflects a method of the class. Furthermore, Figure 3.9 illustrates the dependency between *Allocation Component* and *Execution Container*. By the combination of an *Allocation Components* and an *Operations* within a *message*, it is possible to regain the information about the executed method and where it was executed.

Chapter 4

The Enterprise Performance Model (EPM)

This chapter describes the meta-model which was developed for the *diagnoseIT* framework. The presented results are the deliverables of work packages WP1.1 (*Requirements Analysis* (2.1.1)), WP1.2 (*Review Existing Meta-Models and Check Applicability* (2.1.2)), and WP1.3 (*Specification and Implementation of the diagnoseIT Meta-Model*s (2.1.3)).

The remainder of this chapter is structured as follows. Section 4.1 constitutes the requirements of the EPM. A detailed description of the meta-model is given in Section 4.2. The chapter closes with an overview how the meta-model was implemented using the Eclipse Modeling Project (EMP).

4.1 Requirements

This section outlines the requirements which the EPM has to fulfil. During the requirement analysis it turns out that the most important requirements are those, which were contributed from the participants of the *diagnoseIT* project. In particular, the importance of an easy to use meta-model was stressed. Accordingly, the EPM is designed as simple as possible and thus can be considered as initial version for the *diagnoseIT* project. To reduce the complexity of the model, especially complex extensibility mechanisms were neglected. Since the EPM has to cover very different components of a system under analysis (SUA) a further requirement was to partition the meta-model in complementing sub meta-models. This partitioning can be considered as a simplification of the entire EPM as well.

Table 4.1. Complementing sub meta-models of the EPM

Enterprise Performance Model				
Software Module	Resource Environment	Allocation	Behaviour	Instrumentation

As depicted in Table 4.1, five complementing sub meta-model were identified. According to the table the EPM has to cover the software module, the resource environment, and the allocation. The allocation model is used to map a certain software module to the resource environment which is executing the software module. To allow subsequent performance analysis, the internal behavior of the software module must be covered. In the context of this thesis the behaviour means the execution traces, executed within a monitored

software module. In addition, to software module, resource environment, allocation, and behavior, the current instrumentation of the SUA is contained in the EPM.

The instrumentation describes how monitoring data is collected and thus directly influences the content of the other meta-models. Since the instrumentation meta-model is out of scope of this thesis, an initial version was defined but neither implemented nor tested.

The next step in the requirements analysis was to define the explicit requirements for each of the complementing sub meta-models. The entire list of requirements is depicted in Table 6.4. As can be seen from the table, the meta-model for software modules covers properties needed to reverse engineer the systems architecture. In particular, these are types, operations and signatures. To reconstruct component connections also the links between operations, respectively types, must be available. Since the incorrect usage of frameworks might lead to performance degradations (e.g. wrong definition of object-relational mapping files), the software module meta-model contains the employed frameworks.

Table 4.2. Enterprise Performance Model requirements

Enterprise Performance Model			
Software Module	Resource Environment	Allocation	Behaviour
Type	Node	Software Module	Message
Operation	Node Link	Execution Entity	Message Kind
Operation Link	Execution Entity		Duration
Signature	JVM		Allocation
Framework	Resource		Operation

As Table 6.4 shows, the resource environment meta-model covers all relevant information concerning the execution environment. Internally, the resource environment meta-model is split in hardware and software components. Nodes, node links and resources are considered as hardware components. A node describes a processing unit, that means a physical server. A resource represents physical resources like CPU, memory, and disk space. So-called execution entities (e.g., application servers, web servers, or databases) are used to represent the executing software components. Since the *diagnoseIT* approach focuses on Java applications also the underlying Java Virtual Machine (JVM) instances are covered by the resource environment. A single JVM can also be an execution entity.

The allocation model is used to establish a connection between software modules and resource environments. For this purpose the allocation model creates mappings which describe which software module is executed by which execution entity.

All previously described meta-models are related to static information of the SUA. In contrast, the behavior meta-model refers to the dynamic data, which is collected while monitoring a software system. In scope of the *diagnoseIT* approach behavior is equivalent to execution traces. Due to this definition will the behavior meta-model henceforth be named trace meta-model. Execution traces consist theoretically of an infinite amount of messages.

Messages can be of a certain kind (e.g., Call, or Reply) and each message is associated with a certain operation of the software module meta-model. To regain the information on which execution entity (respectively node) the message was executed it contains the mapping information of the allocation model. Additionally, an execution trace contains the execution duration.

4.2 EPM Specification

This section describes the meta-model developed for the *diagnoseIT* approach. As described in Section 3.2.1, the Eclipse Modeling Framework (EMF) unifies Java, Extensible Markup Language (XML), and the Unified Modeling Language (UML). To comply to theses specifications and therefore enable a subsequent transformation to an Ecore model, the abstract syntax of the EPM is presented as UML class diagrams. In order that the model is better understood, UML object and sequence diagrams are used to present exemplary instances of the EPM. In accordance with the requirement that the EPM is subdivided into five complementing sub meta-models, the EPM was designed as shown in Figure 4.1. The depicted meta-model is considered as a conceptual overview, since the actual implementation stipulates that an instance of the EPM can comprise several instances of the sub meta-models. This means for example that *diagnoseIT* is able to monitor two different software modules at the same time.

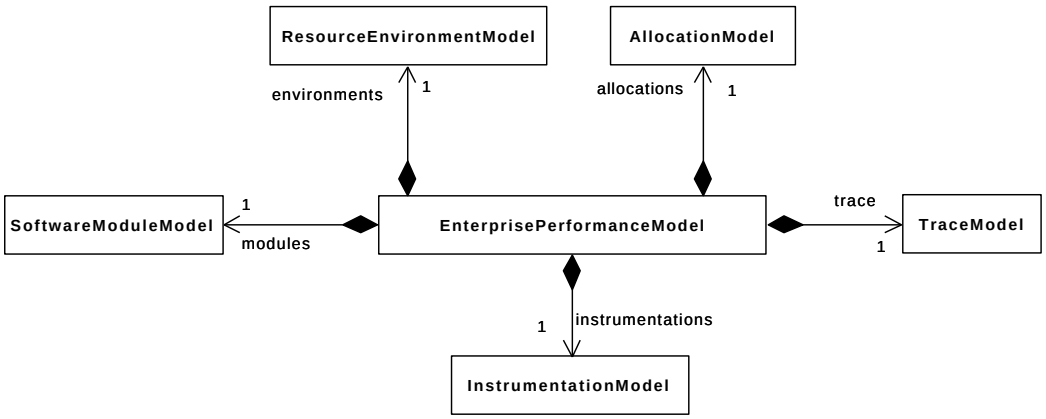


Figure 4.1. Conceptual overview of the EPM

4.2.1 Software Module

The software module meta-model contains all elements to recapture a software system's architecture. The smallest, but closed, building block of a Java application is a class. To avoid naming confusions we reference classes as types.

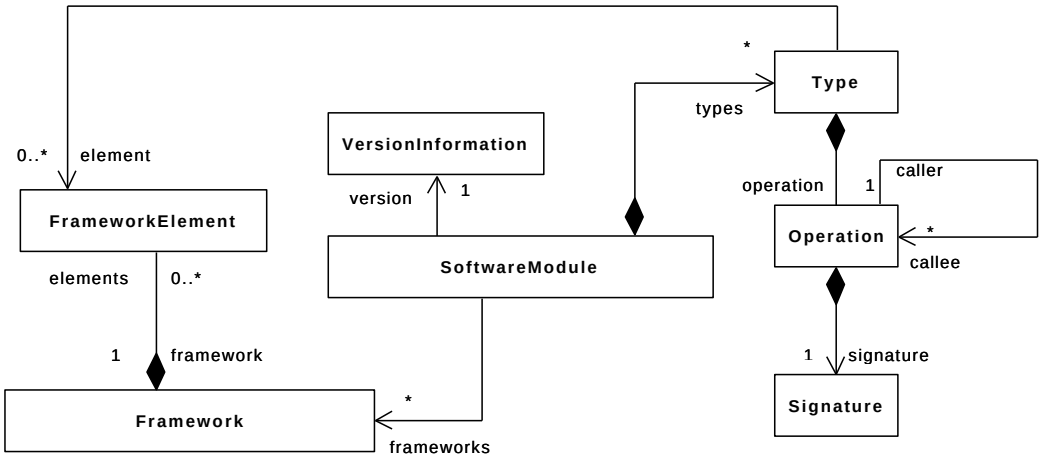


Figure 4.2. Meta-classes for software modules. Note that attributes are not shown.

As depicted in Figure 4.2 a software module (*SoftwareModule*) references types (*Type*), frameworks (*Framework*), and a version (*VersionInformation*). The version (*VersionInformation*) allows the EPM to deal with same software modules but in different versions. Consequently, a software module (*SoftwareModule*) is uniquely identifiable by its name and version.

We define types (*Type*) as lists of operations (*Operation*). Each operation (*Operation*) references exactly one signature (*Signature*). Thus, operations (*Operation*) are unique by design and method overloading is enabled by default. To implement the requirement that component connections must be discoverable (Section 4.1), each operation (*Operation*) defines a list with its callees. A callee is an operation (*Operation*), which an operation (*Operation*) is able to call.

To enable the performance problem diagnosis based on utilized frameworks (Section 4.1), each software module (*SoftwareModule*) references a list of frameworks (*Framework*). Each framework (*Framework*) itself references a collection of elements (*FrameworkElement*). An element (*FrameworkElement*) is considered as the part of a framework which is utilized in a software system. Possible elements (*FrameworkElement*) are, e.g., interfaces, abstract classes, or classes. If a type (*Type*) is somehow part of a framework, it has to reference the corresponding element (*FrameworkElement*).

Figure 4.3 illustrates an exemplary software module meta-model instance of the *JPetStore* application, which is used as running example in this thesis. As depicted *JPetStore* is a software module (*SoftwareModule*). It contains two types (*Type*), *AccountService* and *AccountActionBean*. Additionally, it is shown that it uses the *Stripes* framework (*Framework*). With respect to component links it is depicted that the *getAccount* operation (*Operation*) of *AccountService* links to the *signOn* operation (*Operation*) of the *AccountActionBean*. Furthermore, Figure 4.3 shows that *AccountActionBean* is an *actionBean* element (*FrameworkElement*)

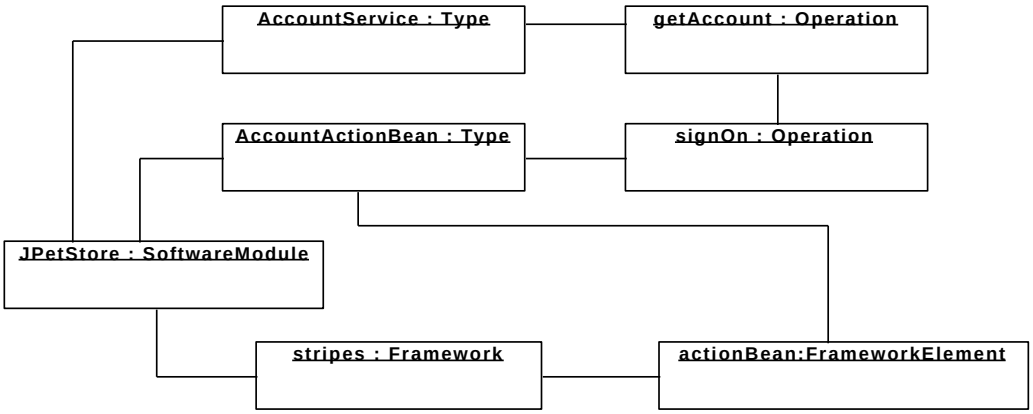


Figure 4.3. Software module meta-model instance of the JPetStore application. Note that attributes are not shown.

of the *Stripes* framework (*Framework*).

4.2.2 Resource Environment

The resource environment meta-model defines the set of available nodes (*Node*). The entire meta-model is depicted in Figure 4.4. As depicted a resource environment (*ResourceEnvironment*) consists of several nodes (*Node*). Each node has information about the installed operating system (*OperatingSystem*) attached. Node links are modeled as list of nodes (*Node*), for each node. Nodes (*Node*) reference a list of execution entities (*ExecutionEntity*). An execution entity (*ExecutionEntity*) is defined as a software component which executes, or at least partially executes, the monitored software system. Execution entities (*ExecutionEntity*) can therefore be of special types: application server (*ApplicationServer*), web server (*WerbServer*), or database (*Database*).

The easiest variation of an execution entity (*ExecutionEntity*) is a simple JVM. If an execution entity (*ExecutionEntity*) is executing a Java application, it references the executing JVM (*JVM*), including properties (*JVMProperties*) and arguments (*JVMArguments*). If a node (*Node*), an execution entity (*ExecutionEntity*), or a JVM (*JVM*) utilizes physical resources it has to reference the resource (*Resource*). A resource (*Resource*) comprises CPUs (*CPU*), memory (*Memory*), and disk space (*HDD*).

Figure 4.5 illustrates an exemplary resource environment meta-model instance. As depicted, the resource environment (*ResourceEnvironment*) *petstore* comprises one node (*Node*) *server1* and one execution entity (*ExecutionEntity*) *jvmEntity*. Since *jvmEntity* is of base type *ExecutionEntity*, it has to have a JVM (*JVM*) instance attached. Furthermore, it is shown that the JVM (*JVM*) *jvm1* utilizes two CPUs (*CPU*) and memory (*Memory*).

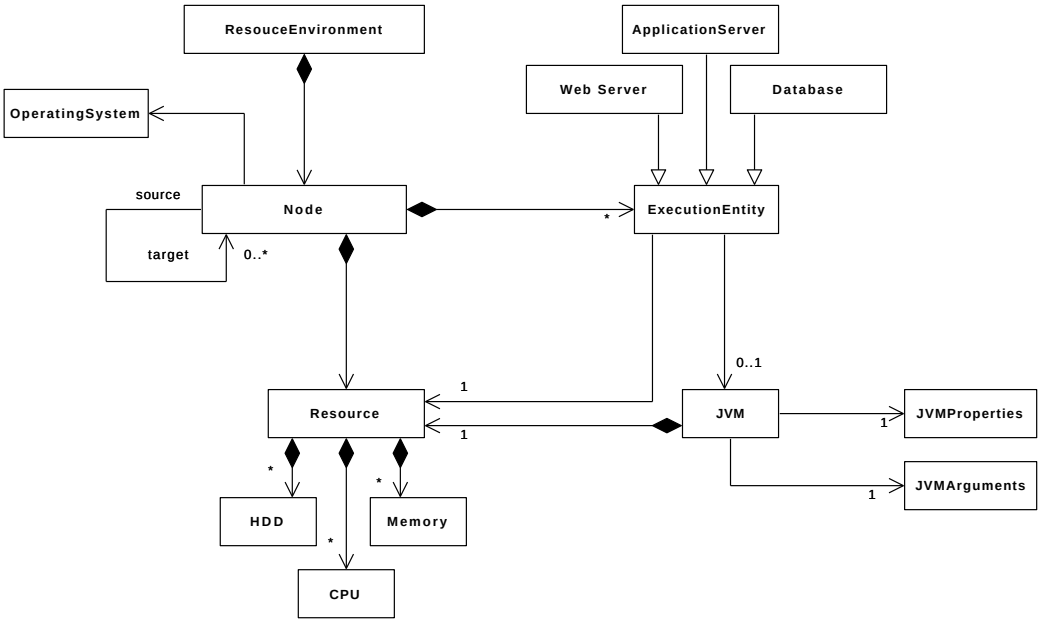


Figure 4.4. Meta-classes for resource environments. Note that attributes are not shown.

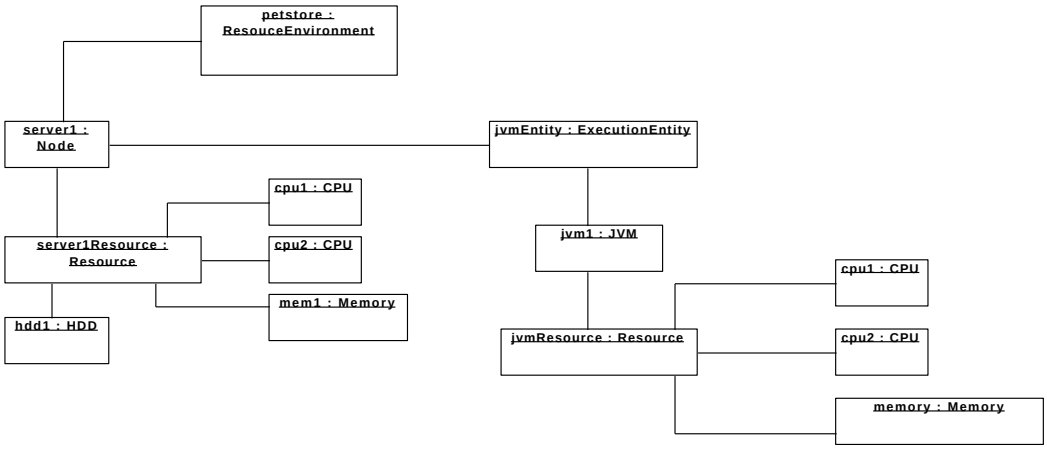
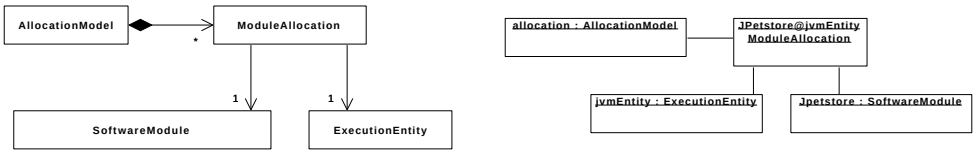


Figure 4.5. Exemplary instance of a resource environment meta-model.



(a) Meta-classes for allcocations

(b) Exemplary instance of the allocation meta-model

Figure 4.6. Meta-classes for allocations and exemplary instance.

4.2.3 Allocation

As described in Section 4.1, the purpose of the allocation model is to map a software module (*SoftwareModule*) to an execution entity (*Execution Entity*). This mapping was achieved by designing the allocation model, as depicted in Figure 4.6a. As can be seen, the allocation model (*AllocationModel*) contains a list of module allocations (*ModuleAllocation*). Each module allocation (*ModuelAllocation*) references one software module (*SoftwareModule*) and one execution entity (*ExecutionEntity*). For our running example this results in an allocation model (*AllocationModel*), as depicted in Figure 4.6b.

4.2.4 Trace

Figure 4.7 shows the meta-model which is used to represent execution traces. Each trace (*Trace*) is associated to a certain software module (*SoftwareModule*), hence the software module (*SoftwareModule*) is considered as owner of the trace (*Trace*). As depicted, a trace (*Trace*) consists of a list of messages (*Message*), which represent method calls. Since a method call always consists of one calling and one called method, each message comprises two operation executions (*OperationExecution*). Operation executions (*OperationExecution*) clearly define what (*Operation*) was executed and where (*ModuleAllocation*) it was executed. By the module allocation (*ModuleAllocation*), it becomes possible to determine the executing node (*Node*).

Furthermore, a message (*Message*) is of certain sort (*MessageSort*). Current available sorts (*MessageSort*) are: *SynchCall*, *AsynchCall*, *Reply*, *Database*, and *Exception*. In addition, to the default message (*Message*), the extended types *DatabaseMessage* and *ExceptionMessage* are available. Exception messages *(ExceptionMessage)* are used to indicate that errors occurred while executing the trace. The occurred error should be identifiable by error codes or stracktraces provided by the exception message (*ExceptionMessage*). Database messages (*DatabaseMessage*) are markers that a database call was executed. The kind of the database call is defined by a database operation (*DatabaseOperation*). Current available database operations (*DatabaseOperation*) are: *Select*, *Insert*, *Delete*, and *Update*.

To complete the running example, an exemplary execution within the *JPetStore* applica-

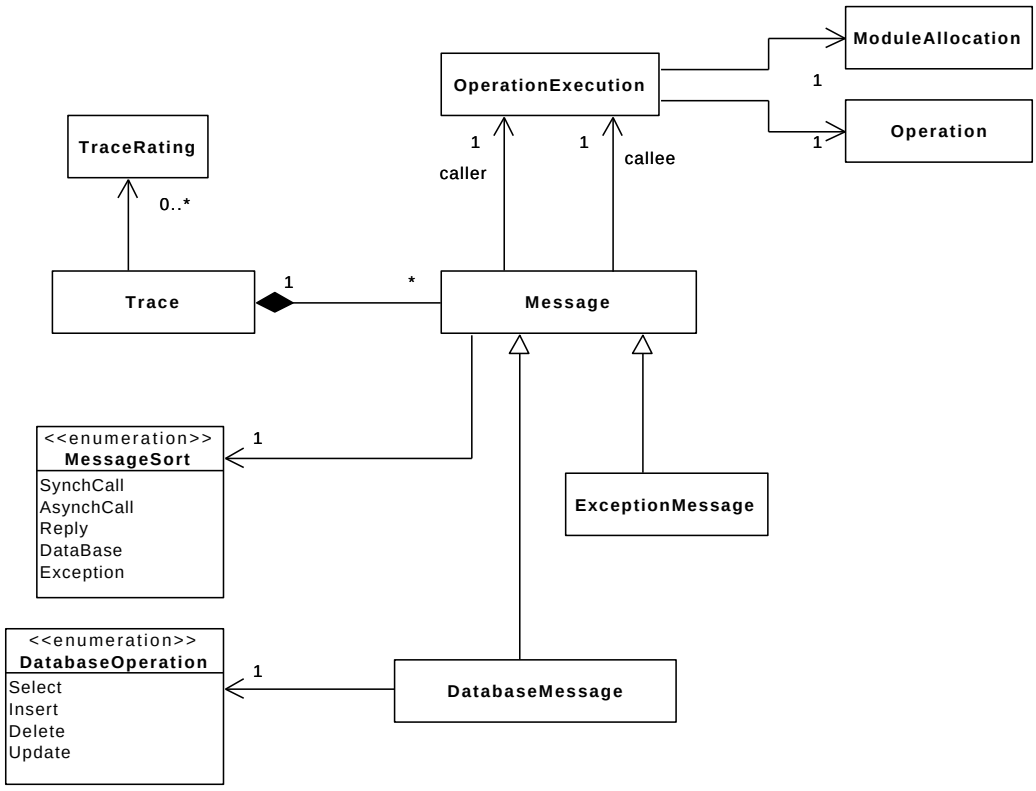
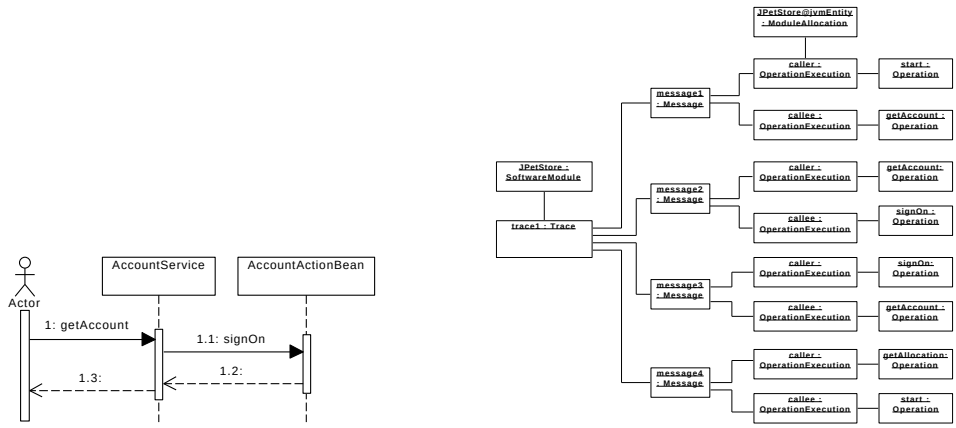


Figure 4.7. Meta-classes for traces. Note that attributes are not shown.

tion is depicted in Figure 4.8a. As shown, a user calls the *getAccount* operation (*Operation*) to retrieve his account information. However, before the account information is returned, the *AccountService* internally delegates the call to the *signOn* operation (*Operation*) of the *AccountActionBean*. The result of transforming the exemplary execution into the *diagnoseIT* trace meta-model is depicted in Figure 4.8b. We assume that all messages (*Message*) are executed by the same execution entity (*ExecutionEntity*). Therefore, the *JPetStore@jvmEntity* module allocation (*ModuelAllocation*) is shown only once. As shown, *trace1* consists of four messages (*Message*). *Message1* and *message2* of sort (*MessageSort*) *SynchCall*, *message3* and *message4* of *Reply*.

4.3 EPM Implementation

The carried out steps to transform the defined meta-models (4.2) into Java source code artifacts are described hereinafter. The entire transformation process is performed by



(a) Exemplary execution within the *JPetStore* application.

(b) Exemplary trace meta-model instance.

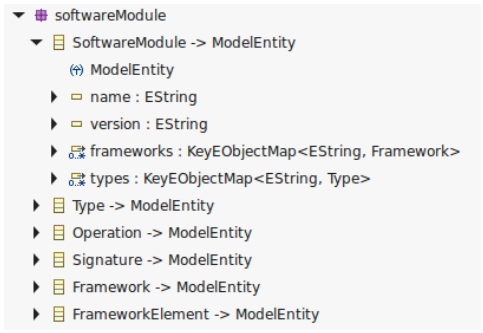
Figure 4.8. Example transformation from a method call to a trace meta-model instance.

using the Eclipse Modeling Framework (EMF), which is available as plugin for the Eclipse integrated development environment (IDE). As described in Section 3.2.1, the EMF is (i) a modeling framework and (ii) a code generation tool. Accordingly, we first transfer the meta-models to Ecore and then generate the Java source code. Exemplary for all meta-models, the steps are described for the software module meta-model.

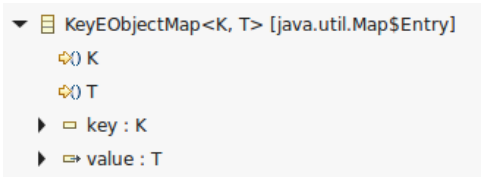
4.3.1 Ecore Modeling

Each Ecore model is a composition of the four base classes: *EClass*, *EAttribute*, *EReference*, and *EDataType* (3.2.1). The first step is to create *EClass* representations for all entities of the software module meta-model, as shown in Figure 4.9a. In addition, we decided that all model elements inherit from a common *ModelEntity* class. By means of this base *ModelEntity* class it is possible to distribute common functionality to all model elements with very little effort. At present, the *ModelEntity* has only one property, named *key*, to clearly identify model elements. Once all classes are prepared, the inter-dependencies can be established.

A software module (*EClass*) has four attributes: name, version, frameworks, and types. First two are modeled as attributes (*EAttribute*), latter as references (*EReference*). To illustrate how attributes and classes are defined in detail, an excerpt of the mandatory, most important, and most likely to change configuration values is depicted in Table 4.3. With respect to the software module (*EClass*) it is shown (i) that the inheritance of *ModelEntity* is defined by the *SuperType* property, (ii) that the name property (*EAttribute*) is of type (*EDataType*) string (*EString*), and (iii) that the types property (*EReference*) is of type (*EType*)



(a) Ecore representation of the softare module meta-model



(b) Ecore defintion of the KeyObjectMap class

Figure 4.9. Ecore modeling in Eclipse IDE

Table 4.3. Excerpt of Ecore model configuration properties.

EClass(SoftwareModule)		*EAttribute*(name)		*EReference*(types)	
Property	Value	Property	Value	Property	Value
Name	SoftwareModule	Name	name	Name	types
ESuperType	ModelEntity	EType	EString	EType	KeyObjectMap
Interface	false	Lower Bound	0	Lower Bound	0
Abstract	false	Upper Bound	1	Upper Bound	-1
				Containment	true

KeyObjectMap.

EMF provides common data types (*EDataType*) like *ESting*, *ELong*, *EMap*, *EObject*, etc., by default. Except for user-defined enumeration (*EEnum*), these are the data types which can be used as attributes (*EAttribute*). Relations to other data types (*EDataType*), e.g., the types reference (*EReference*), must be modeled as references (*EReference*).

The types reference (*EReference*) is a special feature. The amount of types as part of a software system can easily go into the 10-100 thousands. Therefore, we decided to model the types reference (*EReference*) as map. Thus, a fast an easy access to types is possible. This is achieved by (i) an upper bound of *-1* and (ii) a custom data type *KeyObjectMap*. The negative upper bound generally converts references (*EReference*) in lists. Since neither attributes (*EAttribute*) can be containments nor can references (*EReference*) by maps (*EDataType*), a custom type (*EClass*) as data type (*EDataType*) for maps is needed. Figure 4.9b uncovers the internal structure of *KeyObjectMap*, a simple key-value pair. The key as attribute (*EAttribute*) and the value as reference (*EReferнce*). Proper usage is depicted in Figure 4.9a.

4.3.2 Model-2-Text (M2T) Transformation

Once all meta-models are defined, the code generation can be carried out. Each modeling project within the Eclipse IDE includes in addition to the Ecore model a so-called generation model. The generation model provides a large amount of configuration values to affect the generated source code. A detailed functional description of all properties is available in [Steinberg et al., 2008]. To generate the EPM source code, the properties need to be set as depicted in Table 4.4. Since the EPM instances are stored in a Connected Data Objects (CDO) repository we need to assure that a CDO repository is able to handle all EPM elements properly. This is enabled by extending the CDO root class (*org.eclipse.emf.internal.cdo.CDOObjectImpl*), respectively interface (*org.eclipse.emf.cdo.CDOObject*). The *Base Package* property is used to integrate the generated source code in an the required package hierarchy. After a successful code generation the *SoftwareModule* interface looks like the following:
public interface de.unistuttgart.metamodel.softwareModule.SoftwareModule extends ModelEntity
As can be seen the *SofwareModule* extends our root class *ModelEntity* and *ModelEntity* is generated to:

public interface de.unistuttgart.metamodel.common.ModelEntity extends CDOObject

The purpose of the *ModelEntiy* base class was described in Section 4.3.1 and since this is the root class of all further EPM entities it is the only entity which inherits from *org.eclipse.emf.cdo.CDOObject*.

Table 4.4. Generation Model Properties

Property	Value
Base Package	de.unistuttgart.metamodel
Root Extend Interface	org.eclipse.emf.cdo.CDOObject
Root Extend Class	org.eclipse.emf.internal.cdo.CDOObjectImpl

Chapter 5

The *diagnoseIT* Framework

This chapter describes the realization of the *dianoseIT* framework. The deliverables of the previously roughly described work packages (WP1.4 and WP2.2-WP2.4) are presented in a structured, detailed manner. Also, the Enterprise Performance Model (EPM) of the previous chapter is taken and embedded in the framework's overall architecture.

The chapter is structured as follows. Section 5.1 summarizes the gathered requirements. Section 5.2 gives a short introduction to employed third-party libraries and frameworks. Section 5.3 provides a detailed overview of the developed framework as well as detailed insights to certain components.

5.1 Requirements

This section outlines the requirements for the EPM model repository, the System Model Maintenance Interface (SMMI), the Application Performance Management (APM) tool integration, and the *diagnoseIT* prototype architecture (see work packages WP1.4 (2.1.4) and WP2.1(2.2.1)). The requirements are split in functional requirements (FRs) and non-functional requirements (NFRs).

5.1.1 Functional Requirements (FRs)

The following introduced FRs assure that the developed approach meets the goals **G1** (*Enterprise Performance Model and Repository*) and **G2** (*System Model Maintenance Interface (SMMI) and Prototyp*) which were motivated in sections 2.1 and 2.2.

FR1 Configuration

diagnoseIT application instances are supposed to be deployed in various environments. Therefore, several components of the *diagnoseIT* framework require a flexible configuration mechanism. This mainly concerns components which serve as database backends.

FR2 Stand-Alone EPM Model Repository

By default, model repositories built on top of the Connected Data Objects (CDO) framework, run within an Eclipse environment. To bypass this prerequisite, *diagnoseIT* provides a CDO

environment which, is detached from Eclipse. Providing access to the EPM is the main task of the model repository. This encloses creating, reading, updating, and deleting parts of EPM instances. Additionally, the repository is partitioned to reflect the identical structure as the EPM (see Section 4.1).

FR3 Separate Data Storage for Static and Dynamic Data

Since the EPM covers static and dynamic data, *diagnoseIT* provides access to two database engines. This allows a separate storage and the possible overhead of CDO can be neglected for dynamic data. Arising dynamic data, e.g., execution traces, are stored separately in a *NoSQL* database.

FR4 System Model Maintenance Interface (SMMI)

As depicted in Figure 2.1, the SMMI serves as access point to the EPM model repository. Hence, all components which dependent on repository data utilize the SMMI. In summary, the internal Application Programming Interface (API) of the repository is published by the SMMI.

FR5 Adapter as integration facility

The *diagnoseIT* framework provides a configurable adapter as integration facility. This guarantees a seamless, unambiguous integration for third-party APM tools. The adapter grants access to all necessary information and resources, which might be necessary for APM tools.

FR6 Asynchronous Representational State Transfer (REST) API

All communication between a *diagnoseIT* application and adapters is realized by using REST services. An adapter provides all necessary interface descriptions to establish a bi-directional connection. The data exchange is realized with Data Transfer Objects (DTOs). By the use of DTOs, the EPM's internal structure is hidden and the data exchange is kept more lightweight.

FR7 Data Pre Processing

An adapter provides an extensible data pre-processing mechanism. The pre-processing is mainly used to prevent multiple transmissions of the same data and thus to limit the network traffic to a minimum. Additionally, the data is analyzed and further information is extracted.

FR8 Inaccessible Data Provisioning

As long as an adapter processes the latest data, it is essential that additional information is retrievable from the connected APM tool. This data exchanged is established by an optional supplementary interface. However, the adapter must not rely on the existence of this interface since it can not be assumed that each APM tool is capable to provide the required data.

FR9 Access Control

Each adapter has to log on to the *diagnoseIT* application. The adapter provides an identification number to the system and receives a valid session identification in return. The session identification is henceforth transmitted in addition to each request.

5.1.2 Non-Function Requirements (NFRs)

Non-functional requirements (NFRs) describe certain quality criteria to be met by the developed framework.

NFR1: Reusability

Software systems as well as requirements evolve and change over time. Hence, the *diagnoseIT* framework supports a modular structure to enable partial replacements of components. Additionally, it must be configurable (FR1) to be deployed in various execution environments.

NFR2: Extensibility

In addition to reusability, *diagnoseIT* provides expansion capabilities. By extension points it is possible to add additional functionality which either can not be provided from *diagnoseIT* or which might be replaced in the future.

NFR3: Context Awareness

All interactions with the model repository are executed in an enclosing context. With respect to the EPM this means, a context provides information what the origin of the data is (e.g. which *software module* is updated). This enables streamlined interface definitions, because less parameters are needed.

5.2 Supporting Software and Libraries

The development of the *diagnoseIT* framework is supported by third-party libraries/frameworks. Relying on third-party products reduces the development time, increases code

quality and enforces encapsulation. The used libraries are introduced in the following.

5.2.1 Dropwizard (Vers. 0.8.0)

Dropwizard is an application framework which is composed of stable and mature Java libraries. The goal of Dropwizard is to provide a simple and lightweight bundle containing performant and reliable implementations of everything a production-ready web application needs. By using Dropwizard, the application code remains focused on what it is actually intended to do [Dropwizard]. Although, it has not yet reached a 1.0 version it already provides a lot of support to develop RESTful applications.

In contrast to traditional Java web applications, Dropwizard applications are not shipped as Web application Archives (WARs) and deployed in application servers like Tomcat[1] or JBoss[2]. Instead, the applications are assembled as single Java Archives (JARs), including all required dependencies. Thus, an application can be deployed in every environment without worrying about necessary libraries. This supports FR1 and NFR1. Dropwizard applications enable their web application characteristic by starting a Jetty HTTP server internally. Jetty is a pure Java Web server and *javax.servlet* container. Due to its small size it is particularly suitable to be integrated into other software [Eclipse Foundation, 2015]. Additionally, Dropwizard embeds the Jersey framework for building RESTful applications.

5.2.2 Jersey (Vers. 2.1)

To ease the development of Web services build according to the REST architectural style, the Java API for RESTful Web Services (JAX-RS) specification was defined. JAX-RS defines a common API how RESTful Web service are build with Java [Hadley and Sandoz, 2009]. Jersey is a toolkit to simplify development of RESTful Java client-server applications and serves as JAX-RS reference implementation. But, in addition to compliance of the JAX-RS specification the toolkit provides additional features for further simplifications [Oracle Corporation, 2015b].

5.2.3 HK2 (Vers. 2.4.0-b15)

To increase reusability, testability, and maintainability of Java source code, the JSR-330[3] specification was developed to provide an extensible dependency injection API. HK2 is a lightweight and dynamic dependency injection framework, which implements the JSR-330 specification [Oracle Corporation, 2015a]. As Jersey internally uses HK2, benefit of using HK2 is a seamless integration in Jersey and the Dropwizard environment. Using HK2 supports NFR1 and NFR2.

[1] http://tomcat.apache.org/
[2] http://www.jboss.org/
[3] https://www.jcp.org/en/jsr/detail?id=330

5.2.4 Guava (Version 18.0)

Guava is a collection of free libraries for Java that is provided from Google. The library is intended as extension to the default Java development library and is used all over the Google Java services. Beside basic utility functions to increase comfort using Java, Guava provides, e.g., extended collections, concurrency libraries, caching, string processing, and simplified input/output processing [Google, 2015]. Within the *diagnoseIT* approach especially the basic Java language utilities, collections, and parts of the concurrency library are used.

5.2.5 MongoDB (Version 3.0.2)

According to FR3, *diagnoseIT* uses the MongoDB document store (see 3.3), to store dynamic trace data. MongoDB was chosen for several reason. The main reason was the already existing basic knowledge. This was supported by MongoDB's capabilities to handle large amount of data, as well as the availability of Java support.

5.3 Framework Implementation

This section describes the implementation of the *diagnoseIT* framework and provides detailed insights in the internal architecture. A high-level architecture overview is depicted in Figure 5.1. As can be seen, the framework is split in two major components: *Application* and *Adapter*. *Application* comprises all core components which provide the operational aspects of *diagnoseIT*. This includes, Web resources representing a REST API, the SMMI implementation, and the EPM repository. *MongoDBClient*, *CDOClient*, and *CDOServer* provide database functionality and are connected to corresponding database servers. The current implementation assumes that *CDOServer* is an embedded component. Hence, it starts and stops with the *Application*. However, Section 5.3.2 will demonstrate that *CDOServer* is easyly extractable and independently executable.

The *Adapter* component serves as interface for APM tools to communicate with *diagnoseIT*. For this purpose, the *Adapter* comprises all functionalities to process input data (*PreProcessor*) and to transmit it to the server (*WebResourceProxy*). APM tools interact with a so-called *Runtime*, which is considered as façade to the offered underlying components.

5.3.1 Application

Prior to a detailed description of the components shown in Figure 5.1, this section describes the internal architecture of the overall application. Mechanisms how components are started, connected, and stopped are introduced. Additionally, it is shown how frameworks and libraries described in Section 5.2 are utilized. The core classes of an application are depicted in Figure 5.2, classes of third-party libraries are accordingly prefixed, Dropwizard (DW), Jersey/JAX-RS (RS), and HK2.

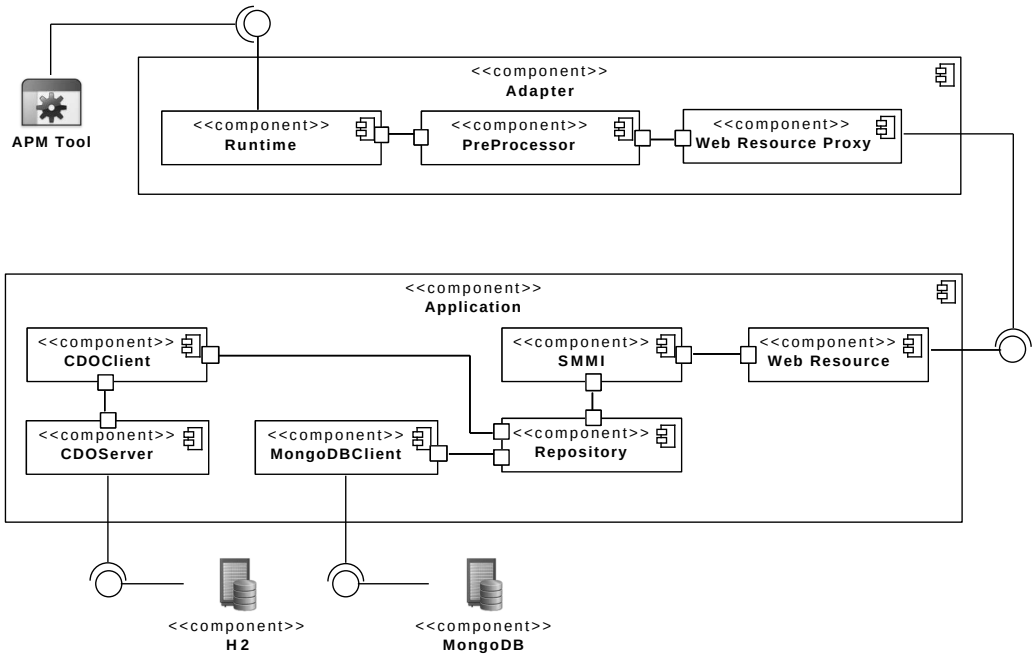


Figure 5.1. High-level *diagnoseIT* architecture overview.

The base class for all applications is *AbstractApplication*. By extending *DW.Application*, this is also the entry point to the Dropwizard framework. Each *AbstractApplication* requires an *ApplicationConfiguration*. Since an *ApplicationConfiguration* is not created manually but generated from a configuration file, this is described at the end of the section. Dropwizard provides a feature to define bundles and register them in a *DW.Application*. Bundles have to implement the *DW.Bundle* interface and are invoked while a *DW.Application* is starting to perform a certain piece of work. This feature is used to implement an *ApplicationBootstrapBundle*, which is responsible to create and initialize all further components.

The first step while an application is bootstrapped, is to create a *HK2.ServiceLocator*. As described in Section 5.2.3, HK2 is a dependency injection framework and a *HK2.ServiceLocator* is the container for all injectable classes. The classes to be managed by the container are defined by a *HK2.Binder* which is created by implementations of *AbstractApplication*. This architectural style supports NFR1 and NFR2.

After creating the *HK2.ServiceLocator*, an internal *AutomaticBootstrap* process is started to scan the classpath for subclasses of *IModelRepository*, *RS.Path*, and *DW.Managed*. All detected classes are instantiated by using the prior created *HK2.ServiceLocator*. This ensures that all classes get their dependencies properly injected. The further process differs by the corresponding type. *IModelRepositories* can directly be initialized. *RS.Path* and *DW.Managed*

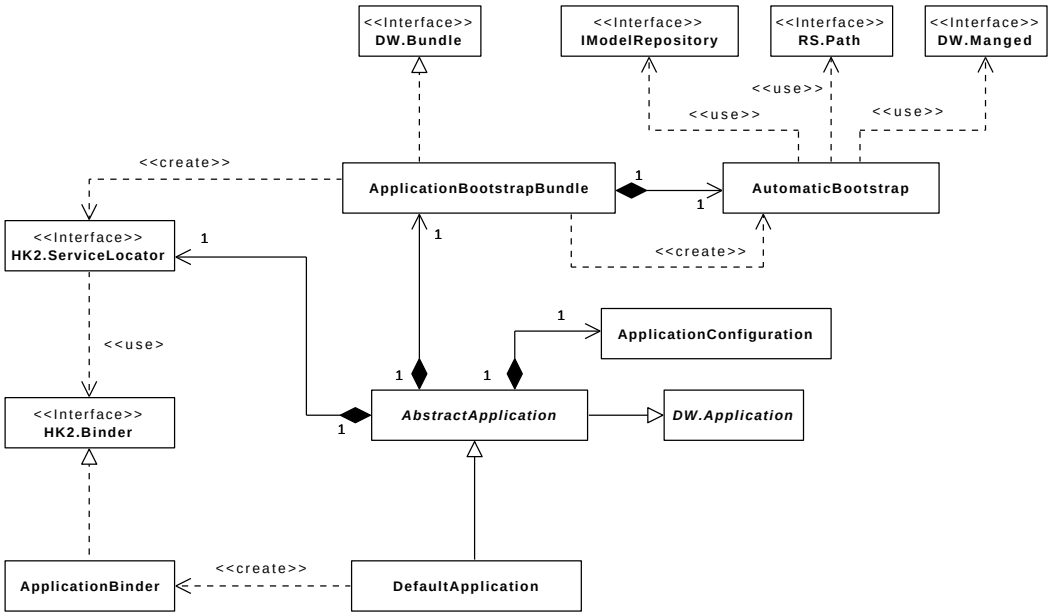


Figure 5.2. Core classes of the *Application* component.

are registered in the Dropwizard environment. The *DW.Managed* interface is used to integrate a class into the life-cycle of a Dropwizard application, this means implementing instances are started and stopped in common with the application. *RS.Path* is an annotation, defined by JAX-RS, to indicate a resource class. In short, a resource is a *REST* API definition and registered in the Jersey environment. A detailed description of resources follows in Section 5.3.5.

For an application to be started, it requires a configuration file in the YAML Ain't Markup Language (YAML) human-readable data serialization format, which is subsequently transformed in an *ApplicationConfiguration* instance. As the sample configuration file (Listing 5.1) shows, the configuration comprises settings for Dropwizard (e.g., *applicationContextPath*) and the *diagnoseIT* framework (e.g., configurations for the CDO server and client). Listing 5.2 demonstrates how an application is configured and requested to start the internal Jetty server. This flexible configuration mechanism supports FR1.

Listing 5.1. Example application configuration

```
#Dropwizrad configuations
server:
  applicationContextPath: /diagnoseIT
  applicationConnectors:
    - type: http
      port: 8080
```

```
7
#DiagnoseIT configurations
9 classpathScanning: ["de.unistuttgart.diagnoseIT.server"]
embeddedCDOServer: True
11 adapterWhiteList: ["1","2"]

13 cdoServerConfiguration:
  verbose: False
15   gracefulShutDown: True
  configurationFile: "META-INF/default-cdo-server.xml"
17
cdoClientConfiguration:
19   repository: "EnterprisePerformanceModel"
  connector: "localhost:2036"
```

Listing 5.2. Application start up

```
java -cp application.jar de.uni.ExampleApplication server configuration.yml
```

5.3.2 CDO Client and Server

Since the Eclipse Modeling Project (EMP) model repository is built on top of the CDO framework, the initial step in the course of this thesis was to implemented the CDO server and CDO client components. Normally, CDO repositories are created and used inside a running Eclipse integrated development environment (IDE). This is in contradiction with FR2 and thus *diagnoseIT* needed a fully functional, Eclipse independent, CDO environment. When researching how this can be achieved, it turns out that additional effort is needed [Stepper, 2015]. There exist several working examples which are, however, also contrary to FR2. They are primarily static and do not offer the whole configuration capabilities as if CDO runs in an Eclipse environment [Eclipse Foundation, 2014c].

Listing 5.3. CDO Server Configuration

```
1 <?xml version="1.0" encoding="UTF-8"?>
<cdoServer>
3     <acceptor type="tcp" listenAddr="0.0.0.0" port="2036"/>
    <repository name="EnterprisePerformanceModel">
5         <property name="optimisticLockingTimeout" value="10000"/>
        <store type="db">
7             <property name="connectionKeepAlivePeriod" value="60"/>
            <dbAdapter name="h2"/>
9             <dataSource class="org.h2.jdbcx.JdbcDataSource"
                    URL="jdbc:h2:/tmp/db/EnterprisePerformanceModel"/>
11         </store>
    </repository>
13 </cdoServer>
```

Within Eclipse, a CDO server component is fully configurable by means of a single Extensible Markup Language (XML) file. An example configuration file is provided in

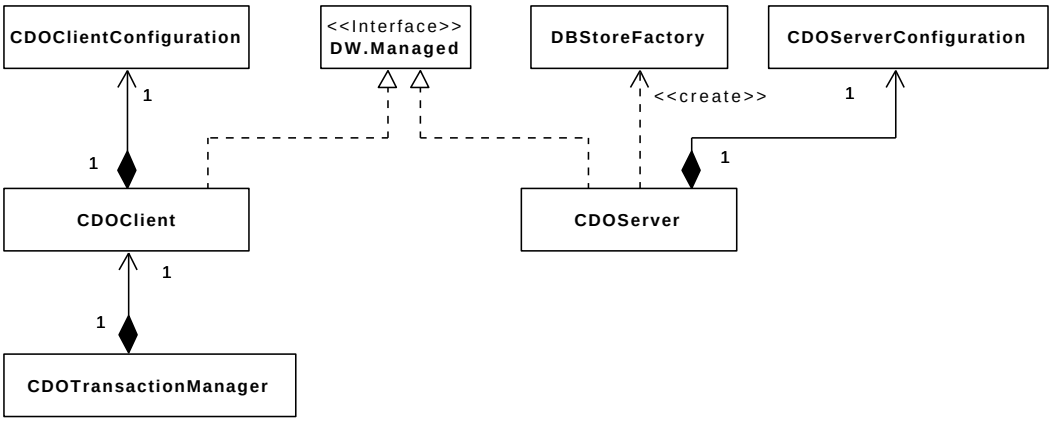


Figure 5.3. Core classes of the CDO server and client components

Listing 5.3. The configuration file indicates that a CDO server will be activated on TCP port 2036 and it provides a CDO repository with name 'EnterprisePerformanceModel'. In addition, it can be seen that a H2[4] database engine is used. A complete CDO server configuration reference is provided by [Eclipse Foundation, 2014b].

To meet FR1, FR2, and NFR1, *diagnoseIT* adopts this configuration mechanism. On the one hand this makes it possible to reuse existing configurations files and on the other hand, existing knowledge how CDO servers are configured can be reused. To keep the configuration and coding experience as similar as possible, the CDO source code was examined for reusable parts. Reuse was possible in two ways. First, direct reuse of certain components. Second, components must be cloned and adjusted. The main reason for adjustments was a lacking OSGI environment and the thus resulting malfunction, misconfiguration of certain components.

The core classes to initialize the CDO environment are depicted in Figure 5.3. As shown, *CDOServer* and *CDOClient* are configurable by additional configuration classes (*CDOClientConfiguration* and *CDOServerConfiguration*). The server configuration is mainly needed to make the server configuration file known to the *CDOServer* implementation. *CDOClientCongfiguration* provides information how to connect to the *CDOServer*.

To ensure a proper integration in the application life-cylce, the *DW.Manged* interface is implemented. To simplify transaction handling, a *CDOTransactionManger* is available. This is used to create, modify, and commit transactions. To reduce the amount of created transactions, new transactions are only created by an explicit statement. Otherwise, an existing transaction is reused. *DBStoreFactory* provides necessary implementations which match the *store* property of the server configuration. An example, how a fully functional CDO environment is initialized, started, and stopped, is shown by Listing 5.4.

[4]http://www.h2database.com/html/main.html

Listing 5.4. CDO Environment Usage

```
1  //1. CDOServer creation and start
   CDOServerConfiguration serverConfiguration = new CDOServerConfiguration();
3  serverConfiguration.setConfigurationFile("cdo-server.xml");
   CDOServer server = new CDOServer(serverConfiguration);
5  server.start();

7  //2. CDOClient and  CDOTransactionManager creation
   CDOClientConfiguration clientConfiguration = new CDOClientConfiguration();
9  clientConfiguration.setConnector("localhost:2036");
   clientConfiguration.setRepository("EnterprisePerformanceModel");
11 CDOClient client = new CDOClient(clientConfiguration);
   CDOTransactionManager transactionManger = new CDOTransactionManager(client);
13
   //Initialize a transaction, Create a new resource and add a new SoftwareModule
15 CDOTransactionContext context = transactionManger.createTransaction();
   CDOResource myResource = context.getTransaction()
17         .getOrCreateResource("MySoftwareModule");
   myResource.getContents()
19         .add(SoftwareModuleFactory.eINSTANCE.createSoftwareModule());

21 //Commit to ensure the new SoftwareModule is persisted
   context.commit();
23
   //stop server and transactionManager
25 transactionManger.terminate();
   server.stop();
```

5.3.3 Model Repositories

FR2 expresses the need for a stand-alone model repository to store the Enterprise Performance Model (EPM). This FR basically consists of two parts. First, the already presented stand-alone CDO environment (see 5.3.2). Second, a model repository implementation utilizing the CDO functionalities. As depicted in Figure 5.1, a repository is the only component with access to the data stores.

To ensure a proper separation of concern, the overall model repository is subdivided in several sub-repositories. The separation is inherited from the EPM itself. Consequently, this results in the following sub-repositories: *ResourceEnvironmentRepository*, *SoftwareModuleRepository*, *AllocationRepository*, and *TraceRepository*. Each sub-repository provides *CRUD* functionality for the corresponding part of the EPM. A rough overview of the core classes, a repository implementation consists of, is depicted in Figure 5.4. For each sub-repository it is mandatory to implemented the *IModelRepository* interface. Otherwise, the automatic bootstrap process (see 5.3.1) is not able to detect and initialize it.

Model repositories can access a *ContextManager*, which is the premise, to fulfil NFR3. The *ContextManager* always provides a valid *Context* for the current invocation of the model repository. This means, conversely, that each invocation of a model repository

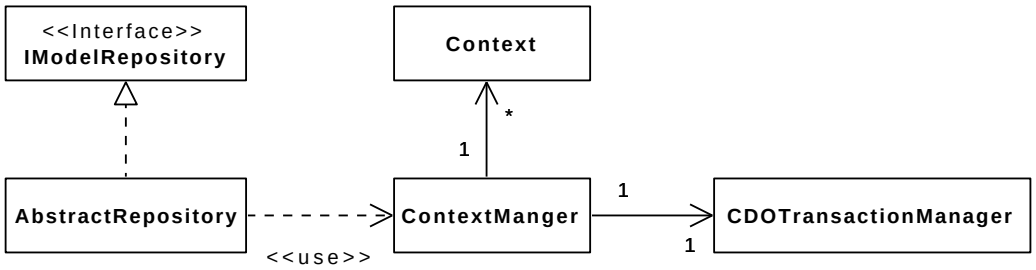


Figure 5.4. Core classes of the model repository component

has to be wrapped in a certain *Context*. Corresponding functionality is provided by the *ContextManager*. The provided *Context* supplies further metadata, e.g, which software module should be updated.

To support FR3, the *TraceModelRepository* has additional functionalities to persist execution traces in a *NoSQL* database. Currently, this is the in Section 5.2.5 introduced MongoDB.

5.3.4 System Model Maintenance Interface (SMMI)

As indicated in Section 2.2, the purpose of the SMMI is to publish a common API to maintain the EPM model repository. As depicted in Figure 2.1, the SMMI has two usage areas. On the one hand, it serves as access point to the EPM model repository for other, but internal *diagnoseIT* components. On the other hand, it provides functionality to populate the EPM model repository with new data, provided by foreign APM tools. From the perspective of APM tools, the current implementation is mainly designed to support updating the EPM model repository, rather than retrieving data. Internal *diagnoseIT* components, however, gain unlimited access to EPM instance.

5.3.5 Web Resources

JAX-RS defines a resource as Java class that uses JAX-RS annotations to implement a Web resource. A resource class is a Plain Old Java Object (POJO) with at least one *@Path* annotation. The *@Path* annotation defines a relative Uniform Resource Identifier (URI) path. The base URI depends on the deployment context of the web application [Hadley and Sandoz, 2009]. The *diagnoseIT* framework contains currently two resource classes. *ApplicationResource* and *SystemModelMaintenanceResource*. The first provides functionality for example, to register an adapter, or to check whether the application is available. The latter is the Web resource which servers as entry to the SMMI for APM tools.

An excerpt of the *SystemModelMaintenanceResource* is shown in Listing 5.5. As depicted the class, as well as the *update* method are annotated with *@Path*. In combination with

the *applicationContextPath*, taken from the application configuration (see Listing 5.1), this results in *http://server:8080/diagnoseIT/systemModelMaintenance/update* as endpoint to invoke the update method over the Internet. Furthermore, the listing shows that the method has to be invoked as POST request and consumes data serialized with JSON. Besides the *JAX-RS* functionality, it can be seen how HK2 injects the actual *SystemModelMaintenanceService* and how service invocations are wrapped in a certain *Context*. The current *Context* always depends on a valid session identification, provided by the user. Using the Jersey framework as JAX-RS implementation enables compliance with FR6.

Listing 5.5. Excerpt of *SystemModelMaintenanceResource*

```
@Path("/systemModelMaintenance")
public class ModelMaintenanceResource {
    @Inject
    private SystemModelMaintenanceService maintenanceService;
    @POST
    @Path("/update")
    @Consumes(MediaType.APPLICATION_JSON)
    public void update(final UpdateRequest request,
                       final @Suspended AsyncResponse response) {
        ContextManager.executeInContext(request.getSessionID(), () -> {
            response.resume(maintenanceService.update(request));
        });
    }
}
```

5.3.6 APM Tool Integration – Adapter

The final step while implementing the *diagnoseIT* framework was to implement the APM tool integration, as described in Section 2.2.3. In analyzing the requirements for the integration interface, it turns out that it is beneficial to provide an extended integration mechanism. This is because different APM tools can be connected to *diagnoseIT*, and each tool might have its on data format. Hence, every tool has to transform its data, to comply with the *diagnoseIT* format. To simplify the use of the *diagnoseIT* framework, an extensible integration adapter was implemented. The resulting requirements for the adapter are FR5-9. In the following, a brief introduction of the adapter's internal functioning is given.

An excerpt of the internal core classes is provided by Figure 5.5. The core component of each *Adapter* is the *Runtime* which is responsible for ensuring that all further components are properly created and initialized. As depicted, a *Runtime* is individual configurable by a *RuntimeConfiguration*. The configuration is, as for the *Application* (5.3.1), provided as YAML file. Like shown in Listing 5.6, a *RuntimeConfiguration* provides information how to connect to a certain *diagnoseIT* application as well as a context which is needed for interactions with the System Model Maintenance Interface (SMMI). While the *Adapter is starting*, the *identification* property is used to log on to the *Application*. If the *identification* is known, the *Application* returns a valid session identifier. This is an initial implementation of FR9 and

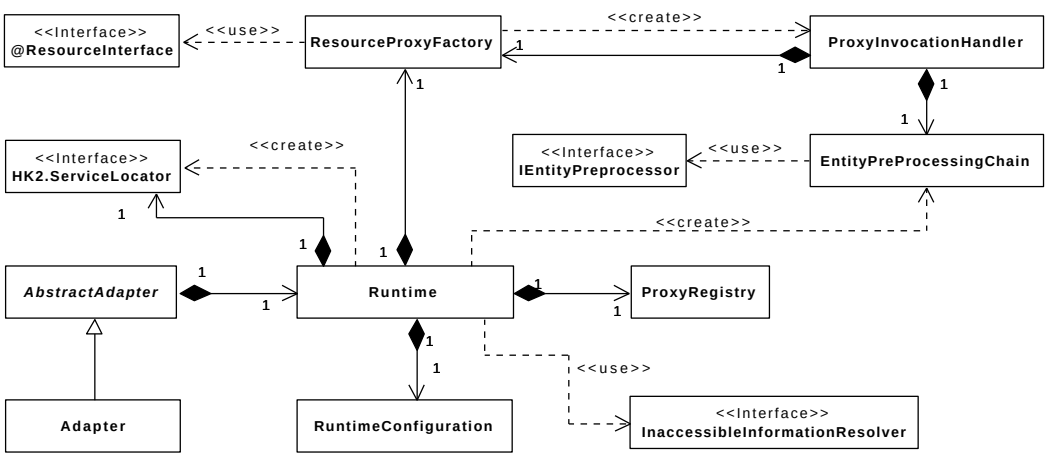


Figure 5.5. Core classes of the adapter component.

might be enhanced in the future.

When a *Runtime* is initializing, it creates proxies which enable the communication with the *Application*. All classpath classes annotated with *@ResourceInterface* are collected and the *ResourceProxyFactory* creates corresponding proxies. All generated proxies are subsequently available in the *ProxyRegistry*. As Figure 5.5 indicates, proxies can access an *EntityPreProcessingChain*. With respect to FR7, this chain is executed along with each proxy invocation. The *EntityPreProcessingChain* can be equipped with an arbitrary amount of *IEntityPreprocessors*.

The *Runtime* uses an *InaccessibleInformationResolver* interface. The purpose of this interface is to provide an optional callback mechanism whereby the *Adapter* can request additional information from a connected APM tool. Hence, the *InaccessibleInformationResolver* is a summarizing interface which currently comprises *ITypeResolver* and *IResourceEnvironmentResolver*. The first is used to gather additional information for a certain type, e.g., which interfaces it implements or if it is annotated. The latter resolves certain parts of a resource environment, e.g., resolve a *Node* to a corresponding name (see Section 4.2.2).

Listing 5.6. Example Runtime Configuration (adapter-runtime-configuration.yml)

```
identification: "1"
endpoint: "http://localhost:8080/diagnoseIT"
toolName: "MonitoringTool"
classpathScanning: ["de.unistuttgart"]
context:
  module: "JPetstore"
  moduleVersion: "1"
  resourceEnvironment: "JPetstoreEnvironment"
```

Table 5.1. Extractable EPM Model Entities

SoftwareModule			ResourceEnvironment		
	Default	Supported		Default	Supported
Type	✓	✓	*Node*	-	✓
Operation	✓	✓	*ExecutionEntity*	-	✓
Signature	✓	✓	*JVM*	-	✓
FrameworkElement	-	✓			

5.3.7 Enterprise Performance Model (EPM) Element Extraction

In the scope of this thesis, a so-called *TraceUpdateRequestPreProcessor* was developed. This preprocessor is always available in the *EntityPreProcessingChain* and is accordingly invoked for each entity which passes the *Adapter*. Benefit of *TraceUpdateRequestPreProcessor* is that depending on the input *Trace*, certain parts of the associated *SoftwareModule* as well as *ResourceEnvironment* are reconstructable. This becomes possible, because a *Trace* has detailed knowledge of *Operation* and *ModuleAllocation* (see Section 4.2.4). Table 5.1 gives an overview, which entities are extractable by default, and which can solely be extracted with further support of the APM tool. In order to ensure this support, the tools have to provide an implementation of the beforehand introduced *InaccessibleInformationResolver* interface. In order to resolve *Nodes*, *ExecutionEntities*, and *JVM*s, the tools receive the corresponding name of the entity.

A more complex task is to dissolve the affiliation of a *Type* to a certain *Framework*. To remove this task from APM tools, the *Adapter* provides a *FrameworkScanner*. In order to ensure correct functionality, the *FramworkScanner* is initialized with corresponding meta-data. The meta-data comprises information about the *Framework*, more specifically name, classpath pattern, and the framework's classification, as well as for the associated *FrameworkElements*, more specifically, name and recognition patterns. Currently, the *FrameworkScanner* supports a *Type's* inheritance hierarchy and class-level annotations as patterns to assign it a *FrameworkElement*. Nevertheless, the recognition patterns must be contributed from the APM tool, by implementing the *ITypeResolver* interface (see Table 5.1).

Detecting *FrameworkElements* is a three step process. First, all recognition patterns of a *Type* are requested from the APM tool. Second, the obtained recognition patterns are checked against the classpath patterns of all *Frameworks*. If there is a match, we can assure that this *Type* is at least somehow related to the *Framework*. As third and final step, the *Type* recognition patterns are compared with the initially provided *FrameworkElement* meta-data. If this step succeeds, the *Type* is henceforth linked with this *FrameworkElement*, accordingly identifiable as part of a *Framework*.

5.4 *diagnoseIT* Framework in Action

After the preceding technical descriptions, this section provides a brief usage example. The example demonstrates (i) how a new *Adapter* is created, configured and started, (ii) how the *IModelMaintenanceResource* is obtained and used, and (iii) the internal, but abstracted, control flow within the *diagnoseIT* framework. The example is from the perspective of an APM tool. From this it follows the assumption that a running *diagnoseIT* application instance is available at a certain endpoint (see Listing 5.2). To reuse the adapter runtime configuration from Listing 5.6, we assume a *diagnoseIT* instance is available at *http://localhost:8080/diagnoseIT*.

Lines 1-6 of Listing 5.7 show how a new *Adapter* is created,configured and started by using an *AdapterBuilder*. The builder provides convenience functionality to setup new adapters, e.g., define the configuration file and set the *InaccessibleInformationResolver*. An exemplary implementation of *InaccessibleInformationResolver* is illustrated in Listing 5.8. The provided sample implementation simply returns empty representations of the requested entities.

Listing 5.7. Simple example how *Adapter* is used to transmit a single *Trace*.

```
Adapter adapter = AdapterBuilder.newAdapter(Adapter.class)
        .configureFromFile("adapter-configuration.yml")
        .inaccessibleResolver(new MyInaccessibleInformationResolver())
        .build();

adapter.start();

TraceDTO traceDTO = TraceDTO.trace(start, end)
        .syncCallMessage()
        .from("A.methodA():void", 100, "jvm1", "server0")
        .to("B.methodB():String", 100, "jvm1", "server0")
        .syncReplyMessage().from("B.methodB():String", 100, "jvm1", "server0")
        .to("A.methodA():void", 100, "jvm1", "server0")
        .build();

IModelMaintenanceResource maintenanceResource = adapter.api().maintenanceResource();
maintenanceResource.update(new UpdateRequest().forTrace(traceDTO));

adapter.stop();
```

Sequence Diagram 5.6 illustrates the control flow after starting an *Adapter*. Internally, the *Runtime* initiates a registration process to log on to the system. To register, the *IApplicationResource* proxy asynchronously invokes the *ApplicationResource* REST API. The request is parameterized with a *ClientRegistrar*, which corresponds to a subset of the *RuntimeConfiguration* (Listing 5.6), consisting of context and identifier. As depicted, the *Runtime* blocks until it receives a session identifier in return as the identifier is mandatory for all further processing. Once the session identifier is available, it is attached to the *Runtime* and the *Adapter* is ready for use.

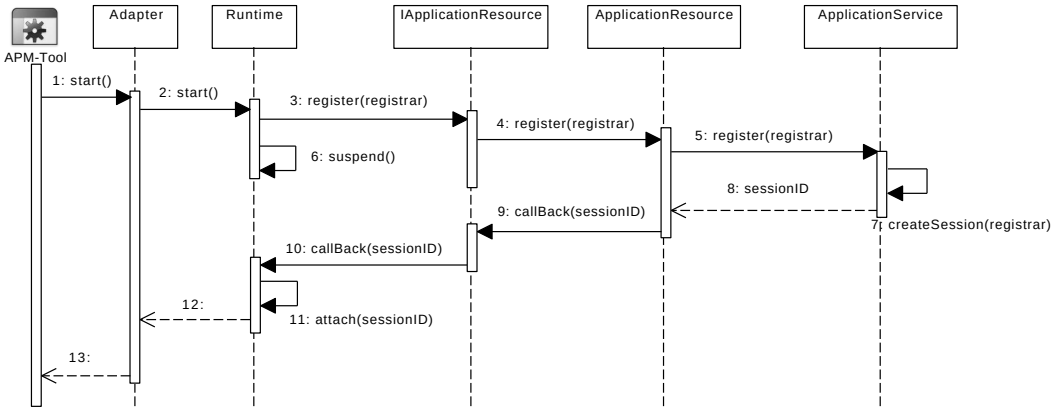


Figure 5.6. Control flow to register an *Adapter* at a *diagnoseIt* application.

How the *Adapter* is used to update the EPM, by sending a trace, is illustrated in lines 8-17 of Listing 5.7. Since the data transport is realized by DTOs, a new *TraceDTO* is constructed (lines 8-14). To construct traces as described in Section 4.2.4 a builder is provided. The final steps to transmit the recently created trace are (i) obtaining the *IModelMaintenanceResource*, (ii) wrapping the DTO in a *UpdateRequest* DTO, and (iii) invoking the update operation of *IModelMaintenanceResource* (lines 16-17).

Listing 5.8. Example *InaccessibleInformationResolver* implementation

```
1  private class MyInaccessibleInformationResolver extends AbstractInaccessibleInformationResolver {
       @Override
3      public NodeDTO resolveNode(String environment, String nodeName) {
           return NodeDTO
5                  .newBuilder(nodeName).build();
       }
7
       @Override
9      public ExecutionEntityDTO resolveExecutionEntity(String environment, String nodeName, String executionEntity) {
           return ExecutionEntityDTO.
11                 newBuilder(ExecutionEntityDTO.ExecutionEntityType.JVM, executionEntity, nodeName)
                   .build();
13     }

15     @Override
       public ResolvedType resolveType(String fullQualifiedClassName) throws TypeResolutionException {
17         return Utils.resolveType(fullQualifiedClassName);
       }
19 }
```

The by the *IModelMaintenanceResource.update()* initiated control flow is depicted in Figure 5.7. In order to increase the diagrams readability, certain steps were simplified and *UpdateRequest* is entitles as *Request*.

Whenever a new *Request* is available, the in Section 5.3.6 introduced *EntityPreProcessingChain* is executed. By default, the chain contains two preprocessors: *SessionIdDistributor* and *TraceUpdateRequestPreProcessor*. The purpose of *SessionIdDistributor* is, to enrich processed DTOs with the previous received session identifier. Proceeding from the previous description (5.3.7), the *TraceUpdateRequestPreProcessor* extracts further information from the trace and enriches the *Request* with it. The processed *Request* is subsequently transmitted to the server-side *SystemModelMaintenanceResource*.

Prior to the final update process, the *ContextManager* (5.3.3) has to ensure context awareness. For this, the *ContextManager* creates, depending on the session identifier, a new *Context* and activates it. As described in Section 5.3.3, the *Context* dictates which parts of which repository are updated. Once the *Context* is established, the SMMI implementation is invoked. The *SystemModelMaintenanceService* unwraps the DTOs from *Request*, transforms them in *CDOObjects* (4.3.2), and updates the corresponding repository. Since, up to four model repositories might be changed, Figure 5.7 indicates it as looping call to *IModelRepositoriy*. This comprises updates of *TraceRepository*, *SoftwareModulRepository*, *AllocationRepository*, and *ResourceEnvironmentRepository* (see Chapter 4). After updating the repositories, the previous valid *Context* is restored and the return value is send back to the *Adapter*.

Table 5.2 shows the exemplary content of the EPM at the end of the preceding described update process. As seen, it contains one *SoftwareModule* (JPetStore), one *ResourceEnvironment* (JPetstoreEnvironment), and one *AllocationModel*. The available *Types* of the *SoftwareModule* directly reflect the message definition made in Listing 5.7. This also applies to the *ResourceEnvironment*. In addition, the *AllocationModel* states, that a *SoftwareModule* JPetStore is executed by *ExecutionEntity* jvm1. Remarkable is the unique identifier *JPetStore@1::jvm1@server0* of this allocation, which is so descriptive that the entire allocation is recoverable.

Table 5.2. Exemplary content of the EPM in accordance with the described update process.

EnterprisePerformanceModel				
SoftwareModule				
Name	Version	Type		
JPetStore	*1*	Name	Operation	Signature
		A	*methodA*	*A.methodA():void*
		B	*methodB*	*B.methodB():void*
ResourceEnvironment				
Name		Node		
JPetstoreEnvironment		Name	ExecutionEntity	
		server0	*jvm1*	
AllocationModel				
Identifier				
JPetStore@1::jvm1@server0		SoftwareModule	ExecutionEntity	
		JPetStore	*jvm1*	

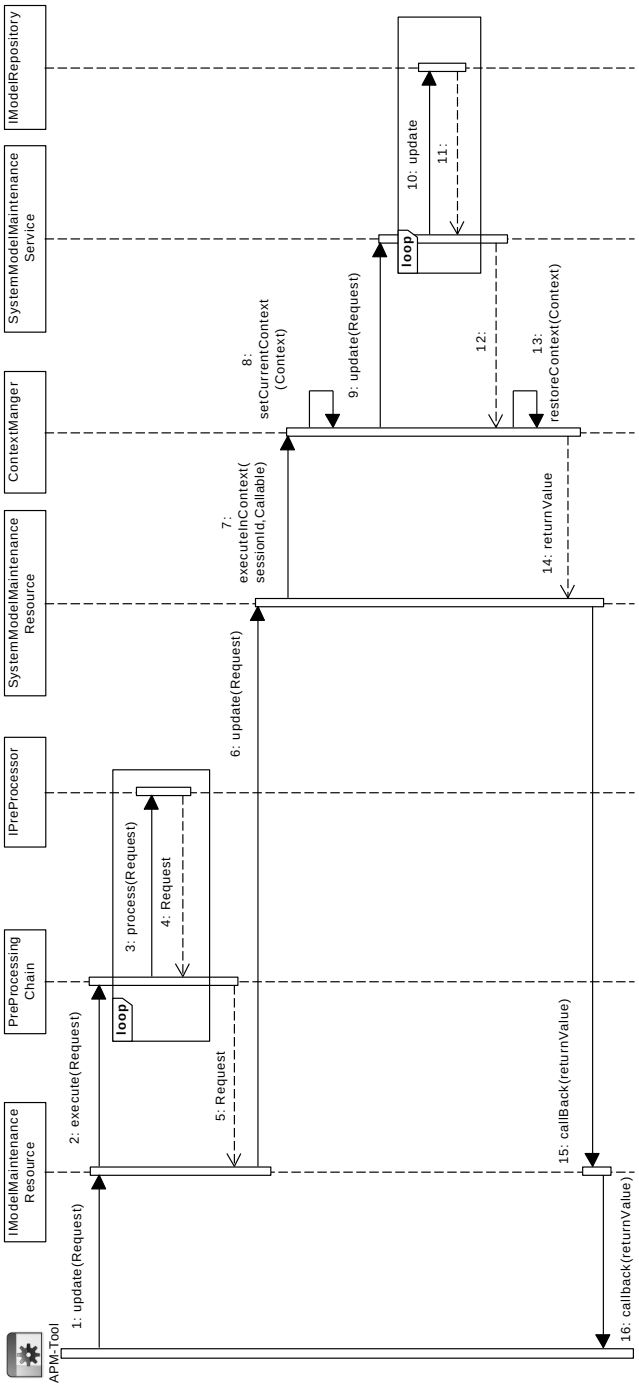


Figure 5.7. Control flow to update the EPM model repository.

Chapter 6

Evaluation

This chapter targets the evaluation of the *diagnoseIT* framework implementation, by answering the research questions raised in (Section 2.3). The research questions regarding the evaluation are formulated as follows:

- **RQ4:** Does the approach meet all previously specified requirements?
- **RQ5:** Is the approach applicable in real world scenarios?

In this thesis we employed the Goal Question Metric (GQM) approach as measurement mechanism in order to answer the research questions. Caldeira and Rombach [1994] proposed GQM as framework to transfer questions in measurable goals. The application of the GQM, in the scope of this thesis, is subject of Section 6.1. The results of the conducted proof-of-concept evaluation are presented in Section 6.2. Section 6.3 presents the results of the carried out lab experiment.

6.1 Evaluation Goals

This section describes how the GQM approach is applied in this thesis. According to Caldeira and Rombach [1994], the application of GQM results in a measurement model, which targets at certain issues for the interpretation of measurement data. The measurement model is divided into the following three levels: Conceptual level (Goal), operational level (Question), and quantitative level (Metric). A goal is defined for one of three types of measuring objects (products, processes, and resource) and from a certain viewpoint. Questions emboss the way how the achievement of a goal is performed. A metric defines data which enables answering one or more questions [Caldeira and Rombach, 1994].

The GQM goals defined for the *diagnoseIT* framework are presented in the following. To avoid name confusion with goals from Chapter 2, we refer them as *evaluation goals (EGs)*.

EG1: Assessing Functionality of the *diagnoseIT* Framework

As stated in Section 2.2, a big part of this thesis was to implement a prototype of the *diagnoseIT* framework. *EG1* is used to answer **RQ4** (*Does the approach meet all previously specified requirements?*). Thus, the purpose of EG1 is to evaluate the functionality of all developed components. The formal definition of EG1, as proposed by Caldeira and

Table 6.1. EG1: Assessing the functionality of the *diagnoseIT* prototype, from a user's viewpoint.

Goal	EG1	
	Purpose	Assessing the
	Issue	functionality of
	Object	the *diagnoseIT* prototype
	Viewpoint	of a users viewpoint
Question	Q1.1	Does the *diagnoseIT* prototype meet all previously specified requirements?
Metrics	M1	Functioning of components
Question	Q1.2	Up to what granularity is Kieker able to serve the EPM?
Metrics	M2	Enterprise Performance Model (EPM) coverage

Rombach [1994], is presented in Table 6.1. EG1 comprises two questions. The first is to evaluate if the prototype works as expected. The second to evaluate to what extent the APM tool Kieker is able to equip the EPM. This EG is addressed by the in Section 6.2 presented proof-of-concept evaluation.

EG2: Assessing Scalability of System Model Maintenance Interface (SMMI)

With respect to the goals of this thesis, the SMMI is the core component to interact with a *diagnoseIT* application. Since the SMMI is used from a various amount of APM tools, it has to scale along with the emerging workload. Hence, EG2 targets **RQ5** (*Is the approach applicable in real world scenarios?*). The formal definition of EG2 is depicted in Table 6.2.

As shown, EG1 defines three questions. For each of the scalability attributes responsiveness, memory consumption, and CPU utilization a corresponding combination of question and metric was formulated. This EG is addressed by the in Section 6.3 presented lab experiment.

6.2 Proof-of-Concept Evaluation

In the proof-of-concept evaluation we deploy a, with Kieker instrumented, Java-base enterprise application to an application server. A developed analysis tool receives the monitoring data and transfers it to a *diagnoseIT* instance. The evaluation methodology is presented in Section 6.2.1, followed by the experimental setting in Section 6.2.2. The evaluation results are presented in Section 6.2.3.

Table 6.2. EG2: Assessing the scalability of the System Model Maintenance Interface (SMMI) in lab experiments.

Goal	EG2	
	Purpose Issue Object Viewpoint	Assessing the scalability of the System Model Maintenance Interface (SMMI) in lab experiments
Question	Q2.1	How responsive are model repository updates with increasing load?
Metrics	M3	Response time
Question	Q2.2	How memory intensive are model repository updates with increasing load?
Metrics	M4	Memory consumption
Question	Q2.3	How CPU intensive are model repository updates with increasing load?
Metrics	M5	CPU utilization

6.2.1 Evaluation Methodology

As described, this proof-of-concept evaluation is mainly used to evaluate the functionality of the developed *diagnoseIT* framework. The evaluation is conducted as follows. A *diagnoseIT* application is started. In addition, on a different physical machine, a Java-based enterprise application is deployed to a Tomcat[1] application server. The deployed application contains a Kieker-based instrumentation. As described in the foundations (3.7.1), we used *Kieker.Analysis* to obtain and process the monitoring data of the application. To establish a connection between *Kieker.Analysis* and *diagnoseIT*, a so-called *DiagnoseITAdapterPlugin* was integrated in Kieker's pipe and filter structure. Hence, Kieker serves as APM tool to serve *diagnoseIT*. The *DiagnoseITAdapterPlugin* receives the processed monitoring data and transmits it to *diagnoseIT*, as described in Section 5.4. The metrics of EG1 are predominantly evaluated qualitative, by demonstrating the applicability of the developed approach.

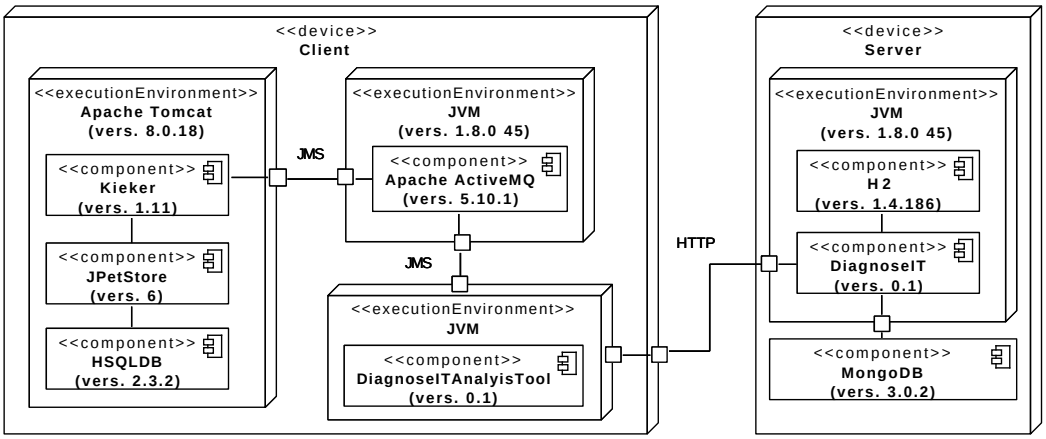


Figure 6.1. Technical infrastructure of the proof-of-concept evaluation

6.2.2 Experimental Setting

This section describes the employed software and hardware infrastructure. The technical infrastructure is depicted in Figure 6.1. The figure gives an overview of software and hardware components, as well connections. As depicted, the infrastructure comprises two physical machines, we refer them as client and server. The technical details, regarding CPU, RAM, Operating System, and the used Java version, are provided in Table 6.3. As shown in Figure 6.1, the client device contains three software execution environments (version numbers are depicted as well). A Tomcat application server, including the instrumented JPetstore application, *Kieker.Monitoring*, and a HSQL[2] database, which is mandatory for JPetStore. An Apache ActiveMQ Java Message Service (JMS) server is used to establish the Kieker monitoring log/stream (see Figure 3.7). The *DiagnoseITAnalysisTool* processes the monitoring data and transfers it to *diagnoseIT*.

The *diagnoseIT* application is deployed to the server device. As show, the H2 database, which serves as data storage for the Enterprise Performance Model (EPM), is embedded into the same execution environment as *diagnoseIT*. The server device also hosts a MongoDB instance.

JPetStore Application

JPetStore is an open source J2EE web application, representing a Web shop. As the name suggests, it offers animals. Basically JPetStore is a sample application to demonstrate the

[1]http://tomcat.apache.org/
[2]http://hsqldb.org/

Table 6.3. Technical details of the employed physical machines.

	Client	Server
CPU	Intel Core i5-3470 3.20GHz x4	Intel Core i5-3470 3.20GHz x4
RAM	4GB	4GB
OS	Linux (Mint) 3.16.0-38-generic x86_64	Linux (Mint) 3.16.0-38-generic x86_64
Java	1.8.0_45	1.8.0_45

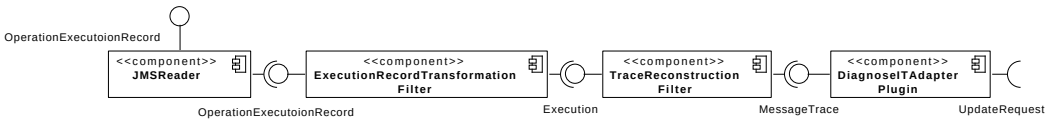


Figure 6.2. Kieker pipe-and-filter structure of the DiagnoseITAnalysisTool

usage of MyBatis 3[3], Spring 3[4] and Stripes[5]. As usual for Web shops, JPetStore provides a HTML Web interface to interact with the application. Therefore typical functionalities, like signing in/off, inspecting products, add/remove products to a shopping cart, and finally purchasing an order, are provided. Technically, JPetStore is build as 3-tier architecture: presentation layer, application layer, and persistence layer, as described in Section 3.4 [MyBatis, 2015].

To receive monitoring data we instrumented JPetStore as follows. As described in Section 3.7.1, *Kieker.Monitoring* can provide different kinds of monitoring data. In our proof-of-concept evaluation we were only interested in execution traces. Accordingly, *Kieker.Monitoring* was configured to monitor all method invocations within the sample application. The thus requested *OperationExecutionRecords* (3.7.1) are passed to JMS and received from the *DiagnoseIIAnalysisTool*.

DiagnoseITAnalysisTool

Based on *Kieker.Analysis*, we developed the *DiagnoseITAnalysisTool*. The purpose of this tool is to receive, process and forward the monitoring data. According to the description in Section 3.7.1, we designed a pipe-and-filter structure as depicted in Figure 6.2. The chain consists of four Kieker plugins:

- *JMSReader*
  A Kieker plugin to receive *OperationExecutionRecords* from JMS and subsequently dispatch them to the pipe-and-filter chain. This filter is already provided by Kieker.

- *ExecutionRecordTransformationFilter*

[3]https://mybatis.github.io/mybatis-3/
[4]http://spring.io/
[5]https://stripesframework.atlassian.net/wiki/display/STRIPES/Home

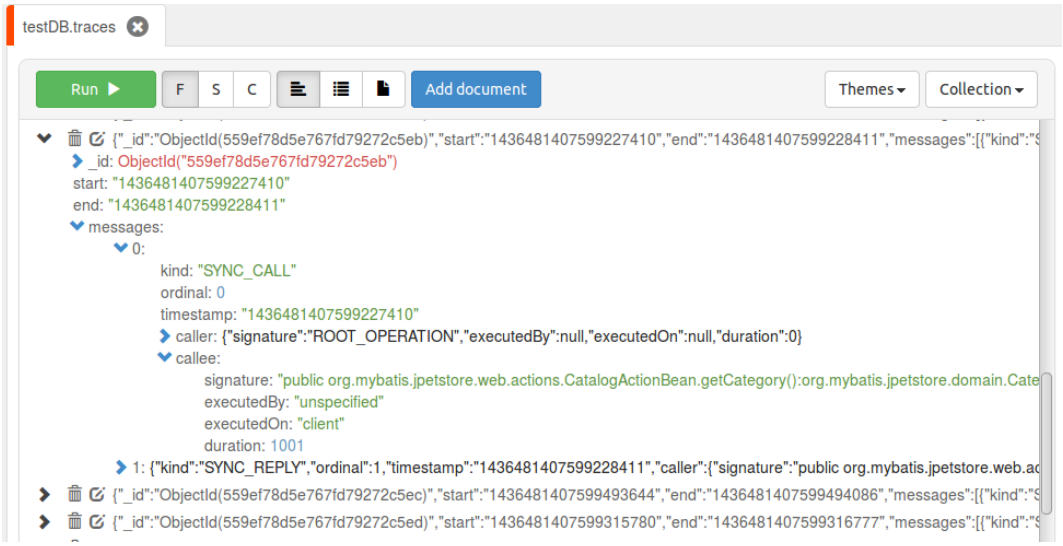


Figure 6.3. Content of the *TraceRepository* within Mongo Management Studio (MMS).

The purpose of this filter is to transform *OperationExecutionRecords* in *Executions*. An *OperationExecutionRecord* is a flat representation of an *Executions* what means that all available monitoring data is represented as simple types (e.g., string, integer, long, etc.). The *Execution* representation in contrast, enables the succeeding execution trace reconstruction. This filter is already provided by Kieker.

- *TraceReconstructionFilter*
  The *TraceReconstructionFilter* assembles all incoming *Executions* to a valid *MessageTrace*. The Kieker *MessageTrace* meta-model was already introduced in Section 3.7.2. This filter is already provided by Kieker.

- *DiagnoseITAdapterPlugin*
  The purpose of the *DiagnoseITAdapterPlugin* is to transform an incoming *MessageTrace* into a *diagnoseIT* compliant representation. To transfer the trace data to *diagnoseIT*, the plugin internally utilizes the *diagnoseIT Adapter* as described in Section 5.4.

6.2.3 Experiment Results

In order to gather the results of the poof-of-concept application, two additional tools were needed. To inspect the dynamic data, i.e., the received execution traces, which are stored in a MongoDB instance we used the Mongo Management Studio (MMS)[6]. MMS connects to a running MongoDB and provides a Web interface to inspect the database. Since CDO is

[6]http://www.litixsoft.de/mms/

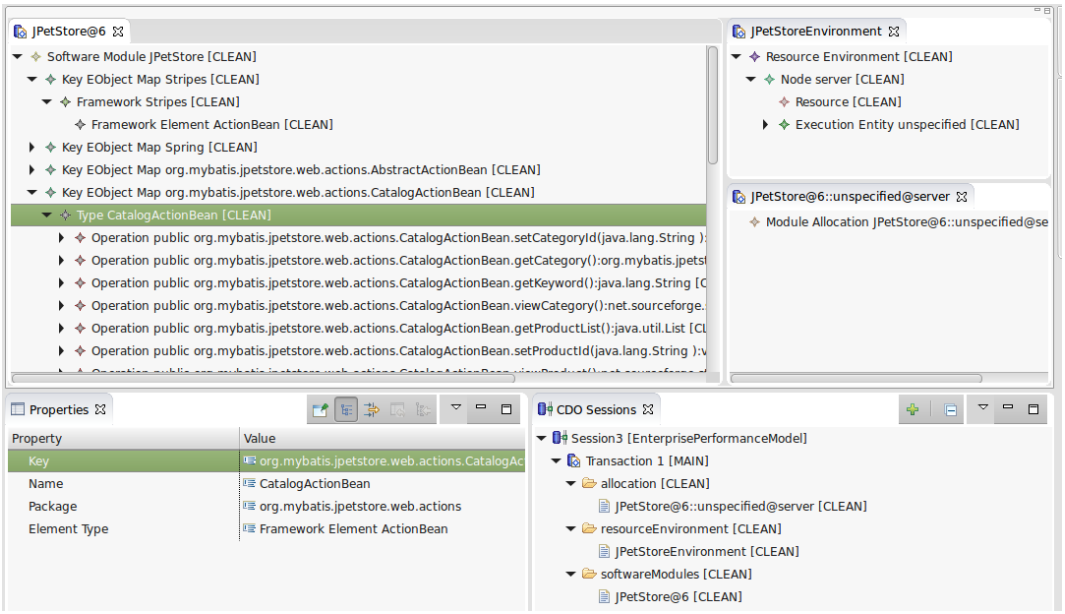


Figure 6.4. Repository overview within Eclipse IDE.

a member of the EMP, the Eclipse IDE offers the functionality to connect to running CDO servers and present the content of the repositories. This feature was used to inspect the content of the EPM.

Figure 6.3 depicts an excerpt of the *TraceRepository*, after the JPetStore Web application was used to run through several use cases, like inspecting the products catalog, searching for products, and finally purchasing an order. As shown, the trace reflects the trace meta-model as defined in Section 4.2.4. Based on the provided example trace we can infer that there was an execution trace, initiated by the user (ROOT_OPERATION), to a certain component, called *CatalogActionBean*. Additionally, it is depicted that the *CatalogActionBean* is executed by an unspecified *ExecutionEntity*, which executes on a *Node* called client. With respect to the *ResourceEnvironement*, with Kieker we are able to determine the name of the *Node*, but we are not able to determine any information related to the *ExecutionEntity*. That is why *ExecutionEntity* is unspecified.

As described in Section 5.3.7, this information is used to extract additional entities for the *SoftwareModuleRepository* and the *ResourceEnvironmentRepository*. The contents of which are depicted by Figure 6.4. In the top left corner, it is shown that the *SoftwareModuleRepository* consists of two *Frameworks* (Stripes and Spring) and several *Types*. As illustrated, *CatalogActionBean* (Figure 6.3), comprises several *Operations*, e.g., setCategory(), getCategory(), viewCategory(), etc. Further properties of *CatalogActionBean* are presented in the bottom left corner. Besides name and package it can be seen that *CatalogActionBean*

Table 6.4. Summary which parts of the Enterprise Performance Model are resolveable with Kieker.

Enterprise Performance Model			
Software Module	Covered	Resource Environment	Covered
Type	✓	Node	●
Operation	✓	Node Link	✓
Operation Link	✓	Execution Entity	✗
Signature	✓	JVM	✗
Framework		Resource	✗
Allocation	Covered	Trace (Behaviour)	Covered
Software Module	✓	Message	✓
Execution Entity	✗	Message Kind	✓
		Duration	✓
		Allocation	✗
		Operation	✓

is an *ActionBean* and is consequently part, or user, of the Stripes *Framework*. The top right corner shows the content of *ResourceEnvironmentRepository* and *AllocationRepository*. Due to the employed technical infrastructure (6.1), the *AllocationRepository* contains only one allocation and the *ResourceEnvironment* the corresponding *Node* and *ExecutionEntity*. As depicted neither the *Node* nor the *ExecutionEntity* have any resource attached. The reason for this is the same as for the unspecified *ExecutionEntity*, Kieker does not provide this information. At least not by default. Kieker already provides the *Kieker.Monitoring* methods as Java Management Extensions (JMX) service, but rather to configure the monitoring than requesting further data [Kieker Project]. Possible solutions for this shortcomings are discussed in the future work section (8.3).

Result Summary

With respect to the addressed evaluation goal EG1 (*Assessing functionality of the diagnoseIT framework*) with the enclosed questions Q1.1 (*Does the diagnoseIT prototype meet all previously specified requirements?*) and Q1.2 (*Up to what granularity is Kieker able to serve the* EPM*?*) we draw the following conclusions:

- Q1.1: From a functional point of view, the proof-of-concept evaluation has shown that all components of the implemented *diagnoseIT* framework work as envisioned in Chapter 2 and described in Chapter 5. The prototype provides an adapter implementation as integration facility for third-party APM tools, in order to supply *diagnoseIT* with monitoring data which is subsequently processed and, depending on the type of data, stored in different databases. The completeness of the EPM strongly depends on the data provided by the APM tool. If monitoring data is provided in form of traces, *diagnoseIT* is

instantly able to recover architectural information of the system under analysis (SUA). With respect to information concerning the resource environment, which executes the SUA, the evaluation revealed that *diagnoseIT* is highly dependent on the capabilities of the APM tool.

▻ Q1.2: This evaluation revealed, that the Kieker APM tool is already able to provide large parts of the EPM. The main reason for this is that during the requirements analysis for the EPM, the internal Kieker meta-models were analyzed. Consequently, certain parts, especially the trace meta-model, of the EPM are inspired by the Kieker meta-models. Table 6.4 provides an overview of those parts of the EPM which are resolvable by Kieker. For each top-level entity of the EPM it is noted if the entity is (i) ✓resolvable, (ii) ● partially resolvable, or (iii) ✗ not resolvable.

As can be seen, all entities of the software module meta-model are resolvable out of the box. As described in Section 5.4, the meta-data for frameworks is provided separately and thus the framework entity is not considered in this evaluation. However, as presented in the results, types are associated with frameworks. This was enabled by adding the JPetStore source code to the classpath of the *DiagnoseITAnalysisTool*. Hence, it was possible to resolve the inheritance hierarchy and possible annotations for given full qualified class names on the client machine. In terms of the resource environment meta-model, the name of a node and node links are resolvable. Execution entities, Java Virtual Machines (JVMs), and information of utilized resources are absent. The irresolvable execution entities follows an incomplete allocation meta-model as a consequence. To ensure a sound and usable EPM, *diagnoseIT* provides a so-called *unspecified* generic execution entity. As shown in Figure 6.3, the trace model is also affected by the absent execution entity.

6.3 Lab Experiment

This section presents the results of applying the *diagnoseIT* approach in a lab experiment. To assess the scalability of the implemented System Model Maintenance Interface (SMMI), *diagnoseIT* is put under load and the performance metrics are measured. The employed evaluation strategy is presented in Section 6.3.1, followed by the description of the experimental setting in Section 6.3.2. Section 6.3.3 presents the results.

6.3.1 Evaluation Methodology

To assess the scalability of the implemented approach we decided to conduct lab experiments to ensure controlled, repeatable conditions. Definition 3 provides a definition of scalability, by Smith and Williams [2002], which we incur in this thesis.

Definition 3 *Scalability is the ability of a system to continue to meet its response time or throughput objectives as the demand for software functions increases [Smith and Williams, 2002].*

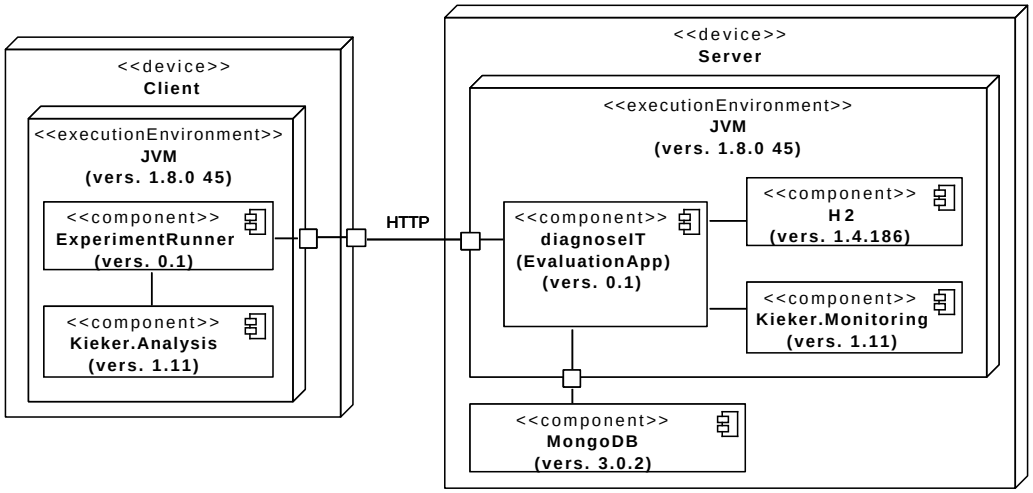


Figure 6.5. Technical infrastructure of the lab experiment.

The experiments were conducted as follows. The *diagnoseIT* application was connected to a dynamic Kieker trace generator. To generate synthetic traces, the trace generator uses previously recorded Kieker monitoring data from the JPetStore sample application. Depending on the configuration of the generator, different scenarios can be simulated. In this experiment we distinguish between two scenarios.

First, increased intensity scenario. This means that in a fixed time period the amount of generated traces is increased continuously, but a trace's content is always the same. Second, increased data variation scenario. That means that the amount of generated traces is constant, but the internals of the traces are modified. An example of this, with respect to the Kieker trace meta-model (see 3.7.2), is modifying the *AllocationComponent* of each execution. Hence, a distributed system is simulated. For *diagnoseIT* this results in constant updates of the *ResourceEnvironmentRepository* and the *AllocationRepository*. To obtain the required measurement data for metrics M3 (response time), M4 (memory consumption), and M5 (CPU utilization), the *diagnoseIT* application is instrumented with Kieker itself.

6.3.2 Experimental Setting

This section describes the employed software and hardware infrastructure which was used to conduct the lab experiment. Figure 6.5 outlines the technical infrastructure, technical details of the used physical machines are equivalent to the ones of the proof-of-concept evaluation and can consequently be taken from Table 6.3.

The *diagnoseIT* instance to be monitored is along with *Kieker.Monitoring* deployed to the server device. As shown in Figure 6.5, *diagnoseIT* is deployed as *EvaluationApplication*, which

corresponds to a slightly modified version of *diagnoseIT*. The client machine comprises *Kieker.Analysis* and a so-called *ExperimentRunner*. Benefit of the *ExperimentRunner* is that it has capabilities to automatically execute predefined experiment plans.

Data Preparation

As described in Section 6.3.1, the lab experiment was conducted with generated synthetic trace data. To gather the initial dataset, we reused the experimental setting of the proof-of-concept evaluation (Section 6.2.2) and used the JPetStore sample application for approximately 2 minutes. Following this, the monitoring data was inspected to gain detailed insights. The original monitoring data consists of 1905 records (*OperationExecutionRecord*), 1248 traces, 18 types, and one resource environment. From these data we can deduce that the average size of a trace within the JPetStore application comprises $\approx 1,5$ records. To ensure from the outset a slightly higher load, we chose a medium-sized trace, comprising 11 records, as input trace for the trace generation process. Building on this single trace, we defined that the root dataset for all further experiments is 1000 times this trace. Additionally, we defined that this dataset comprises 200 types, 2 resource environments, and it has a total duration of 180 seconds.

EvaluationApplication

The *EvaluationApplication* is a three features enhanced version of *AbstractApplication* (see Section 5.3.1). Due to the extensibility mechanisms of *diagnoseIT* the required enhancements were possible without modifying the *diagnoseIT* code base. In order to advice *Kieker.Monitoring* to sample CPU utilization, memory consumption, and garbage collection, we added additional configuration options by extending the default *ApplicationConfiguration*. In addition, we provided a instrumented implementation of *SystemModelMaintenanceService*, by modifying the used *HK2.Binder*. The last addition was a so-called *ExperimentResource* Web Resource (see Section 5.3.5). The resource provides a REST API for the *ExperimentRunner*, comprising functionality to store experiment results, delete databases, and to reboot the entire *EvaluationApplication*.

ExperimentRunner

To ensure controlled, repeatable experiments, we implemented a *ExperimentRunner* to execute a predefined experiment plan. An exemplary experiment plan is provided by Listing 6.1. Basically a plan is divided in three sections. First, general configuration properties which are valid for all experiments. However, they might be overwritten. Second, a definition for an experiment which is always executed in advance to ensure the JVM is initialized and all classes are loaded. Third, an arbitrary list of experiment definitions. The internal functioning of the *ExperimentRunner* is depicted in Figure 6.6 and explained in the following.

```
1  adapterConfig: "experiment-adapter.yml"
   pluginAdapterConfig: "plugin-adapter.yml"
3  inputDir: "trace"

5  rampUpExperiment: "Also a experiment definition"
   experiments:
7      - name: "60sec2000Traces100Types2Environments"
         duration: 60
9        traceCount: 2000
         types: 100
11       typeDistribution: [70,40]
         environmentDistribution: 2
13       repeats: 3
```

Listing 6.1. Exemplary experiment plan to simulate a three times repeated experiment which runs 60 seconds, generates 2000 execution traces, 100 types are spread over the traces, and the traces are distributed on 2 resource environments.

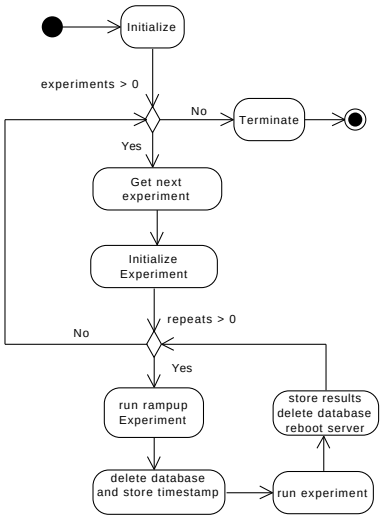


Figure 6.6. Activity diagram to visualize the functioning of the *ExperimentRunner*.

The first step is to initialize the *ExperimentRunner*. In this step, the experiment plan is loaded and based on the provided configuration file (Listing 6.1 line 1) a *diagnoseIT Adapter* is created. The *Adapter* is needed to access to previous introduced *ExperimentResource*. In a next step it is checked whether more not yet executed experiments are available and if so, the next one is taken and initialized. While initializing a new experiment, the later used synthetic traces are generated. The generation process comprises five steps:

1. The first step is to load the previously determined initial trace which is provided as Kieker monitoring data. The location of the monitoring data must be provided along with the experiment plan.
2. The *traceCount* property defines the total amount of generated traces. In the scope of the generation process this means how often steps 3-5 are repeated.
3. This step targets the *types* and *typeDistribution* properties. *Types* define the amount of types which should be distributed over the chosen traces, *typeDistribution* defines the distribution, the emergence percentage, of types over all traces. At the start of monitoring of a software system, each type is considered as new, but after a certain time most are known. This scenario can be simulated by setting the *typeDistribution* property. For example, a *typeDistribution* of [70,40] means that 70% of all types are distributed to the first 40% of all traces. With respect to the example experiment plan (6.1) this means accordingly that 70 types are distributed to traces 1-800, the remaining 30 types to traces 801-2000.
4. After the types are distributed, this step handles the *environmentDistribution* property.

The value of the property defines the amount of possible hosts. Each record of a trace gets randomly a new host assigned.

5. The last step is to modify the execution time of each *OperationExecutionRecord*. Depending on the provided *duration* property and the overall amount of *OperationExecutionRecords* the duration for each record is calculated. This modification is needed to simulate an increasing amount of traces in the same time period.

To ensure sound experiment results, all generated synthetic traces are stored and the same set of traces is reused for all repeats of the experiment. As shown in Figure 6.6, as next step the *ExperimentRunner* checks if the experiment should be executed one more time. The total number of repeats is defined by the *repeats* property of the experiment description.

As described before, the experiment plan contains a special, so-called, rampup experiment description. Formally, this is exactly the same description as for all experiments, but with a different purpose. The rampup experiment is carried out before each regular experiment execution, to ensure a proper initialized, ready (i) *diagnoseIT* application, and (ii) JVM. This is needed to avoid a falsification of later measured performance data due to, e.g., extensive class loading or database initialization operations. Once the rampup experiment finishes, the *ExperimentRunner* utilizes the *ExperimentResource* to instruct the *diagnoseIT* application to remove all rampup data from the databases, as well as to store the time stamp when the rampup was finished. The timestamp is needed to distinguish between rampup and actual experiment in the monitoring data.

To carry out the actual experiment as next step, a new experimental environment is created for each run of an experiment. In this case, experimental environment is equivalent with a new *Kieker.Analysis* instance. In the experimental setting description of the proof-of-concept evaluation (Section 6.2.2), we introduced the *DiagnoseITAnalyisTool* along with the employed Kieker pipe-and-filter structure to connect *Kieker.Analysis* and *diagnoseIT*. Since in the lab experiment the monitoring data is generated artificially, rather than provided by a real instrumented SUA, the Kieker pipe-and-filter setup had to be slightly modified.

As depicted in Figure 6.7, the *JSMReader* was replaced by a *PipeReader*. Using the *PiperReader* allows programmatically dispatching records to the pipe-and-filter chain. Hence, as soon as an experiment starts, all previous generated synthetic traces are passed to the *PipeReader*. In addition, a *RealtimeRecordDelayFilter* was inserted in between *PipeReader* and *ExecutionRecordTransformationFilter*. This filter delays the transmission of incoming monitoring records, in accordance with a record's timestamp. Since the generated traces have calculated timestamps, this is a beneficial feature to simulate real time monitoring. The rest of the experiment proceeds as same as described in Section 6.2.2. If an experiment ends, the *ExperimentRunner* instructs the server to (i) store the experiment results along with the recorded monitoring data, (ii) delete all databases, and (iii) reboot the *diagnoseIT* application. These three operations are mandatory to ensure same conditions for each experiment. Subsequently, the *ExperimentRunner* determines if this experiment should be executed once more. If all repeats are already executed, the next experiment is selected

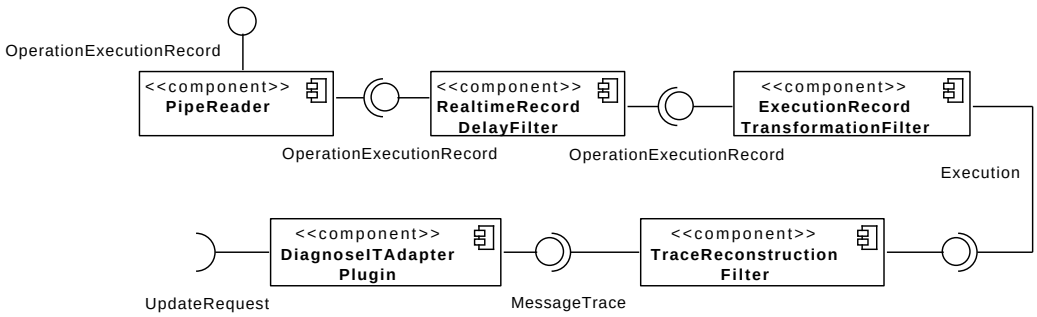


Figure 6.7. Kieker pipe-and-filter structure used in lab experiments

and initialized. If no more experiment is available, the *ExperimentRunner* is terminated.

Experiment Plan Preparation

The last step in preparing the experiments was to define the experiment plan. Previously, we described that the experiment covers two scenarios – Scenario 1 (increased intensity) and Scenario 2 (increased data variation). The final experiment plan is depicted in Table 6.5. For clarity, the original monitored data as well as the defined root dataset, respectively experiment, are presented on top of the table. To distinguish the experiments, we defined a common naming schema. The name of an experiment always contains information of those parameters that have been changed compared to the root experiment. The pattern is as follows: *TracesXTypesXEnvironments*. Accordingly, an experiment called *4x4x10* has four times more traces, four times more types, and simulates 10 resource environments. With respect to the root experiment, this sums up to 4000 traces, 800 types, and 10 resource environments.

As Table 6.5 depicts, Scenario 1 starts with the root experiment, then the amount of traces is always doubled. This means conversely that twice as many adapters are simulated. In Scenario 2, the definitions of *4x*, *8x*, and *16x* are reused, but the amount of types is continuous doubled. The different variants of Scenario 2 are referenced as Scenario 2a, 2b, and 2c.

6.3.3 Experiment Results

This section starts with a short introduction how the results are presented, followed by the results of Scenario 1 and Scenario 2.

Table 6.5. The experiment plan.

	Name	Record	Trace	Type	Env	t(s)
	Original	1905	1248	18	1	120
	Root	11000	1000	200	2	180
Scenario 1	1x	11000	1000	200	2	180
	4x	44000	4000	200	2	180
	8x	88000	8000	200	2	180
	16x	176000	16000	200	2	180
	32x	352000	32000	200	2	180
	64x	704000	64000	200	2	180
Scenario 2a	4x2x10	44000	4000	400	10	180
	4x4x10	44000	4000	800	10	180
	4x8x10	44000	4000	1600	10	180
	4x16x10	44000	4000	3200	10	180
	4x32x10	44000	4000	6400	10	180
Scenario 2b	8x2x10	88000	8000	400	10	180
	8x4x10	88000	8000	800	10	180
	8x8x10	88000	8000	1600	10	180
	8x16x10	88000	8000	3200	10	180
	8x32x10	88000	8000	6400	10	180
Scenario 2c	16x2x10	176000	16000	400	10	180
	16x4x10	176000	16000	800	10	180
	16x8x10	176000	16000	1600	10	180
	16x16x10	176000	16000	3200	10	180
	16x32x10	176000	16000	6400	10	180

Result Presentation

With respect to M3 (response time), M4 (memory consumption), and M5 (CPU utilization) we chose line graphs and Box-and-Whisker plots (will be called Boxplot from now on) as appropriate representation style.

A Boxplot is used to summarize and display the distribution of a dataset. Boxplots are mainly used to highlight five important characteristics of a dataset: the first (Q1), second (Q2 or median), and third (Q3) quartile, as well as the inter quartile range (IQR). In addition, the minimum, maximum values are shown. The IQR is a measurement for the scattering of a dataset, but compared to a simple range calculation (difference between maximum and minimum) it is less influenced by outlier values. The IQR is defined as the range between Q3 and Q1. Q1 defines that 25% of the values are smaller than Q1, Q3 that 25% are greater than Q3. As depicted in Figure 6.8, Boxplots represent the IQR as a rectangle from Q1 to Q3. Q2, the median, is represented as line within the rectangle and separates the dataset in two equally sized halves, i.e., 50% are smaller and 50% are greater than Q2. The so-called whiskers represent the minimum, maximum values as short black lines and are connected

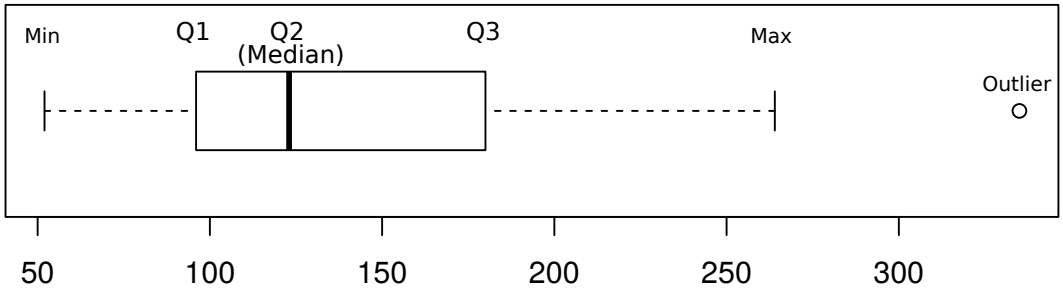
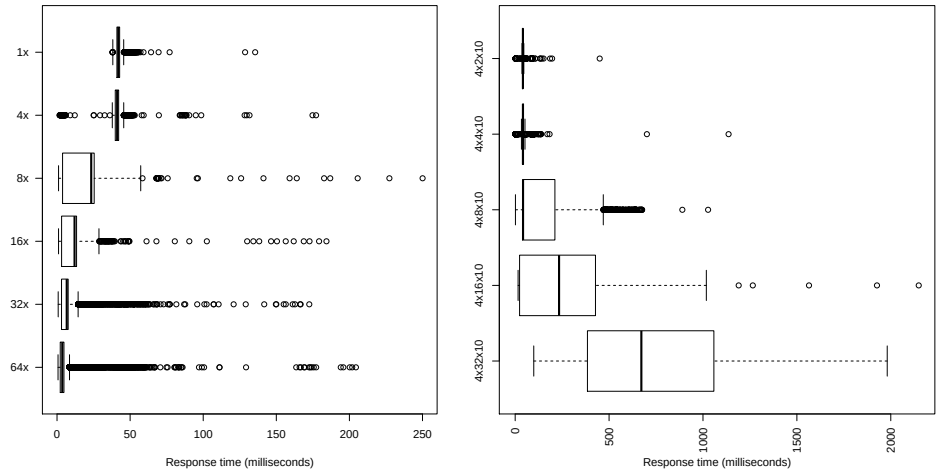


Figure 6.8. Example of a Box-and-Whisker Plot.



(a) SMMI response times for Scenario 1 **(b)** SMMI response times for Scenario 2a

Figure 6.9. Response time measurements for Scenarios 1 and 2a.

to the IQR by a dotted line [Boslaugh, 2012]. However, the minimum and maximum values are restricted to: $min \geqslant Q1 - 1.5 \cdot \text{IQR}$ and $max \leqslant Q3 + 1.5 \cdot \text{IQR}$. All values which exceed this range are considered as outliers and depicted as circles.

In addition to the visual representation of the results, a summary table are provided. The purpose of the summary tables is to provide additional information on the data. As additional data is provided, the mean, the confidence interval (CI), the standard deviation (SD), the median, and the p90 quartile. The mean defines the arithmetic average of all values and is the sum of all values divided by the total number of values. In between the CI are significant. The SD defines the average range of the the values around the mean. The p90 quartile defines that 90% are smaller and 10% are greater than the value of p90.

Scenario 1

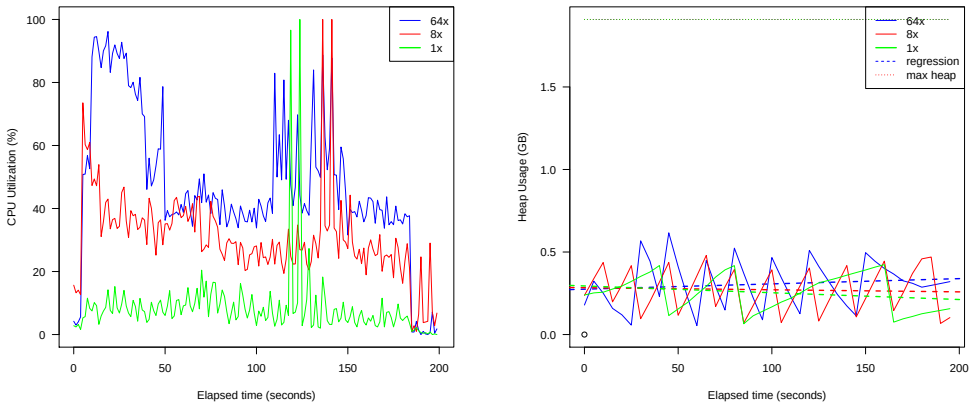
In the following the measuring results for experiments *1x*, *4x*, *8x*, *16x*, *32x*, and *64x* are presented. The summary of all measurements is depicted in Table 6.6.

- Figure 6.9a shows the measured response times for the System Model Maintenance Interface (SMMI). It is noticeable that with an increasing amount of traces, the average response times decrease. This observation is confirmed by all values of Table 6.6. Experiment *1x*, the root experiment, starts with a mean value of 43.51 milliseconds and a CI of $\pm$ 0.92. The last experiment of Scenario 1 has a mean value of 4.67 milliseconds and a CI of $\pm$ 0.05. Also the SD is continuously decreasing. The main reason for this is presumably that "only" 200 types are distributed on an increasing amount of traces (up to 64000). This leads to increased time intervals between CDO accesses. Conversely, we can say that the MongoDB performs quite well.
- Figure 6.10a illustrates the CPU utilization of the *diagnoseIT* application for experiments *1x*, *8x*, and *64x*. As can be seen, *8x* and *64x* have high peak values at the beginning. Furthermore, it is shown that all experiments have high peaks in the last third of the experiment. Unfortunately, the causes for this phenomenon are still unclear and included in all measurements. Besides this peaks it can be seen, that in contrast to response times, the CPU utilization increases along with the amount of traces.
- Figure 6.10b shows the memory consumption measurements of experiments *1x*, *8x*, *64x*. As depicted, the memory consumption is approximately constant for all experiments. The only conspicuous are two elevated peaks of *64x* at the beginning.

Scenario 2

In the following the results of Scenarios 2a, 2b, and 2c are presented. As described in Section 6.3.1, the purpose of Scenario 2 is to evaluate the behaviour of the implemented *diagnoseIT* prototype if it is put under load with different variations of data.

- Figure 6.9b shows the measured response times for Scenario 2a. Compared with the results of Scenario 1 (see Figure 6.9a), the results are very different. It is immediately apparent that increasing the amount of types has a significant impact on the response times. However, considering the amount of data that is transmitted to *diagnoseIT* in a rather short period of time, an average response time below 300 milliseconds (see Table 6.6) – stops after *4x16x10* – is acceptable. As can be seen in the summary table, the next doubling step to *4x32x10* causes a big performance degradation. Looking at the associated CPU utilization in Figure 6.12a, it can be seen that there is a medium load, but *diagnoseIT* needs, compared to the preceding experiments, almost three times as long to process *4x32x10*. If the SMMI response times of Scenario 2a (Figure 6.11a) are compared to 2b and 2c (Figures 6.11a, 6.11b), the same pattern as already known



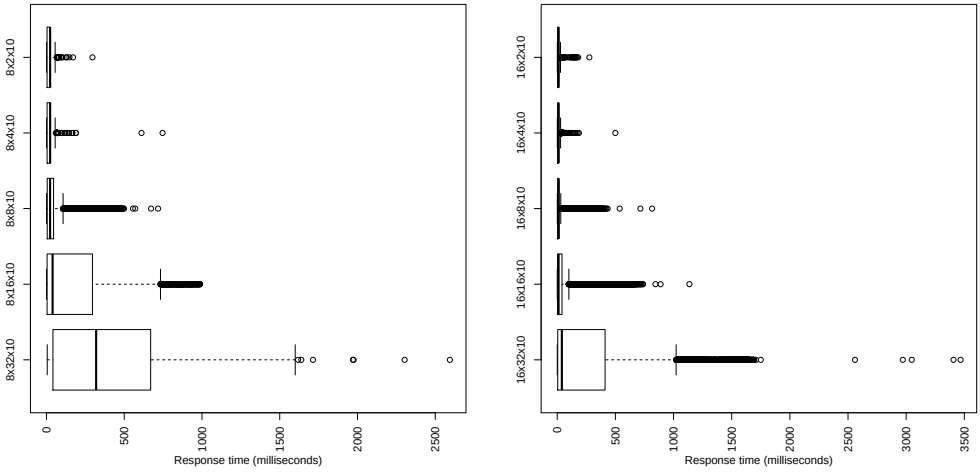
(a) CPU utilization (Scenario 1) **(b)** Memory consumption (Scenario 1)

Figure 6.10. Performance measurements for Scenario 1

from Scenario 1 is identifiable. An increasing number of traces, reduces the response times. The observation here is that this phenomenon is not related to the overall amount of types, as long as the compared experiments are dealing with the same amount. However, despite the reduced response times, *16x32x10* has a big amount of outliers with a duration up to 3500 milliseconds. With respect to the experiment plan (6.5), we can see that in experiment *16x32x10* approximately 978 ($176000 records / 180 sec$) records are arriving at the *diagnoseIT* application per second. From these measuring results we conclude that the current prototypical implementation is only partially able to cope with this volume of data.

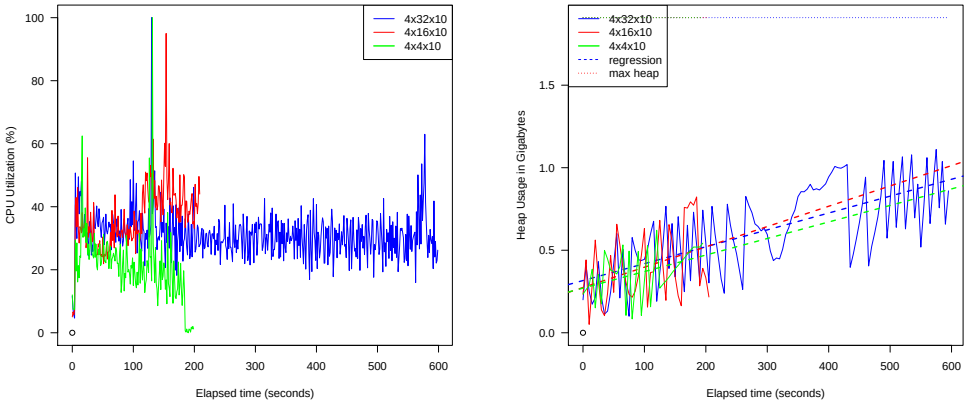
▻ With respect to the CPU utilization, measured in Scenario 2 (Figures 6.12a, 6.13a, and 6.14a), it can be seen that all series of measurement include hight peaks at almost the similar point in time. Apart from this fact, the CPU utilization differs only marginally in the different experiments.

▻ Figures 6.12b, 6.13b, and 6.14b show the measured memory consumption of Scenario 2. It is shown that, compared to Scenario 2, the increased amount of data is directly reflected in the memory consumption. An interesting observation is that Scenario 2b, the medium experiment, requires slightly more memory than 2a and 2c. Additionally, it can be seen that the longer an experiment takes the memory consumption increases. According to the experiment result data, it is not clear so far whether this is a serious issue or if the memory consumption levels off after a certain period.



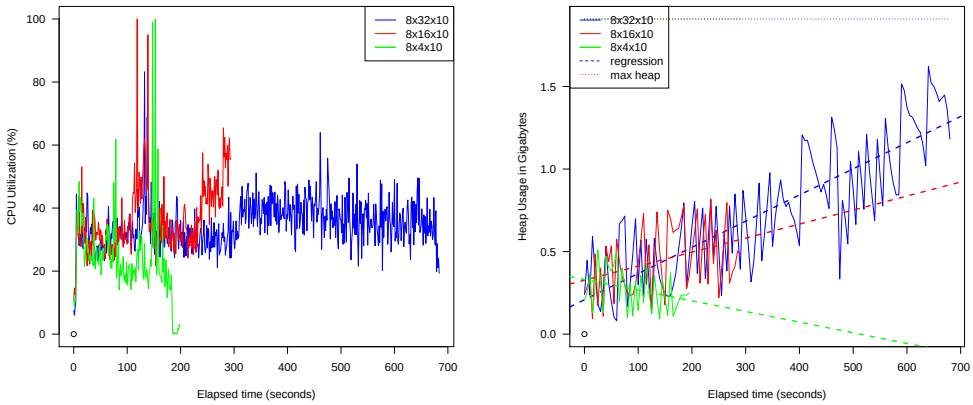
(a) SMMI response times for Scenario 2b **(b)** SMMI response times for Scenario 2c

Figure 6.11. Response time measurements for Scenario 2b,c



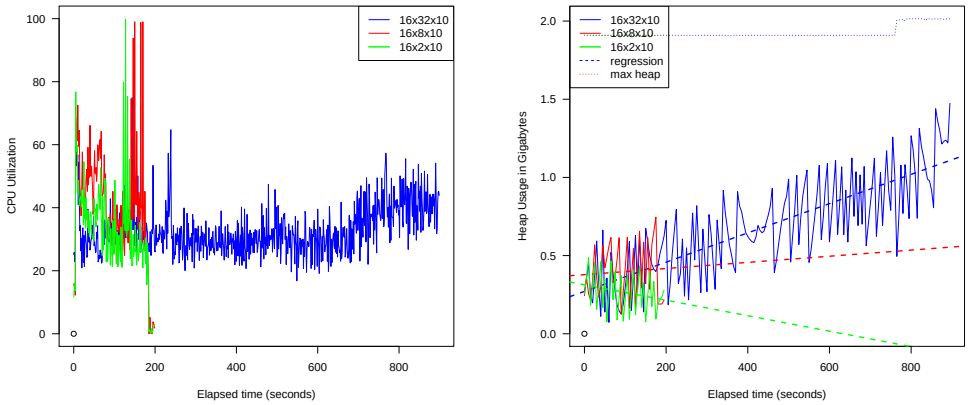
(a) CPU utilization (Scenario 2a) **(b)** Memory consumption (Scenario 2a)

Figure 6.12. Performance measurements for Scenario 2a



(a) CPU utilization (Scenario 2b) **(b)** Memory consumption (Scenario 2b)

Figure 6.13. Performance measurements for Scenario 2b



(a) CPU utilization (Scenario 2c) **(b)** Memory consumption (Scenario 2c)

Figure 6.14. Performance measurements for Scenario 2c

Result Summary

With respect to the addressed evaluation goal EG2 (*Assessing Scalability of the System Model Maintenance Interface (SMMI)*) with the enclosed questions Q1.1 (*How responsive are model repository updates with increasing load?*), Q1.2 (*How memory intensive are model repository updates with increasing load?*), and Q1.3 (*How CPU intensive are model repository updates with increasing load?*) we draw the following conclusions:

- Q1.1: This evaluation revealed that the responsiveness (M3) of repository updates strongly depends on the composition of the arriving data. This is mainly due to the fact that an hight amount of previously unknown data force updates of all parts of the Enterprise Performance Model (EPM). However, we showed that up to a certain point, respectively a combination of amount of traces and amount of data variation, *diagnoseIT* provides acceptable response times. If this point is exceeded, (in this experiment that was the step from *4x16x10* to *4x32x10*), a deterioration of response times is identifiable and single calls might last up to 3000 milliseconds. Since this performance degradation is not identifiable in the results of of Scenario 1, we assume that the current CDO integration is responsible for it. Possible improvements and adaptations are provided in Section 8.3.
- Q1.2: The experiments showed that an increasing level of data variation is directly reflected in the used memory (M3). But we showed that the *diagnoseIT* prototype is able deal with various kinds of data variations without memory errors. However, it must be examined whether the memory consumption is acceptable in longer lasting tests, or if sever memory issues arise.
- Q1.3: With respect to M5 (CPU utilization), the evaluation revealed that non of the conducted experiments forced a longer lasting above-average high CPU utilization. We showed that, independently of the experiment, the CPU usage of all experiments are comparable. However, all measurement series comprise, so far inexplicably, high peaks during the experiments. This is an occasion for further investigation.

Table 6.6. Experiment results summary

		Name	Mean	CI	SD	Median	p90	Name	Mean	CI	SD	Median	p90	
Scenario 1	**RT (ms)**	1x	43.51	± 0.92	14.87	41.86	49.34	4x2x10	41.81	± 0.33	10.75	41.18	46.82	**Scenario 2a**
		4x	41.31	± 0.29	9.42	41.13	43.43	4x4x10	44.56	± 0.91	29.37	41.30	87.59	
		8x	18.51	± 0.35	15.81	23.40	45.61	4x8x10	135.57	± 4.92	158.84	41.84	395.78	
		16x	9.95	± 0.13	8.44	11.99	16.74	4x16x10	258.69	± 6.84	220.77	233.68	561.12	
		32x	6.79	± 0.07	6.25	6.64	12.51	4x32x10	745.98	± 13.29	428.62	671.87	1354.3	
		64x	4.67	± 0.05	6.20	3.60	7.69							
	CPU (%)	1x	8.32	± 1.73	11.07	6.79	13.73	4x2x10	18.94	± 1.53	9.77	18.75	25.84	
		4x	21.36	± 2.35	15.08	20.91	27.17	4x4x10	20.74	± 1.78	11.42	21.17	29.51	
		8x	23.36	± 2.03	13.01	23.24	33.41	4x8x10	24.20	± 1.96	12.53	23.90	37.10	
		16x	30.50	± 2.15	13.74	29.06	43.29	4x16x10	36.61	± 1.6	10.50	35.18	49.32	
		32x	40.32	± 3.38	21.63	36.73	72.55	4x32x10	31.13	± 0.66	7.31	30.99	38.24	
		64x	47.11	± 3.63	23.22	42.25	85.00							
	Memory (GB)	1x	0.25	± 0.04	0.11	0.25	0.39	4x2x10	0.31	± 0.04	0.12	0.30	0.47	
		4x	0.30	± 0.04	0.14	0.29	0.49	4x4x10	0.37	± 0.04	0.14	0.38	0.52	
		8x	0.31	± 0.04	0.13	0.31	0.47	4x8x10	0.34	± 0.05	0.14	0.32	0.51	
		16x	0.27	± 0.04	0.13	0.29	0.44	4x16x10	0.40	± 0.07	0.21	0.35	0.75	
		32x	0.34	± 0.05	0.14	0.34	0.53	4x32x10	0.62	± 0.05	0.25	0.63	0.98	
		64x	0.31	± 0.05	0.14	0.31	0.50							
Scenario 2b	**RT (ms)**	8x2x10	18.75	± 0.34	15.35	23.54	45.90	16x2x10	10.11	± 0.13	8.35	12.01	17.76	**Scenario 2c**
		8x4x10	20.02	± 0.42	19.33	23.61	46.08	16x4x10	11.15	± 0.15	9.98	12.13	23.95	
		8x8x10	58.33	± 2.24	102.37	23.44	209.11	16x8x10	17.64	± 0.52	33.24	12.34	36.12	
		8x16x10	174.56	± 5.22	238.03	39.51	572.48	16x16x10	70.41	± 2.11	136.00	11.82	286.55	
		8x32x10	423.25	± 8.3	378.85	319.45	1002.09	16x32x10	276.21	± 6.41	413.38	39.85	938.42	
	CPU (%)	8x2x10	22.78	± 1.72	11.01	22.56	32.41	16x2x10	31.83	± 2.31	14.77	31.51	45.62	
		8x4x10	23.49	± 2.08	13.33	22.35	34.50	16x4x10	32.72	± 2.41	15.43	33.23	47.60	
		8x8x10	29.44	± 2.52	16.15	28.25	46.01	16x8x10	40.04	± 2.88	18.45	39.17	60.99	
		8x16x10	36.58	± 1.44	11.25	33.63	50.05	16x16x10	45.43	± 1.54	10.97	45.26	56.50	
		8x32x10	34.71	± 0.62	7.33	33.87	43.67	16x32x10	32.92	± 0.56	7.69	31.60	42.89	
	Mem. (GB)	8x2x10	0.29	± 0.04	0.12	0.29	0.45	16x2x10	0.27	± 0.04	0.12	0.25	0.44	
		8x4x10	0.27	± 0.04	0.12	0.25	0.42	16x4x10	0.34	± 0.04	0.13	0.37	0.49	
		8x8x10	0.38	± 0.06	0.18	0.39	0.61	16x8x10	0.40	± 0.06	0.17	0.36	0.62	
		8x16x10	0.45	± 0.05	0.19	0.43	0.73	16x16x10	0.49	± 0.05	0.17	0.49	0.71	
		8x32x10	0.75	± 0.06	0.38	0.72	1.31	16x32x10	0.69	± 0.04	0.31	0.66	1.09	

Chapter 7

Related Work

This section discusses work that is related to scope of this thesis. The discussed work can be classified in meta-modeling and instrumentation and monitoring (IaM) approaches. The remainder of this sections is structured as the research topics introduced in Chapter 2. At first approaches with scope on meta-modeling will be presented. Follow by an approach with scope on IaM.

The Architecture-Driven Modernization Task Force (ADMTF), a sub subdivision of the Object Management Group (OMG), developed the Structured Metrics Meta-Model (SMM). The main objective of SMM is to provide an expandable meta-model to describe measures for software systems. The ADMTF defines a measure as a method to compute values for certain elements within a software systems (e.g. algorithm to count lines of code or compute average response times). The three core elements of SMM are: Measure, Measurement and Measurand. A Measure defines what and how will be measured and can be applied to several model elements. The result of executing a Measure is stored as Measurement. Each Measurement includes a relationship to the Measurand (i.e. the measured model element) [Object Management Group, Inc, 2012b; Frey et al., 2011]. In the scope of this thesis, SMM might be used to store measured values of the SUA in the EPM. But at this point it is not yet clear to which extent measurements at all must be stored.

Due to some shortcomings of SMM, Frey et al. [2011] proposed the MAMBA approach. MAMBA extends the SMM meta-model to address two main issues: (i) aggregate functions are limited to sum, maximum, minimum, average, and standard deviation, and (ii) SMM has no native support for periodic measures. Both shortcomings are bypassed by extending the existing SMM meta-model with classes for custom aggregate functions as well as classes to defined periodic measurements. Furthermore, MAMBA provides tool support for runtime execution of SMM models and a query language [Frey et al., 2011]. In the scope of this thesis it has to be clarified if the extensions of MAMBA are actually needed or if SMM is sufficient.

The SLAstic meta-model was developed for representing architectural information of a SUA. To enable different specific architectural views on a system, the SLAstic meta-model is partitioned into four complementing sub-modules. The type repository model defines all available software components, including required and provided interfaces. In addition, it defines available types of execution containers and software/hardware resources. The component assembly model specifies the assembly components as well as the interconnection between them. An assembly component is an instance of a component

type from the type repository model. The execution environment model specifies the set of available execution containers as well as information about their interconnections. The mapping from assembly components to the executing execution container is captured from the deployment component model. Furthermore, the SLAstic approach contains a extensible set of meta-classes to specify the system behaviour and usage as well as concepts for annotating architectural entities for Quality of Service (QoS) measures and measurement instrumentation [van Hoorn, 2014]. In the scope of this thesis, the SLAstic approach might be used to reflect architectural information about the SUA within the EPM.

A further approach of the ADMTF is the Common Information Model (CIM). The purpose of CIM is to provide a conceptual information model to describe management that is not bound to a particular implementation. CIM thus enables interchanging management information between applications. CIM is partitioned in the CIM Specification and the CIM Schema. The former describes the language, naming, meta schema and mapping techniques to other management models (e.g. Simple Network Management Protocol (SNMP) and Management Information Base (MIB)). The meta schema is the formal description of the model and defines terms, usage, and semantics. The latter provides a set of classes with properties and associations as actual model description. The CIM schema thus enables organizing all relevant information about the managed environment [Object Management Group, Inc, 2012a]. In the scope of this thesis, CIM might be used to enabled management information exchange of the SUA.

Wert and Heger [2014] proposed an approach for instrumentation and monitoring (IaM) for automated software performance analysis. This approach addresses the not negligible performance overhead of code instrumentation as well as the manual effort of selective code instrumentation. Their main criticism of state-of-the-art IaM approaches is the lacking support for automatic adaptation of the instrumentation. Hence, any change of the instrumentation causes a reboot of the SUA. To provide the missing support, Wert and Heger [2014] introduced an Instrumentation Description Model (IDM) and an Adaptable Instrumentation and Monitoring (AIM) framework. IDM is an extensible meta-model to describe instrumentation and monitoring instructions for SUAs in an abstract, descriptive, and context-independent way. IDM provides two ways to capture measurement data: (i) sampling (i.e. periodically retrieve CPU utilization), and (ii) control flow instrumentation (i.e. retrieve response times from the control flow of the SUA). AIM is the counterpart of IDM, hence the execution engine of IDM instances. AIM utilizes the JVM Tool Interface (JVM TI) to enable run-time instrumentation of the SUA. In the scope of this thesis, AIM could be used to reflect information of the current instrumentation within the EPM.

Chapter 8

Conclusions and Future Work

This chapter draws conclusions of the presented work. A summary is provided in Section 8.1, followed by a discussion in Section 8.2. Section 8.3 presents outstanding issues for possible future work.

8.1 Summary

In Chapter 1 the *diagnoseIT* project was introduced, in which context this thesis emerged. The main objective of *diangnoseIT* is to enrich existing APM processes with automated configuration of instrumentation as well as automated performance problem detection. For this purpose a framework was envisioned and in this thesis three components of this framework were developed and contributed to the project.

In Chapter 4 we presented the Enterprise Performance Model (EPM), a meta-model to keep track of all relevant information of a system under analysis (SUA). The EPM includes meta-models to represent software modules, resource environments, allocation models, and execution traces. We used the Ecore meta-meta-model of the Eclipse Modeling Framework (EMF) to implement the EPM. In order to equip the EPM with persistence capabilities, we implemented a model repository on top of the Connected Data Objects (CDO) framework.

Once the EPM was defined and implemented, we implemented the associated framework. The framework comprises two major components – the *diagnoseIT Application* and a so-called *Adapter* (see Chapter 5). The *Application* provides all required functionality to maintain the EPM, this functionality is made accessible by publishing a Representational State Transfer (REST) API. The provided API is supported by the *Adapter* and can be used by any third-party application to interact with the *Appliaction*.

The implemented framework was evaluated in Chapter 6. To evaluate different aspects of the developed system, we conducted a proof-of-concept evaluation as well as lab experiments. The results obtained are divided into two parts, from a functional perspective, we can say that all components work as envisioned. Considering the performance measurements we have to say that there are still open issues.

8.2 Discussion

The following discussion targets goals **G1** (*Enterprise Performance Model and Repository*) and **G2** (*System Model Maintenance Interface (SMMI) and Prototype*) with the enclosed research questions (**R1-R5**) defined in Chapter 2. **G3** (*Evaluation*) is implicit discussed since the evaluation results are used to answer research questions **R4** and **R5**. Each research question is answered and considered in isolation.

RQ1: Which properties have to be covered by the EPM?

In Section 4.1, we presented the results of the requirements analysis regarding the EPM. As stated, the current implementation is considered as initial version for the *diagnoseIT* project. The meta-model is designed as simple as possible, according to the requirements of the project's participants, but nevertheless it includes all relevant parts to cover entire software modules, resource environments, execution traces, and allocations. An allocation defines a mapping of a software module to the resource environment which is executing it. Since the *diagnoseIT* projects targets Java enterprise applications, the resource environment meta-model is optimized to include information of the executing JVMs. Summarizing we say that the proposed EPM is a good starting point for *diagnoseIT* and can easily be extended to comply with new requirements.

RQ2: How can the different parts of the meta-model be combined into one meta-model?

In order to comply with the separation of concern design principle, the EPM is split into five complementing sub meta-models: *ResourceEnvironmentModel*, *AllocationModel*, *SoftwareModuleModel*, *TraceModel*, and *InstrumentationModel*. With respect to the *InstrumentationModel* we have to say that it is envisioned and was defined in a first version, but it was neither implemented nor tested. Further details how the different sub meta-model are interconnected are provided in Chapter 4.

RQ3: What is an appropriate technology stack for the *diagnoseIT* prototype?

Since the technology stack concerns all parts of the framework this was a crucial aspect of this thesis. In summary we have implemented a Java Web application which publishes a REST API. The implementation was support by several third-party libraries.

The required model-repositories are implemented on top of EMP's CDO framework. Benefit of using CDO is the out-of-box support for EMF/Ecore models, thus also for our EPM. As additional database backend for dynamic data we used the Not only SQL (NoSQL) database engine MongoDB.

The encompassing framework architecture is implemented by the use of well-known and mature Java frameworks. When selecting the frameworks was ensured that they either implement a certain specification or de-facto standards. Considering this requirement

we chose Jersey (Java API for RESTful Web Services (JAX-RS) specification), HK2 (JSR-330 specification), Dropwizard (Combines Jersey and HK2), and Google Guava. A more detailed description of the employed frameworks is available in Chapter 5.

RQ4: Does the approach meet all previously specified requirements?

In Chapter 6 this research question was refined to an evaluation goal (EG) as follows: *Assessing functionality of the diagnoseIT framework*, with the enclosed questions Q1.1 (*Does the diagnoseIT prototype meet all previously specified requirements?*) and Q1.2 (*Up to what granularity is Kieker able to serve the EPM?*)

The EG (respectively the research question) was evaluated by conducting a proof-of-concept evaluation (Section 6.2). The results of the evaluation confirm that, from a functional point of view, all specified requirements are fulfilled. We were able to instrument and monitor a Java Web application with the Kieker framework and to process, store, and inspect the monitored data with the *diagnoseIT* framework. This enables us to identify goals **G1** and **G2** as fulfilled.

RQ5: Is the approach applicable in real world scenarios?

In Chapter 6 this research question was refined to an evaluation goal (EG) as follows: *Assessing Scalability of the System Model Maintenance Interface (SMMI)*), with the enclosed questions Q1.1 (*How responsive are model repository updates with increasing load?*), Q1.2 (*How memory intensive are model repository updates with increasing load?*), and Q1.3 (*How CPU intensive are model repository updates with increasing load?*)

The EG (respectively the research question) was evaluated by conducting a lab experiment (Section 6.3). The results of the experiment showed that the applicability is not yet guaranteed. With respect to response time measurements the results are reasonable, but the results for CPU utilization and memory consumption enforce further investigations. In certain situations were not negligible CPU utilization peaks recognizable. The causes for this peaks are so far inexplicable. The related test results are available in Chapter 6.

8.3 Future Work

In the following, outstanding issues for possible future work are outlined.

- As described in Chapter 4, the EPM includes a further meta-model to cover information of the current instrumentation of the SUA. This meta-model needs to be defined and integrated in the EPM as well as the *diagnoseIT* framework.
- In the current version of the *diagnoseIT* framework, the CDO server is realized as embedded component. With the current implementation of the CDO server it should be possible with little effort to create a fully stand-alone CDO server component.

- A further improvement of the model repositories would be to connect the CDO server to a NoSQL database. In the course of this thesis it was investigate if CDO could be connected to the same MongoDB instance which is used to store the dynamic trace data. CDO already provides a so-called *CDO/MongoDB Store* which is intended to solve exactly this idea. But *CDO/MongoDB Store* is developed for MongoDB v.1.6.5, *diagnoseIT* uses v.3.0.2. Since it is not guaranteed that the *CDO/MongoDB Store* will ever be adapted to a newer version of MongoDB, it is possible to implement a custom *CDO Store* to enable this improvement.
- To improve the Kieker integration, *Kieker.Analysis* needs to be enhanced with a functionality to request information of the monitored system from *Kieker.Monitoring*. This functionality enables a proper implementation of the *InaccessibleInformationResolver* interface (Chapter 5, Section 6.2).

Bibliography

[Bass et al. 2012] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[Boslaugh 2012] S. Boslaugh. *Statistics in a nutshell*. " O'Reilly Media, Inc.", 2012.

[Brambilla et al. 2012] M. Brambilla, J. Cabot, and M. Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[Caldeira and Rombach 1994] V. Caldeira and H. D. Rombach. The goal question metric approach. *Encyclopedia of software engineering*, 2(1994):528–532, 1994.

[Chikofsky et al. 1990] E. J. Chikofsky, J. H. Cross, et al. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

[Czarnecki and Helsen 2006] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[diagnoseIT 2015] diagnoseIT, 2015. URL `http://diagnoseit.github.io/`.

[Dropwizard] Dropwizard. *Dropwizard User Guide*. URL `http://www.dropwizard.io/manual/index.html`.

[Eclipse Foundation 2014a] Eclipse Foundation. Eclipse modeling project, 2014a. URL `http://eclipse.org`.

[Eclipse Foundation 2014b] Eclipse Foundation. Cdo/server configuration reference, 2014b. URL `https://wiki.eclipse.org/CDO/Server_Configuration_Reference`.

[Eclipse Foundation 2014c] Eclipse Foundation. Cdo/user contributed documentation, 2014c. URL `http://wiki.eclipse.org/CDO/User_Contributed_Documentation`.

[Eclipse Foundation 2015] Eclipse Foundation. Jetty - servlet engine, 2015. URL `http://www.eclipse.org/jetty/`.

[Fowler 2002] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[Frey et al. 2011] S. Frey, A. van Hoorn, R. Jung, W. Hasselbring, and B. Kiel. Mamba: A measurement architecture for model-based analysis. 2011.

[Garlan et al. 2010] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON First Decade High Impact Papers*, pages 159–173. IBM Corp., 2010.

[Google 2015] Google. Guava - user guice, 2015. URL http://code.google.com/p/guava-libraries/wiki/GuavaExplained.

[Hadley and Sandoz 2009] M. Hadley and P. Sandoz. Jax-rs: Java™ api for restful web services. *Java Specification Request (JSR)*, 311, 2009.

[Hamilton and Miles 2006] K. Hamilton and R. Miles. *Learning UML 2.0*, volume 286. O'Reilly, 2006.

[Kambalyal 2010] C. Kambalyal. 3-tier architecture. 2, 2010. URL http://channukambalyal.tripod.com/.

[Kieker Project] Kieker Project. *Kieker 1.10 User Guide*. URL http://kieker-monitoring.net/documentation/.

[Krogmann 2012] K. Krogmann. *Reconstruction of software component architectures and behaviour models using static and dynamic analysis*, volume 4. KIT Scientific Publishing, 2012.

[Ludewig 2003] J. Ludewig. Models in software engineering–an introduction. *Software and Systems Modeling*, 2(1):5–14, 2003.

[Matevska 2010] J. Matevska. Software-architekturbeschreibung. In *Rekonfiguration komponentenbasierter Softwaresysteme zur Laufzeit*, pages 49–72. Springer, 2010.

[Medvidovic and Taylor 2000] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70–93, 2000.

[MyBatis 2015] MyBatis. Mybatis - jpetstore, 2015. URL https://github.com/mybatis/jpetstore-6.

[Object Management Group, Inc 2012a] Object Management Group, Inc. Common information model (cim), January 2012a. URL http://www.dmtf.org/standards/cim.

[Object Management Group, Inc 2012b] Object Management Group, Inc. Architecture-driven modernization (adm): Structured metrics meta-model (smm), January 2012b. URL http://www.omg.org/spec/SMM/.

[Oracle Corporation 2015a] Oracle Corporation. Hk2 - a light-weight and dynamic dependency injection framework, 2015a. URL https://hk2.java.net/2.4.0-b26/.

[Oracle Corporation 2015b] Oracle Corporation. Jersey - restful web services in java, 2015b. URL https://jersey.java.net/.

[Smith and Williams 2002] C. Smith and L. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software.* Addison-Wesley, 2002.

[Steinberg et al. 2008] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework.* Pearson Education, 2008.

[Stepper 2008] E. Stepper. Connected data objects (cdo) - the emf model repository. 2008. URL https://eclipse.org/cdo/documentation/presentations/EclipseCon_2008/CDO-Presentation.pdf.

[Stepper 2015] E. Stepper. Cdo model repository documentation., 2015. URL http://www.eclipse.org/cdo/documentation/.

[van Hoorn 2014] A. van Hoorn. *Model-Driven Online Capacity Management for Component-Based Software Systems*. Number 2014/6 in Kiel Computer Science Series. Department of Computer Science, Kiel University, Kiel, Germany, 2014. Dissertation, Faculty of Engineering, Kiel University.

[van Hoorn et al. 2009] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: Design and application of the kieker framework. 2009.

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

[Weber 2010] S. Weber. Nosql databases. *University of Applied Sciences HTW Chur, Switzerland*, 2010.

[Wert and Heger 2014] A. Wert and C. Heger. Adaptable instrumentaion and monitoring, 2014. URL http://sopeco.github.io/AIM/.

Decleration

I declare that this thesis is the solely effort of the author. I did not use any other sources and references than the listed ones. I have marked all contained direct or indirect statements from other sources as such. Neither this work nor significant parts of it were part of another review process. I did not publish this work partially or completely yet. The electronic copy is consistent with all submitted copies.

---

place, date, signature