TECHNISCHE UNIVERSITÄT MÜNCHEN
Fakultät für Informatik
Lehrstuhl für Wirtschaftsinformatik (I 17)
Prof. Dr. Helmut Krcmar

Integration von Performancebewusstsein in Entwicklungsumgebungen für komponentenbasierte Unternehmensanwendungen

Alexandru Danciu

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Hans-Joachim Bungartz

Prüfer der Dissertation:

1. Prof. Dr. Helmut Krcmar
2. Prof. Bernd Brügge, Ph.D.

Die Dissertation wurde am 05.12.2018 bei der Technischen Universität eingereicht und durch die Fakultät für Informatik am 29.05.2019 angenommen.

Zusammenfassung

Ziel: Ziel dieser Arbeit ist die Erarbeitung eines Ansatzes für die Einbettung von Performancebewusstsein in integrierte Entwicklungsumgebungen für komponentenbasierte Unternehmensanwendungen zur Unterstützung der Entwickler mit Einsichten über die erwartete Antwortzeit ihrer Implementierung, bevor diese überhaupt ausführbar ist.

Methode: Die Arbeit lässt sich im Bereich der gestaltungsorientierten Forschung einordnen. Im Rahmen einer Literaturanalyse werden existierende Forschungsergebnisse strukturiert und Forschungslücken abgeleitet. Darauf aufbauend wird ein Konzept für die Unterstützung von Performancebewusstsein bei Entwickler komponentenbasierter Unternehmensanwendungen entworfen. Das resultierende Konzept wird beispielhaft für die Unterstützung von Antwortzeitvorhersagen für Java-EE-Komponenten umgesetzt. Im Rahmen eines kontrollierten Experiments wird die Präzision der Antwortzeitvorhersagen evaluiert. Zur Untersuchung und Bewertung des Einflusses von Performancebewusstsein auf Entwickler und die daraus resultierende Antwortzeit von Implementierungen wird ein Experiment mit Versuchspersonen durchgeführt. Als nächstes wird ein Konzept für die Verwaltung und Versionierung von Performancemodellen entwickelt. Das Ergebnis wird beispielhaft instanziiert und mit dem zuvor entwickelten Ansatz für Performancebewusstsein integriert.

Resultate: Der Ansatz für Performancebewusstsein unterstützt Entwickler mit Antwortzeitvorhersagen ausgehend von der Struktur, der Wiederverwendung, dem Kontrollfluss und der Parametrisierung von Komponenten. Für die Berechnung von erwarteten Antwortzeiten werden statische und dynamische Informationen integriert und Puffermechanismen berücksichtigt. Durch die Verfügbarkeit des Ansatzes für Performancebewusstsein können Entwickler durchschnittlich über drei Mal mehr Performancebugs beheben. Der Ansatz unterstützt ein methodisches Vorgehen bei der Analyse von Quelltext.

Der Ansatz des Performance Model Management Repository bietet ein Meta-Modell für die Versionierung sowie Verwaltung von Komponenten- und Schnittstellenspezifikation in einer zentralen Ablage, die Übertragung von Änderungen an Spezifikationen auf Performancemodelle und die Verwaltung von hardwarespezifischen Ressourcenbedarfen.

Implikationen: Die Quantifizierung des Einflusses von Performancebewusstsein auf Entwickler demonstriert die Relevanz dieses Konzeptes. Durch die Anwendung des Ansatzes für Performancebewusstsein in der Praxis können Entwicklungsabteilungen und Softwaredienstleister die Qualität der implementierten Softwarelösungen erhöhen ohne dabei die Produktivität der Entwickler zu gefährden. Herausforderungen bei der Umsetzung des Ansatzes im Kontext von Java EE werden durch eine beispielhafte Implementierung aufgezeigt. Die Integration des Performance Model Management Repository mit dem Ansatz für Performancebewusstsein beschreibt, wie Modellartefakte zwischengespeichert und wiederverwendet werden können.

Abstract

Purpose: The aim of this thesis is to develop an approach for embedding performance awareness into integrated development environments for component-based enterprise applications to support developers with insights on the expected response time of their implementation before it is even executable.

Methodology: The research presented in this thesis is guided by the methodological foundation of Design Science. To address the main goal of this thesis, an approach for supporting developers of component-based enterprise applications with performance awareness is developed. The resulting approach is then instantiated for supporting response time estimations for Java EE components. In a controlled experiment, the precision of the estimations is evaluated. To investigate if this approach can help developers improve the performance of software implementations, a human-oriented experiment is conducted. Next, an approach for the management and versioning of performance models is developed. The result is instantiated and integrated with the previously developed performance awareness approach.

Results: The Performance Awareness approach supports developers with response time estimations based on the structure, reuse, control flow, and parameterization of components. For calculating expected response times, static and dynamic information are integrated, and buffer mechanisms are considered. While employing the approach, subjects fix on average over three times more performance bugs. The awareness approach enables a methodical investigation of code.

The Performance Model Management Repository approach addresses the versioning and management of component and interface specifications in a central database, the transfer of changes to specifications to performance models, and the management of hardware-specific resource requirements.

Implications: By quantifying the impact of performance awareness, the relevance of the concept is demonstrated. By applying the performance awareness approach in practice, development departments and software service providers can increase the quality of implemented software solutions without compromising developer productivity. Challenges of applying the approach in the context of Java EE are identified by an exemplary implementation. The integrating of the Performance Model Management Repository with the performance awareness approach shows how model artefacts can be cached and reused.

Danksagung

Zuallererst möchte ich Prof. Dr. Helmut Krcmar dafür danken, dass er mir die Gelegenheit gegeben hat, an diesem Forschungsthema unter seiner Anleitung zu arbeiten. Vielen Dank für die Begleitung des Vorhabens, für die Unterstützung und das wertvolle Feedback. Ich bedanke mich bei Prof. Bernd Brügge für seine Rolle als Zweitgutachter und bei Prof. Hans-Joachim Bungartz als Vorsitzenden der Prüfungskommission.

Ein besonderer Dank gilt meinen Kollegen am Forschungsinstitut fortiss. Ohne die fachliche und persönliche Unterstützung von Johannes Kroß, Christian Vögele, Felix Willnecker und Andreas Brunnert wäre diese Arbeit nicht zustande gekommen. Vielen Dank an dem gesamten FB3 für die herzliche Aufnahme.

Meinen Kollegen vom SAP University Competence Center, Stephan Gradl, Manuel Mayer, André Bögelsack, Marcus Homann, Vassilena Banova, Christos Konstantinidis, Jörg Schmidl, Sonja Hecht, Harald Kienegger und Holger Wittges, bin ich für die großartigen gemeinsamen Erfahrungen aus Projekten und der Lehre dankbar. Den Kollegen vom Lehrstuhl für Wirtschaftsinformatik danke ich für das wertvolle Feedback während der Doktorandenseminaren.

Nicht zuletzt möchte ich mich bei meiner Familie bedanken. Ich danke meinen Eltern Ioana und Gheorghe für ihre aufopfernde Hingabe und die mir geschenkte Erziehung. Bei meiner Ehefrau Irina möchte ich mich für die bedingungslose Unterstützung und das aufgebrachte Verständnis bedanken.

Inhaltsverzeichnis

Zusammenfassung III
Abstract IV
Danksagung V
Inhaltsverzeichnis VI
Abbildungsverzeichnis XII
Tabellenverzeichnis XVI
Abkürzungsverzeichnis XVIII
1 Einleitung 1
1.1 Motivation 1
1.2 Problemstellung 2
1.3 Ziele der Arbeit 3
1.4 Forschungsfragen 5
1.5 Forschungsansatz 6
1.6 Aufbau der Arbeit 9
2 Theoretische Grundlagen 12
2.1 Softwarequalitätsmodelle 13
2.2 Performancebewusstsein 20
2.2.1 Schwerpunkte von Performancebewusstsein 20
2.2.2 Abgrenzung Performancebewusstsein und Entwicklerrückkopplung 21
2.2.3 Kriterien für Performancebewusstsein bei Entwicklern 22
2.2.4 Aspekte von Performancebewusstsein 22
2.2.5 Andere Arten von Bewusstsein während der Softwareentwicklung 24
2.2.6 Continuous Software Engineering 24
2.3 Komponentenbasierte Softwareentwicklung 25
2.3.1 Phasen der Entwicklung und Wiederverwendung von Komponenten 25
2.3.2 Einflussfaktoren auf die Performance einer Komponente 26
2.3.3 Arten der Anbindung von wiederverwendeten Diensten 27
2.4 Serviceorientierte Architekturen 28
2.5 Java Enterprise Edition 32
2.5.1 Architektur und Komponenten 32
2.5.2 Speicherung von Entity-Instanzen 34

2.5.3 Rollen und Lebenszyklus ... 35
2.6 Modellbasierte Performanceevaluation ... 35
2.6.1 Warteschlangennetze ... 36
2.6.2 Layered Queueing Networks ... 37
2.7 Das Palladio Component Model ... 39
2.7.1 Meta-Ebenen des Palladio Component Model ... 39
2.7.2 Modellartefakte einer PCM-Instanz ... 40
3 Verwandte Forschungsarbeiten ... 43
3.1 Unterstützung von Performancebewusstsein der Entwickler ... 43
3.1.1 Messbasierte Ansätze für Performancebewusstsein ... 44
3.1.2 Modellbasierte Ansätze für Performancebewusstsein ... 49
3.1.3 Adressierte Forschungslücken ... 53
3.2 Reverse Engineering von Komponenten ... 54
3.3 Evaluation der Rückmeldung zum Softwareentwickler ... 56
3.4 Versionskontrollsysteme für Performancemodelle ... 58
3.5 Zusammenfassung ... 60
4 Ansatz für die Unterstützung von Performancebewusstsein ... 61
4.1 Motivation ... 61
4.1.1 Problemstellung ... 61
4.1.2 Forschungsziele ... 63
4.1.3 Kontext ... 63
4.2 Konzept für die Unterstützung von Performancebewusstsein ... 64
4.2.1 Abbildung der Komponentenstruktur ... 66
4.2.2 Klassifizierung von Komponenten ... 68
4.2.3 Abbildung des Kontrollflusses ... 70
4.2.4 Abbildung der Parametrisierung ... 71
4.2.5 Abbildung von Puffermechanismen ... 72
4.2.6 Abbildung des Antwortzeitverhaltens von wiederverwendeten Komponenten ... 73
4.2.7 Abbildung von mehrfachen Implementierungen ... 76
4.2.8 Beieinflussug des Entwicklers mit Hilfe von Performancebewusstsein ... 78
4.3 Technische Umsetzung des Ansatzes ... 79
4.3.1 Erstellung des Performancemodells ... 80
4.3.1.1 Technische Grundlage der Implementierung ... 82

4.3.1.2 Einlesen des Java-Quelltextes 83
4.3.1.3 Erstellung des Repository-Modells 84
4.3.1.4 Erstellung des System-Modells 86
4.3.1.5 Erstellung des Resource-Environment- und Allocation-Modells 87
4.3.1.6 Spezifizierung des Antwortzeitverhaltens von externen Komponenten 88
4.3.1.7 Erstellung des Usage-Modells 89
4.3.1.8 Erstellung der RDSEFF 90
4.3.1.9 Modellierung von gepufferten Aufrufen 92
4.3.2 Erhebung des Antwortzeitverhaltens von wiederverwendeten Diensten 95
4.3.2.1 Monitoring des Antwortzeitverhaltens 95
4.3.2.2 Instrumentierung von Unternehmensanwedungen 96
4.3.2.3 Zuordnung von Komponenten zu Schnittstellen 97
4.3.2.4 Architektur zur Erhebung und Abfrage von Antwortzeitmessungen 98
4.3.3 Simulation des Antwortzeitverhaltens 98
4.3.4 Interaktion mit dem Softwareentwickler 100
4.3.4.1 Konfiguration durch den Softwareentwickler 100
4.3.4.2 Rückmeldung zum Softwareentwickler 102
4.4 Evaluation der Vorhersagegenauigkeit 104
4.4.1 Aufbau des Experiments 105
4.4.2 Experimentumgebung 106
4.4.3 Durchführung des Experiments 110
4.4.4 Ergebnisse 110
4.4.5 Validität der Ergebnisse 115
4.5 Zusammenfassung und Ausblick 117
4.5.1 Limitationen 117
4.5.2 Zukünftige Forschung 119
5 Einfluss von Performancebewusstsein auf die Antwortzeit von Softwarekomponenten 121
5.1 Motivation 121
5.1.1 Problemstellung 121
5.1.2 Forschungsziele 121
5.1.3 Kontext 122
5.2 Gestaltung des Experiments 123
5.2.1 Ziele, Hypothesen und Variablen des Experiments 123

5.2.2 Aufbau des Experiments ... 125
5.2.3 Auswahl von Versuchspersonen ... 127
5.2.4 Objekte des Experiments ... 127
5.2.4.1 Beispielhafte Unternehmensanwendung ... 127
5.2.4.2 Antipatterns als Grundlage von Performancebugs ... 129
5.2.4.3 Konstruktion von Performancebugs ... 131
5.2.4.4 Entwicklungsumgebung ... 137
5.2.5 Instrumentierung des Experiments ... 137
5.2.5.1 Anleitung für Versuchspersonen ... 137
5.2.5.2 Instrumentierung der Entwicklungsumgebung ... 138
5.2.5.3 Nachgelagerte Befragung zur Erhebung der Kenntnisse ... 140
5.2.5.4 Nachgelagerte Befragung zur Erhebung der Benutzerfreundlichkeit ... 141
5.2.6 Verfahren zum Zusammentragen der Daten ... 142
5.2.7 Verfahren zur Analyse der Ergebnisse ... 143
5.2.8 Sicherstellung der Validität der Ergebnisse ... 144
5.2.9 Gegenüberstellung des Experimentaufbaus ... 145
5.3 Durchführung des Experiments ... 146
5.3.1 Teilnehmer des Experiments ... 146
5.3.2 Iterationen des Experiments ... 147
5.3.3 Sicherstellung der Validität der Ergebnisse ... 149
5.3.4 Beispielhafter Output des Plugins für Performancebewusstsein ... 149
5.4 Auswertung der Eigenschaften der Versuchspersonen ... 150
5.4.1 Verteilung der Versuchspersonen anhand der Ausbildung und des Arbeitsberhältnisses ... 150
5.4.2 Verteilung der Versuchspersonen anhand der Programmierkenntnisse ... 152
5.4.3 Verteilung der Versuchspersonen anhand der spezifischen Kenntnisse ... 152
5.5 Auswertung über die behobenen Performancebugs ... 153
5.5.1 Deskriptive Statistik und Datenbereinigung ... 154
5.5.2 Verteilung der Ergebnisse anhand der Erfahrung der Versuchspersonen ... 156
5.5.3 Durchführung des Hypothesentests ... 159
5.6 Auswertung der Zeitaufwände von Probanden ... 160
5.6.1 Deskriptive Statistik und Datenberinigung ... 161
5.6.2 Durchführung des Hypothesentests ... 163
5.7 Auswertung des methodischen Vorgehens von Probanden ... 164

5.7.1 Deskriptive Statistik und Datenbereinigung ... 164
5.7.2 Durchführung des Hypothesentests ... 167
5.8 Auswertung der Benutzerfreundlichkeit ... 168
5.8.1 Deskriptive Statistik und Datenberinigung ... 168
5.8.2 Durchführung des Hypothesentests ... 169
5.9 Interpretierung der Ergebnisse ... 170
5.9.1 Evaluation der Ergebnisse und Implikationen ... 170
5.9.2 Validität des Experiments ... 172
5.9.2.1 Statistische Validität ... 172
5.9.2.2 Interne Validität ... 173
5.9.2.3 Konstruktvaliditát ... 174
5.9.2.4 Externe Validität ... 176
5.9.3 Verallgemeinerbarkeit der Ergebnisse ... 178
5.9.4 Gewonnene Erkenntnisse ... 178
5.10 Zusammenfassung und Ausblick ... 179
6 Performance Model Management Repository ... 182
6.1 Motivation ... 182
6.1.1 Problemstellung ... 182
6.1.2 Forschungsziele ... 183
6.1.3 Kontext ... 184
6.2 Konzept für das Management von Performancemodellen ... 186
6.2.1 Versionierung von Komponenten und Schnittstellen ... 187
6.2.2 Verwaltung von Versionsspezifikationen ... 188
6.2.3 Übertragung von Änderungen an Versionsspezifikationen ... 190
6.2.4 Verwaltung von Ressourcenbedarfe ... 190
6.3 Technische Umsetzung ... 192
6.3.1 Grundlegende Technologien ... 192
6.3.2 Erweiterungen des Palladio Component Model ... 193
6.3.2.1 Versionierung von Komponenten- und Schnittstellenspezifikationen ... 194
6.3.2.2 Verwaltung von Komponenten- und Schnittstellenspezifikationen ... 195
6.3.2.3 Übertragung von Änderungen an Versionsspezifikationen ... 196
6.3.2.4 Verwaltung von Ressourcenbedarfe ... 198
6.3.3 Verwaltung und Bearbeitung von Versionsspezifikationen ... 199
6.3.3.1 Import von Versionsspezifikationen ... 200

6.3.3.2 Export von Versionsspezifikationen ... 201
6.3.4 Integration mit dem Ansatz für Performancebewusstsein ... 202
6.3.4.1 Umfang der Wiederverwendung von Spezifikationen ... 203
6.3.4.2 Versionierung von Spezifikationen ... 204
6.3.4.3 Abruf von existierenden Spezifikationen ... 205
6.3.4.4 Speicherung von Spezifikationen ... 206
6.4 Evaluation ... 206
6.5 Zusammenfassung und Ausblick ... 208
7 Zusammenfassung und Ausblick ... 210
7.1 Zusammenfassung der Ergebnisse ... 210
7.1.1 Erste Forschungsfrage ... 210
7.1.2 Zweite Forschungsfrage ... 211
7.1.3 Dritte Forschungsfrage ... 212
7.2 Beiträge für Wissenschaft und Praxis ... 213
7.3 Annahmen und Limitationen ... 214
7.4 Ausblick ... 216
Literaturverzeichnis ... 221
Anhang A Anleitung zur Durchführung des Experiments ... 236
Anhang A-1 Anleitung für Probanden der Testgruppe ... 236
Anhang A-2 Anleitung für Probanden der Kontrollgruppe ... 240
Anhang B Fragebogen für Versuchspersonen ... 243
Anhang B-1 Fragebogen für Versuchspersonen der Testgruppe ... 243
Anhang B-2 Fragebogen für Versuchspersonen der Kontrollgruppe ... 246

Abbildungsverzeichnis

Abbildung 1-1: Frühzeitige Unterstützung von Softwareentwickler mit automatisierten Performanceevaluationen im Kontext komplexer Unternehmensanwendungen ... 3

Abbildung 1-2: Information Systems Research Framework nach Hevner et al. ... 7

Abbildung 1-3: Struktur der Arbeit und Zuordnung der Kapitel zu Forschungsfragen ... 10

Abbildung 2-1: Cloud, Fog und Edge Computing ... 12

Abbildung 2-2: Produkteigenschaften des FURPS-Qualitätsmodells ... 15

Abbildung 2-3: Modell für Produktqualität nach ISO/IEC 25010 ... 17

Abbildung 2-4: Schnittmenge zwischen Performancebewusstsein und Entwicklerrückkopplung ... 21

Abbildung 2-5: Aspekte von Performancebewusstsein ... 23

Abbildung 2-6: Einflussfaktoren auf die Performance einer Komponente ... 26

Abbildung 2-7: Serviceorientierte Architekturen ... 28

Abbildung 2-8: Arten der Konfiguration von Dependency Injection ... 31

Abbildung 2-9: Architektur der Java EE ... 33

Abbildung 2-10: Lebenszyklus von Java-EE-Applikationen ... 35

Abbildung 2-11: Beispiel für ein Warteschlangennetz ... 36

Abbildung 2-12: Beispiel für ein Layered Queueing Network ... 38

Abbildung 2-13: Meta-Ebenen des Palladio Component Model ... 40

Abbildung 2-14: Modellartefakte einer PCM-Instanz ... 41

Abbildung 3-1: Teilbereiche verwandter Forschungsarbeiten ... 43

Abbildung 4-1: Schritte für die manuelle Abschätzung des Antwortzeitverhalten von Komponenten einer Unternehmensanwendung ... 62

Abbildung 4-2: Ansatz für die Vorhersage der Antwortzeit von Komponenten einer Unternehmensanwendung zur Unterstützung von Performancebewusstsein ... 64

Abbildung 4-3: Idee für die Vorhersage der Antwortzeit von Komponenten ... 65

Abbildung 4-4: Beispiel für die Abbildung einer Java-EE-Anwendung auf Komponenten und Schnittstellen ... 67

Abbildung 4-5: Beispiel für die Abbildung und Abgrenzung von internen und externen Komponenten ... 69

Abbildung 4-6: Beispiel für die Abbildung des Kontrollflusses von Komponenten ... 70

Abbildung 4-7: Beispiel für die Abbildung der Parametrisierung von Dienstaufrufen ... 72

Abbildung 4-8: Beispiel für die Pufferung von Daten innerhalb einer Session ... 73

Abbildung 4-9: Integration von modell- und messbasierten Verfahren für die Performanceevaluierung 76
Abbildung 4-10: Beispiel für die mehrfache Implementierung einer Schnittstelle durch externe Komponenten 77
Abbildung 4-11: Beispiel für Protokollierung des Aufrufs und der Ausführung einer Methode 78
Abbildung 4-12: Rückmeldung der Antwortzeitvorhersage zum Entwickler 78
Abbildung 4-13: Ansatz für die Unterstützung von Performancebewusstsein 79
Abbildung 4-14: Ansatz für die Erstellung des Performancemodells 81
Abbildung 4-15: Ausschlussferfahren für die Identifikation von internen Komponenten 85
Abbildung 4-16: Vereinfachte Darstellung eines automatisch erstellen Repository-Modells 86
Abbildung 4-17: Vereinfachte Darstellung eines automatisch erstellten System-Modells 87
Abbildung 4-18: Vereinfachte Darstellung eines automatisch erstellten Resource-Environment-Modells 88
Abbildung 4-19: Vereinfachte Darstellung eines automatisch erstellten Allocation-Modells 88
Abbildung 4-20: Vereinfachte Darstellung eines automatisch erstellten Usage-Modells 89
Abbildung 4-21: Vereinfachte Darstellung eines automatisch erstellten RDSEFF für MyBean#loadData 91
Abbildung 4-22: Vereinfachte Darstellung eines automatisch erstellten RDSEFF für MyBean#createData 91
Abbildung 4-23: Beispiel für die Spezifikation von gepufferten Aufrufen 94
Abbildung 4-24: Erhebung und Speicherung von Antwortzeitmessungen mit Hilfe des Kieker Frameworks 96
Abbildung 4-25: Instrumentierung des Bytecodes mit Hilfe von AspectJ 97
Abbildung 4-26: Architektur zur Erhebung und Speicherung von Antwortzeitmessungen 98
Abbildung 4-27: Simulation des Performancemodells mit Hilfe von SimuCom 99
Abbildung 4-28: Rückmeldung der Antwortzeitvorhersage zum Entwickler im Quelltexteditor 103
Abbildung 4-29: Rückmeldung der Antwortzeitvorhersage als Tooltip 103
Abbildung 4-30: Rückmeldung der Antwortzeitvorhersage in der Aufrufhierarchie 104
Abbildung 4-31: Treiber und Domains des SPECjEnterprise2010-Benchmark 106
Abbildung 4-32: Überblick über die Schichten der SPECjEnterprise2010 Orders Domain. 107
Abbildung 4-33: Verteilung der gemessenen Antwortzeiten am Beispiel der Komponente OrderSession 114
Abbildung 4-34: Unterscheidung zwischen der Implementierung der Geschäftslogik und des Datenzugriffs 119
Abbildung 5-1: Überblick über die Gestaltung des Experiments 124

Abbildung 5-2: Überblick über angepasste Komponenten der Anwendung Cargo Tracker 132
Abbildung 5-3: Überblick der Hinweise des Plugins für Performancebewusstsein gegenüber den eingefügten Performancebugs 150
Abbildung 5-4: Verteilung der Versuchspersonen anhand des höchsten Abschlusses 151
Abbildung 5-5: Verteilung der Versuchspersonen anhand des Arbeitsverhältnisses 151
Abbildung 5-6: Verteilung der Versuchspersonen anhand der allgemeinen Programmierkenntnisse 152
Abbildung 5-7: Verteilung der Versuchspersonen anhand spezifischer Kenntnisse 153
Abbildung 5-8: Verteilung der behobenen Performancebugs anhand der Experimentgruppen 154
Abbildung 5-9: Behobene Performancebugs je Experimentgruppe 155
Abbildung 5-10: Verteilung der behobenen Performancebugs anhand des höchsten Abschlusses 156
Abbildung 5-11: Verteilung der behobenen Performancebugs anhand des Arbeitsverhältnisses 157
Abbildung 5-12: Verteilung der behobenen Performancebugs anhand der Programmierkenntnisse 157
Abbildung 5-13: Verteilung der behobenen Performancebugs anhand der spezifischen Kenntnisse innerhalb der Testgruppe 158
Abbildung 5-14: Verteilung der behobenen Performancebugs anhand der spezifischen Kenntnisse innerhalb der Kontrollgruppe 159
Abbildung 5-15: Häufigkeitsverteilung der Stichproben über behobene Performancebugs. 159
Abbildung 5-16: Verteilung der Zeit innerhalb der IDE anhand der Experimentgruppen 161
Abbildung 5-17: Verteilung der Zeit beim Navigieren innerhalb der IDE anhand der Experimentgruppen 162
Abbildung 5-18: Verteilung der Anzahl der Kontextänderungen zwischen Java-Dateien ... 162
Abbildung 5-19: Verteilung der Anzahl der Besuche von relevanten Java-Dateien anhand der Experimentgruppen 164
Abbildung 5-20: Verteilung der Anzahl der besuchten Java-Dateien anhand der Experimentgruppen 165
Abbildung 5-21: Verteilung der Anzahl der Änderungen an Dateien anhand der Experimentgruppen 165
Abbildung 5-22: Verteilung der Anzahl ausgeführter Commands anhand der Experimentgruppen 166
Abbildung 5-23: Häufigkeitsverteilung der Stichproben über die Anzahl relevanter Java-Dateien, auf die zugegriffen wurde 167
Abbildung 5-24: Verteilung der Benutzerfreundlichkeit anhand der Experimentgruppen ... 169

Abbildung 6-1: Ansatz für Performancebewusstsein als Anwendungsfall des PMMR 185
Abbildung 6-2: Konzeptuelle Darstellung eines Performance Model Management Repository ... 186
Abbildung 6-3: Beispiel für die Versionierung von Komponenten und Schnittstellen....... 188
Abbildung 6-4: Verwaltung von Komponenten und Schnittstellen ... 189
Abbildung 6-5: Umrechnung der Ressourcenbedarfe während dem Check-in und Check-out ... 191
Abbildung 6-6: Technologische Plattform des Performance Model Management Repository ... 193
Abbildung 6-7: Erweiterung der Ecore-Instanz der PCM-Spezifikation ... 194
Abbildung 6-8: Erweiterung des PCM für die Versionierung von Komponenten und Schnittstellen ... 195
Abbildung 6-9: Erweiterung des PCM für die Verwaltung von Komponenten und Schnittstellen ... 196
Abbildung 6-10: Erweiterung des PCM für die Übertragung von Änderungen an Versionsspezifikationen ... 197
Abbildung 6-11: Erweiterung des PCM für die Verwaltung von Ressourcenbedarfe 199
Abbildung 6-12: Import von Versionsspezifikationen aus existierenden PCM-Instanzen ... 200
Abbildung 6-13: Export von Versionsspezifikationen nach existierenden PCM-Instanzen . 201
Abbildung 6-14: Integration des PMMR mit dem Ansatz für Performancebewusstsein 202
Abbildung 6-15: Beispiel für Änderungen zwischen zwei Durchläufen des Ansatzes für Performancebewusstsein ... 204
Abbildung 6-16: Optimierungspotenzial durch die Wiederverwendung von zwischengespeicherten Komponentenspezifikationen... 207
Abbildung 7-1: Erweiterung des Ansatzes für Performancebewusstsein für den Austausch von Optimierungsmaßnahmen ... 218
Abbildung 7-2: Ökosystem für den Austausch von Optimierungsmaßnahmen zwischen Communities ... 220

Tabellenverzeichnis

Tabelle 1-1: Evaluationsmethoden der gestaltungsorienteierten Forschung 8

Tabelle 2-1: Phasen der Entwicklung von Komponenten und Systemen 26

Tabelle 2-2: Abgrenzung zwischen Enterprise Service Bus und API Gateway 30

Tabelle 3-1: Messbasierte Ansätze für die Unterstützung von Performancebewusstsein der Softwareentwickler 45

Tabelle 3-2: Modellbasierte Ansätze für die Unterstützung von Performancebewusstsein der Softwareentwickler 50

Tabelle 3-3: Komponentenbasierte Ansätze für das Reverse Engineering von Performancemodellen 54

Tabelle 4-1: Berücksichtigung der Einflussfaktoren auf die Performance von Komponenten durch den Ansatz für Performancebewusstsein 66

Tabelle 4-2: Klassifizierung von internen und externen Komponenten 69

Tabelle 4-3: Maße der zentralen Tendenz 74

Tabelle 4-4: Für die Evaluierung der Vorhersagegenauigkeit ausgewählte EJB-Methoden und deren Eigenschaften 109

Tabelle 4-5: Ergebnisse der Evaluierung bei Verwendung des Medians für die Abbildung des Antwortzeitverhaltens 111

Tabelle 4-6: Ergebnisse der Evaluierung bei Verwendung des 95-Perzentils für die Abbildung des Antwortzeitverhaltens 113

Tabelle 5-1: Performance Antipatterns 130

Tabelle 5-2: Eingeführte Performancebugs und die davon betroffenen Klassen und Methoden 135

Tabelle 5-3: Protokollierung von Ereignissen innerhalb der IDE 139

Tabelle 5-4: Gegenüberstellung des Experimentaufbaus aus der vorliegenden Arbeit und der von Horký et al. (2015) 146

Tabelle 5-5: Überblick über angemeldete Versuchspersonen 147

Tabelle 5-6: Iterationen bei der Durchführung des Experiments 148

Tabelle 5-7: Statistische Maße zur Beschreibung der Stichproben über behobene Performancebugs 154

Tabelle 5-8: Tests auf Normalverteilung der Stichproben über behobene Performancebugs 160

Tabelle 5-9: Hypothesentest über behobene Performancebugs 160

Tabelle 5-10: Statistische Maße zur Beschreibung der Stichproben über die Zeit innerhalb der IDE 161

Tabelle 5-11: Tests auf Normalverteilung der Stichproben über der Zeit innerhalb der IDE 163

Tabelle 5-12: Hypothesentest über die Zeit innerhalb der IDE.. 163

Tabelle 5-13: Statistische Maße zur Beschreibung der Stichproben über die Anzahl der Besuche von relevanten Java-Dateien 164

Tabelle 5-14: Tests auf Normalverteilung der Stichproben über die Anzahl der Besuche von relevanten Java-Dateien 167

Tabelle 5-15: Hypothesentest über die Anzahl relevanter Java-Dateien, auf die zugegriffen wurde.. 168

Tabelle 5-16: Statistische Maße zur Beschreibung der Stichproben über die Benutzerfreundlichkeit.. 169

Tabelle 5-17: Tests auf Normalverteilung der Stichproben über Benutzerfreundlichkeit 170

Tabelle 5-18: Hypothesentest über Benutzerfreundlichkeit 170

Abkürzungsverzeichnis

API ... *Application Programming Interface*
AST ... *Abstract Syntax Tree*
CDI ... *Context and Dependency Injection*
CPS ... *Cyber-Physical Systems*
DI ... *Dependency Injection*
ECP ... *EMF Client Platform*
EE ... *Enterprise Edition*
EJB ... *Enterprise JavaBeans*
EMF ... *Eclipse Modeling Framework*
ESB ... *Enterprise Service Bus*
GAST ... *Generalised Abstract Syntax Tree*
GQM ... *Goal Question Metric*
HTML ... *Hypertext Markup Language*
HTTP ... *Hypertext Transfer Protocol*
IaaS ... *Infrastructure as a Service*
IDE ... *Integrated Development Environment*
IIoT ... *Industrial Internet of Things*
IoT ... *Internet of Things*
IS ... *Informationssystem*
JaMoPP ... *Java Model Parser and Printer*
JMS ... *Java Message Service*
JPA ... *Java Persistence API*
KDM ... *Knowledge Discovery Metamodel*
LQN ... *Layered Queueing Network*
MOF ... *Meta Object Facility*
MQTT ... *Message Queuing Telemetry Transport*
OMG ... *Object Management Group*
PCI ... *Performance-Curve-Integration*
PCM ... *Palladio Component Model*
PMMR ... *Performance Model Management Repository*
QFD ... *Quality Function Deployment*
QoS ... *Quality of Service*
RDSEFF ... *Resource Demanding Service Effect Specification*
REST ... *Representational-State-Transfer*
SCM ... *Software Configuration Management*
SISSy ... *Structural Investigation of Software Systems*
SLA ... *Service Level Agreement*
SLOC ... *Source Lines of Code*
SOA ... *Serviceorientierte Architekturen*
SOAP ... *Simple Object Access Protocol*
SoMoX ... *SOftware MOdel eXtractor*
SPE ... *Software Performance Engineering*
SPEC ... *Standard Performance Evaluation Corporation*
SUS ... *System Usability Scale*

UML .... *Unified Modeling Language*
URL .... *Uniform Resource Locator*
VM .... *Virtuelle Maschine*
WSDL........ *Web Services Description Language*
XML .... *Extensible Markup Language*

1 Einleitung

Unternehmensanwendungen unterstützen die Durchführung von Geschäftsprozessen und nehmen mit der voranschreitenden Digitalisierung von Wertschöpfungsketten eine immer wichtigere Rolle ein. Aufgrund einer zunehmenden Komplexität der abgebildeten Geschäftsprozesse werden Unternehmensanwendungen oft als komponentenbasierte Systeme umgesetzt. Eine weit verbreitete Technologie für die Implementierung solcher Unternehmensanwendungen ist die Java Enterprise Edition (EE).

Neben der funktionalen Korrektheit stellt Performance ein wesentliches Qualitätsmerkmal von Unternehmensanwendungen dar. Die Performance einer Unternehmensanwendung kann durch die Metriken Antwortzeit, Durchsatz und Ressourcenverbrauch beschrieben werden (Becker et al. 2013). Antwortzeit beschreibt die Zeit, die ein Nutzer auf die Rückmeldung eines Dienstes warten muss. Die Menge an Operationen, die innerhalb einer bestimmten Zeit durchgeführt werden, wird als Durchsatz beschrieben. Während der Verarbeitung einer Aufgabe werden Ressourcen in einem bestimmten Umfang verbraucht. Unter bestimmten Umständen können sich diese drei Metriken auch gegenseitig beeinflussen. Ein hoher Ressourcenverbrauch kann eine parallel laufende Verarbeitung verlangsamen und gleichzeitig auch die resultierende Antwortzeit erhöhen.

1.1 Motivation

Die Relevanz der drei Performancemetriken hängt stark von der Perspektive auf die Unternehmensanwendung ab. Mit der steigenden Akzeptanz von Cloud-Diensten, wie beispielsweise Infrastructure as a Service (IaaS), werden auch immer mehr Unternehmensanwendungen auf extern betriebene Infrastrukturen migriert. Abgerechnet werden Cloud-Infrastrukturen je Zeiteinheit oder abhängig von der Intensität der Nutzung. Eine schlechte Performance schlägt sich damit mehr denn je auf die Betriebskosten nieder. Der Kunde eines IaaS-Dienstleisters könnte daher primär an einem geringen Ressourcenverbrauch interessiert sein. Die Menge der verarbeiteten Aufgaben könnte für einen Geschäftsprozessverantwortlichen wiederum wichtiger sein. Aus Nutzersicht stellt die Antwortzeit jedoch die wichtigste Performancemetrik dar. Mitarbeiter eines Unternehmens verschwenden durch langsame Unternehmensanwendungen kostbare Arbeitszeit. Die Kunden eines Handelsunternehmens haben im Gegensatz zu den Mitarbeitern die Möglichkeit bei der Konkurrenz einzukaufen. Die Verschlechterung der Antwortzeit einer Internet-Seite kann den Umsatz eines Unternehmens negativ beeinflussen (Kohavi/Longbotham 2007).

Für die Vermeidung von gravierenden Performanceproblemen im produktiven Betrieb ist das Testen der Performance von Unternehmensanwendungen vor deren Veröffentlichung essentiell (Weyuker/Vokolos 2000). Die Durchführung von Performancetests setzt jedoch die Verfügbarkeit lauffähiger Softwareartefakte und eine repräsentative Testumgebung voraus. Hardware, Testdaten und Workload müssen der produktiven Umgebung ähneln. Die Bereitstellung einer entsprechenden Testumgebung verursacht daher Kosten und ist zeitaufwändig. Performancetests müssen darüber hinaus detailliert geplant werden und sollten sich über längere Zeiträume erstrecken (Woodside et al. 2007). Aufgrund der Aufwände und Voraussetzungen werden Performancetests häufig erst gegen Ende des Entwicklungslebenszyklus durchgeführt. Je später

Probleme entdeckt werden, desto aufwändiger ist deren Behebung. Die Behebung eines Fehlers aus der Anforderungsspezifikation während der Testphase kann um den Faktor fünf aufwändiger sein, als während der Implementierung (Stecklein et al. 2004). Performanceprobleme sollten daher so früh wie möglich adressiert werden.

Die Erstellung performanter Softwaresysteme steht im Fokus des Software Performance Engineering (SPE) (Smith 1993). Mit Hilfe von Performancemodellen können Architekturentscheidungen schon in frühen Entwicklungsphasen einer Unternehmensanwendung evaluiert werden. Auf Basis von Architekturspezifikationen können bereits während dem Entwurf Performancemodelle erstellt werden (Cortellessa/Frittella 2007).

Die Erstellung von Performancemodellen ist aufwändig und erfordert Expertenwissen, was dazu führt, dass SPE selten in der Praxis angewendet werden (Woodside et al. 2007). SPE-Ansätze adressieren deshalb auch die automatische Überführung von Architekturmodellen nach Performancemodellen (Bertolino/Mirandola 2004). Aufgrund der formalen Anforderungen an den Architekturmodellen und der vorausgesetzten Modellierungskenntnisse bei Entwicklern werden auch diese Ansätze selten in der Praxis eingesetzt (Koziolek 2010). Die permanente Weiterentwicklung von Architekturspezifikationen stellt eine zusätzliche Hürde in der Erstellung und der Pflege von Performancemodellen dar.

Während der Implementierung haben Entwickler die Möglichkeit die Performance einzelner ausführbaren Komponenten mit Hilfe von messbasierten Ansätzen zu evaluieren (Koziolek 2010). Ansätze für das Reverse Engineering von Quelltext unterstützen die Erstellung von Performancemodellen basierend auf der tatsächlichen Implementierung einer Unternehmensanwendung (Krogmann 2010). Eine kontinuierliche Evaluation der Performance kann die Produktivität eines Softwareentwicklers jedoch negativ beeinflussen. Solange Performanceevaluationen nicht automatisiert durchführbar sind, werden diese von Softwareentwickler in der Praxis nicht angewendet. Das Konzept des Performancebewusstseins adressiert genau diesen Aspekt. Einsichten über die Performance von Softwareanwendungen sollen dem Entwickler bereitgestellt werden und ihn dazu befähigen auf diese Beobachtungen zu reagieren (Tůma 2014).

1.2 Problemstellung

Aus der zuvor beschriebenen Motivation ergibt sich die Anforderung der frühzeitigen Unterstützung von Performancebewusstsein für Softwareentwickler im Kontext komplexer Unternehmensanwendungen mit Hilfe von automatisierten Performanceevaluationen (siehe Abbildung 1-1). Die Abdeckung der einzelnen Aspekte durch existierende Ansätze wird im Folgenden beschrieben.

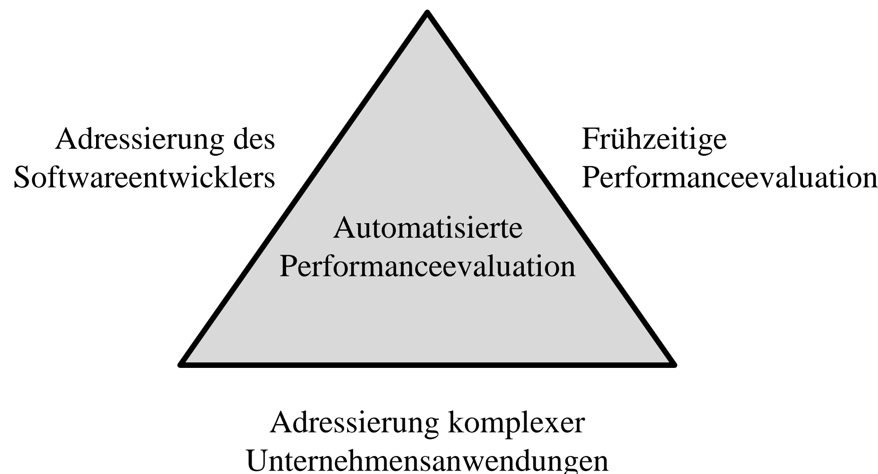


Abbildung 1-1: ***Frühzeitige Unterstützung von Softwareentwickler mit automatisierten Performanceevaluationen im Kontext komplexer Unternehmensanwendungen***
Quelle: Eigene Darstellung

Frühzeitige Performanceevaluation

Ansätze für Performancebewusstsein zielen auf die Automatisierung von Performanceevaluationen auf unterschiedlichen Ebenen, wie Lastgenerierung, Messung, Modellierung, Simulation, Auswertung oder Visualisierung, ab. Setzen entsprechende Ansätze lauffähige Anwendungen voraus, können diese erst nach der Implementierung angewendet werden (Parsons 2007; Wert 2015). Für die Durchführung der Evaluationen werden darüber hinaus Testumgebungen benötigt.

Komplexe Unternehmensanwendungen

Vor der Verfügbarkeit einer lauffähigen Anwendung können relevante Treiber der zu erwartenden Gesamtperformance durch eine systematische Ausführung einzelner Funktionen evaluiert werden (Horký et al. 2015). Aufgrund dieser Einsichten können Entwickler beispielsweise alternative Implementierungen gegenüberstellen. Unternehmensanwendungen können jedoch komplexe Abhängigkeiten zu externen Diensten aufweisen. Einzelne Komponenten können einander wiederverwenden. Objekte müssen von Datenbanken ausgelesen werden können. Laufzeitumgebungen müssen auch bestimmte Konfigurationen aufweisen. Eine automatisierte Ausführung von Komponenten ist dadurch nur schwer realisierbar.

Adressierung des Softwareentwicklers

Andere Ansätze automatisieren zwar wichtige Aspekte der Performanceevaluation, setzten jedoch bei den Entwicklern Expertenwissen und die Nutzung externer Werkzeuge voraus (Westermann 2014). Erfahrungen in der Domäne des SPE und im Einsatz entsprechender Werkzeuge kann nicht bei Entwicklern vorausgesetzt werden. Darüber hinaus kann es auch sein, dass Entwickler Ansätzen des SPE nicht vertrauen (Woodside et al. 2007).

1.3 Ziele der Arbeit

Die Untersuchung, wie Softwareentwickler von komponentenbasierten Unternehmensanwendungen mit Einsichten über die Performance ihrer Implementierung unterstützt werden können,

ist das übergeordnete Ziel der vorliegenden Arbeit. Aus den zuvor beschriebenen Problemstellungen und dem übergeordneten Ziel ergeben sich mehrere untergeordnete Ziele, welche im Folgenden beschrieben werden.

Untersuchung von Komponenten

Einzelne Komponenten komplexer Unternehmensanwendungen durchlaufen zur Erweiterung um neue Funktionen bzw. Korrektur von Fehlern einen kontinuierlichen Lebenszyklus (Brunnert et al. 2014). Zu einem bestimmten Zeitpunkt können diese dabei einen unterschiedlichen Stand aufweisen. Entwickler einer neuen Komponente können beispielsweise existierende Dienste wiederverwenden. Untersucht wird daher, wie Einsichten über die Performance einzelner Komponenten bereitgestellt werden können.

Bewusstsein über das Antwortzeitverhalten

Wie im Rahmen der Motivation der vorliegenden Arbeit schon beschrieben wurde, stellt das Antwortzeitverhalten aus Nutzersicht die wichtigste Performancemetrik dar. Auf der Ebene einer Komponente weist jeder implementierte Dienst ein eigenes Antwortzeitverhalten auf. Untersucht wird daher, wie Einsichten über das Antwortzeitverhalten der einzelnen Dienste einer Komponente bereitgestellt werden können.

Vorhersage des Antwortzeitverhaltens

Wie eingangs beschrieben, können Performanceprobleme einfacher behoben werden, je früher sie identifiziert werden. Sobald eine Softwareversion produktiv eingesetzt wird, verursacht der Einbau jeglicher Korrektur organisatorische Aufwände. Die Messung des Antwortzeitverhaltens von Komponenten setzt eine lauffähige Implementierung und die Bereitstellung einer Testumgebung mit wiederverwendeten Diensten voraus. Der Einsatz von Performancemodelle unterstützt die Abbildung und die Vorhersage des Verhaltens von Komponenten. Untersucht werden soll daher, wie das Antwortzeitverhalten von Komponenten mit Hilfe von Performancemodellen vorhergesagt werden kann, bevor diese ausführbar sind.

Einfluss externer Aufrufe

Neben dem Ressourcenverbrauch zur Laufzeit stellen Aufrufe externer Dienste einen wichtigen Treiber für die Antwortzeit von Komponenten dar (Koziolek 2010). Durch die Analyse des Quelltextes können wiederverwendete Dienste identifiziert und ihr Einfluss auf die Antwortzeit einer Komponente abgeschätzt werden. Untersucht werden soll daher, wie das Antwortzeitverhalten von Komponenten aufgrund von wiederverwendeten Diensten vorhergesagt werden kann.

Verwaltung von Performancemodellen

Die Erstellung und Pflege von Performancemodellen wird aufgrund von komplexen Architekturen, Softwarelebenszyklen und Governance-Strukturen (Brunnert et al. 2014) zunehmend aufwändiger und kann dadurch eine Hürde für die Akzeptanz in der Praxis darstellen. Untersucht werden soll daher, wie Performancemodelle zwischengespeichert und innerhalb komplexer Unternehmensumgebungen ausgetauscht werden können.

Integration in der Entwicklungsumgebung

Damit Performanceevaluationen einfach durch den Entwickler verarbeitet werden können und dessen Arbeitsfluss nicht gestört wird, müssen diese möglichst wenig Aufwand verursachen und kein Expertenwissen voraussetzen. Untersucht werden soll daher, wie Antwortzeitvorhersagen möglichst automatisiert und in die Entwicklungsumgebung integriert werden können.

Einfluss auf Softwareentwickler

Die Akzeptanz von Ansätzen für Performancebewusstsein in der Industrie setzt neben deren korrekte Funktionsweise auch belastbare Aussagen über die Auswirkung durch deren Einsatz voraus (Brunnert et al. 2015b). Untersucht werden soll daher, wie die Performance der Implementierung von Komponenten durch die Unterstützung von Softwareentwickler mit Performancebewusstsein beeinflusst wird.

1.4 Forschungsfragen

Die Realisierung der zuvor formulierten Ziele wird von drei Forschungsfragen geleitet. Zuerst werden die Vorhersage von Antwortzeiten der Dienste von Komponenten und die Rückmeldung der Ergebnisse an Softwareentwickler adressiert. Als nächstes wird der Einfluss der bereitgestellten Vorhersagen auf Entwickler und ihre Implementierungen evaluiert. Abschließend werden die Verwaltung und der Austausch von Performancemodellen untersucht. Im Folgenden werden die Forschungsfragen im Detail beschrieben.

Forschungsfrage 1 ***Wie können Antwortzeitvorhersagen für Softwarekomponenten aufgrund von wiederverwendeten Diensten durchgeführt und für die Schaffung von Performancebewusstsein bei Softwareentwicklern bereitgestellt werden?***

Zur Beantwortung der ersten Forschungsfrage wird ein Konzept für die Unterstützung von Performancebewusstsein bei Softwareentwickler komponentenbasierter Unternehmensanwendungen entwickelt. Das Konzept adressiert die Abbildung des Quelltextes von Komponenten als Performancemodell, die Durchführung von Antwortzeitmessungen für wiederverwendete Dienste und deren Integration in das resultierende Modell, die Vorhersage des Antwortzeitverhaltens und die Rückmeldung der Ergebnisse. Das Ergebnis wird beispielhaft für die Unterstützung von Antwortzeitvorhersagen für Java-EE-Komponenten umgesetzt. Im Rahmen eines kontrollierten Experiments wird die Präzision der Antwortzeitvorhersagen evaluiert.

Forschungsfrage 2 ***Welche Verbesserungen in der Antwortzeit der Dienste einer Komponente können durch die Unterstützung von Performancebewusstsein bei Softwareentwickler erzielt werden?***

Das Ziel der zweiten Forschungsfrage ist es den Einfluss des zuvor entwickelten Ansatzes für Performancebewusstsein auf die Antwortzeit der vom Entwickler implementierten Software zu quantifizieren. Versuchspersonen optimieren im Rahmen einer Untersuchung das Antwortzeitverhalten existierender Java-EE-Komponenten. Eine Gruppe von Personen haben dabei die Möglichkeit Vorhersagen über das Antwortzeitverhalten ihrer Implementierungen abzurufen.

Durch die Gegenüberstellung mit den Ergebnissen einer Kontrollgruppe wird der Einfluss von Performancebewusstsein gemessen.

Forschungsfrage 3 ***Wie können Bausteine von Performancemodellen innerhalb einer Unternehmensumgebung versioniert, verwaltet und ausgetauscht werden?***

Zur Beantwortung der dritten Forschungsfrage wird ein Konzept für das Management von Performancemodellen einzelner Komponenten entwickelt. Das Konzept adressiert die Spezifikation unterschiedlicher Versionen, die Verwaltung von hardwarespezifischen Ressourcenbedarfe und die Integration mit Modellierungsumgebungen. Das Ergebnis wird beispielhaft instanziiert und mit dem zuvor entwickelten Ansatz für Performancebewusstsein integriert. Durch die Integration können bei der Berechnung von Antwortzeitvorhersagen Bausteine von Performancemodellen wiederverwendet werden.

1.5 Forschungsansatz

Im Rahmen der vorliegenden Arbeit wird ein Ansatz für die Unterstützung von Performancebewusstsein entwickelt, umgesetzt und evaluiert. Dementsprechend lässt sich die Arbeit im Bereich der gestaltungsorientierten Forschung einordnen. Das Vorgehen zur Beantwortung der Forschungsfragen orientiert sich deshalb an dem Information Systems Research Framework aus Abbildung 1-2 nach Hevner et al. (2004). Im Zentrum des Frameworks steht die Untersuchung von Informationssystemen (IS), zu denen auch die hier entwickelten Artefakte gehören. Das Umfeld der IS-Forschung besteht aus Personen, Organisationen bzw. Technologien und spezifiziert Problemstellungen. Personen nehmen anhand ihrer Rollen, Kompetenzen und Eigenschaften Anforderungen in Unternehmen wahr. Sie beurteilen Anforderungen im Kontext der Strategie, Kultur und Prozesse eines Unternehmens. Anforderungen beziehen sich auf bestimmte technologische Aspekte, wie beispielsweise Infrastrukturen oder Anwendungen. IS-Forschung untersucht aus dem Umfeld generierte Problemstellungen in zwei Phasen. Theorien werden entwickelt und begründet. Im Gegensatz dazu werden Artefakte hergestellt und anschließend evaluiert. Durch die Adressierung von Problemstellungen aus dem Umfeld mit dem Artefakt, wird die Relevanz der IS-Forschung sichergestellt. Eine Wissensbasis stellt Grundlagen und Vorgehensweisen für die Durchführung von IS-Forschung bereit. Durch die Anwendung der Wissensbasis wird die Stringenz der IS-Forschung sichergestellt. Die Ergebnisse der IS-Forschung werden durch die Anwendung im Umfeld beurteilt und ergänzen die existierende Wissensbasis.

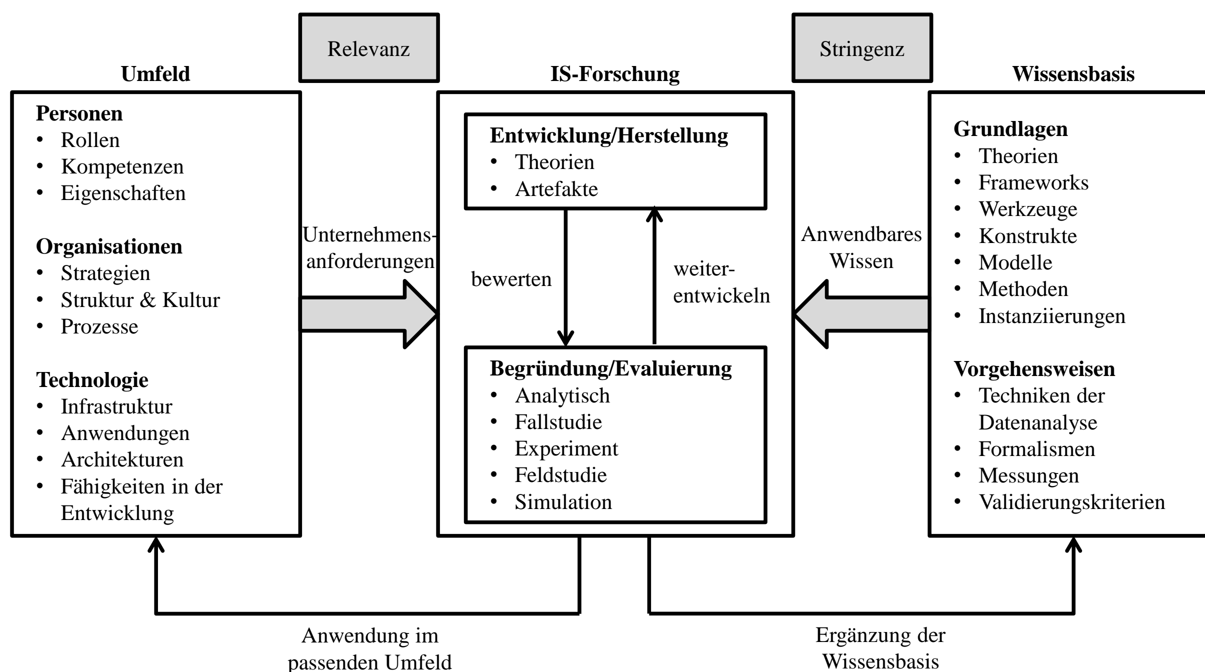


Abbildung 1-2: Information Systems Research Framework nach Hevner et al.
Quelle: In Anlehnung an Hevner et al. (2004, S. 80)

Die Entwicklung von Unternehmensanwendungen repräsentiert die Umwelt der vorliegenden Arbeit. Sowohl interne Entwicklungsabteilungen als auch Softwaredienstleister verfolgen das Ziel Softwarelösungen in einem bestimmten Funktionsumfang unter Einhaltung von nichtfunktionalen Anforderungen zu implementieren. Aufgrund von Zeit- und Kostendruck sowie mangelnder Kompetenzen können SPE-Aktivitäten nur im geringen Umfang durchgeführt werden. Die aus diesem Umfeld resultierende Problemstellung ist die Notwendigkeit von Performancebewusstsein für die Unterstützung von Entwickler mit Einsichten über die Performance ihrer Implementierungen. Die Ergebnisse der vorliegenden Arbeit werden anhand von Versuchspersonen aus dem Umfeld bewertet.

Zur Beantwortung der Forschungsfragen wird das SPE als Wissensbasis herangezogen. Die Einflussfaktoren auf die Performance von Komponenten werden von Koziolek (2010) beschrieben. Meta-Modelle unterstützen die Spezifikation von Performancemodellen (Becker et al. 2009; Bolch et al. 2006). Verschiedene Ansätze unterstützen eine automatische Erstellung solcher Performancemodelle (Krogmann 2010; Brunnert et al. 2013). Ansätze für Performanceevaluationen können in existierende Entwicklungsumgebungen für Java-basierte Software integriert werden (Gallardo et al. 2003). Monitoring-Werkzeuge unterstützten die Durchführung von Performancemessungen (van Hoorn et al. 2009). Versionskontrollsysteme implementieren Funktionen für die Verwaltung von Modellinstanzen (Altmanninger et al. 2009). Die Ergebnisse der vorliegenden Arbeit ergänzen die Wissensbasis im Bereich Performancebewusstsein um neue Konzepte und Erkenntnisse.

Nach Hevner et al. (2004) können Artefakte u.a. hinsichtlich Funktionalität, Vollständigkeit, Konsistenz, Präzision, Performance, Zuverlässigkeit, Benutzerfreundlichkeit und der Eignung für die Organisation evaluiert werden. Methoden für die Evaluierung gestaltungsorientierter Forschung sind in Tabelle 1-1 aufgelistet. Die vorliegende Arbeit evaluiert die Präzision von

Antwortzeitvorhersagen sowie die Benutzerfreundlichkeit und die Eignung des entwickelten Ansatzes mit Hilfe von kontrollierten Experimenten.

Beobachtung	Fallstudie: Detaillierte Beobachtung des Artefakts in der Unternehmensumgebung.
	Feldstudie: Überwachung der Nutzung des Artefakts über mehrere Projekte.
Analyse	Statische Analyse: Untersuchung der Struktur des Artefakts bzgl. statischer Eigenschaften (z.B. Komplexität).
	Architekturanalyse: Untersuchung der Eignung des Artefakts für eine technische Informationssystemarchitektur.
	Optimierung: Demonstration der optimalen Eigenschaften des Artefakts oder Bereitstellung von Schranken für ein optimales Verhalten des Artefakts.
	Dynamische Analyse: Untersuchung der Nutzung des Artefakts bzgl. dynamischer Eigenschaften (z.B. Performance).
Experiment	Kontrolliertes Experiment: Untersuchung der Eigenschaften des Artefakts in einer kontrollierten Umgebung (z.B. Bedienbarkeit).
	Simulation – Ausführung des Artefakts mit Hilfe künstlicher Daten.
Test	Funktionaler (Black Box) Test: Ausführung der Schnittstellen des Artefakts, um Ausfälle zu erkennen und Defekte zu identifizieren.
	Struktureller (White Box) Test: Untersuchung der Abdeckung bestimmter Metriken in der Implementierung des Artefakts (z.B. Ausführungspfade).
Erklärung	Argumentation: Anwendung der Informationen aus der Wissensbasis (z.B. Literatur) um Argumente für die Nützlichkeit des Artefakts zu konstruieren.
	Szenario: Erstellung detaillierter Szenarien um die Nützlichkeit des Artefakts zu demonstrieren.

Tabelle 1-1: Evaluationsmethoden der gestaltungsorienteierten Forschung
Quelle: In Anlehnung an Hevner et al. (2004, S. 86)

Das Information Systems Research Framework aus Abbildung 1-1 beschreibt nur ein grundlegendes Vorgehen. Für die Durchführung gestaltungorientierter Forschung haben Peffers et al. (2006) einen genauen Prozess entwickelt. Dieser besteht aus den folgenden sechs Aktivitäten (Peffers et al. 2006, S. 89ff):

1. **Problemidentifikation und Motivation:** Das Ziel dieser Aktivität ist die Definition eines bestimmten Problems und die Begründung des Nutzens einer Lösung.

2. **Definition der Ziele einer Lösung:** Ausgehend von der Problemdefinition sollen Ziele einer Lösung abgeleitet werden. Ziele können dabei sowohl qualitativ als auch quantitativ formuliert werden.
3. **Konzeption und Entwicklung:** Die anvisierte Lösung soll in Form eines Artefakts, wie z.B. ein Modell oder eine Methode, umgesetzt werden.
4. **Demonstration:** Durch den Einsatz des Artefakts soll dessen Wirksamkeit bei der Lösung des Problems demonstriert werden.
5. **Evaluation:** Die während der Demonstration gemessene Wirksamkeit des Artefakts soll im Rahmen dieser Aktivität der Zielspezifikation der Lösung gegenübergestellt werden. Abhängig von dem Ergebnis kann zur Verbesserung des Artefakts anschließend eine Iteration zur dritten Aktivität vorgenommen werden.
6. **Kommunikation:** Im Rahmen dieser Aktivität sollen das adressierte Problem, die anvisierte Lösung, das Artefakt und die Ergebnisse der Evaluation einer relevanten Zielgruppe kommuniziert werden.

Eine Zuordnung der sechs Aktivitäten nach Peffers et al. (2006) zu den einzelnen Kapitel der vorliegenden Arbeit wird im folgenden Abschnitt beschrieben.

1.6 Aufbau der Arbeit

Die forschungsleitenden Fragestellungen werden durch separate Kapitel adressiert. Diese sind in sich geschlossen, werden jedoch über den resultierenden Ansatz für Performancebewusstsein miteinander integriert. Eine Zuordnung der Kapitel zu Forschungsfragen ist in Abbildung 1-3 dargestellt. Die Inhalte der einzelnen Kapitel werden im Folgenden beschrieben.

In Kapitel 1 wird die Relevanz früher Performanceevaluationen während der Entwicklung von Unternehmensanwendungen begründet. Aus der Motivation wird abgeleitet, dass die beschriebenen Anforderungen durch das Konzept des Performancebewusstseins adressiert werden. Als nächstes wird die Problemstellung der frühzeitigen Unterstützung von Softwareentwickler mit automatisierten Performanceevaluationen im Kontext komplexer Unternehmensanwendungen skizziert und begründet, wieso diese in der Forschung unzureichend adressiert wird. Zur Adressierung der Problemstellung im Rahmen der vorliegenden Arbeit, werden Ziele spezifiziert. Daraus werden drei forschungsleitende Fragestellungen abgeleitet. Anschließend wird das Vorgehen zur Beantwortung der Fragestellungen auf Basis gestaltungsorientierter Forschung beschrieben. Dieses Kapitel adressiert die Aktivitäten der Problemidentifikation/Motivation und der Zieldefinition nach Peffers et al. (2006) auf einer höheren Abstraktionsebene. Bei der Beantwortung der einzelnen Forschungsfragen werden diese jeweils näher spezifiziert.

Kapitel 2 beschreibt zuerst die theoretischen Grundlagen zu den Schwerpunkten der vorliegenden Arbeit. Im Fokus der Arbeit stehen vor allem die Konzepte des Performancebewusstseins und der komponentenbasierten Softwareentwicklung. Darüber hinaus werden relevante Technologien, wie Java EE und das Palladio Component Model vorgestellt.

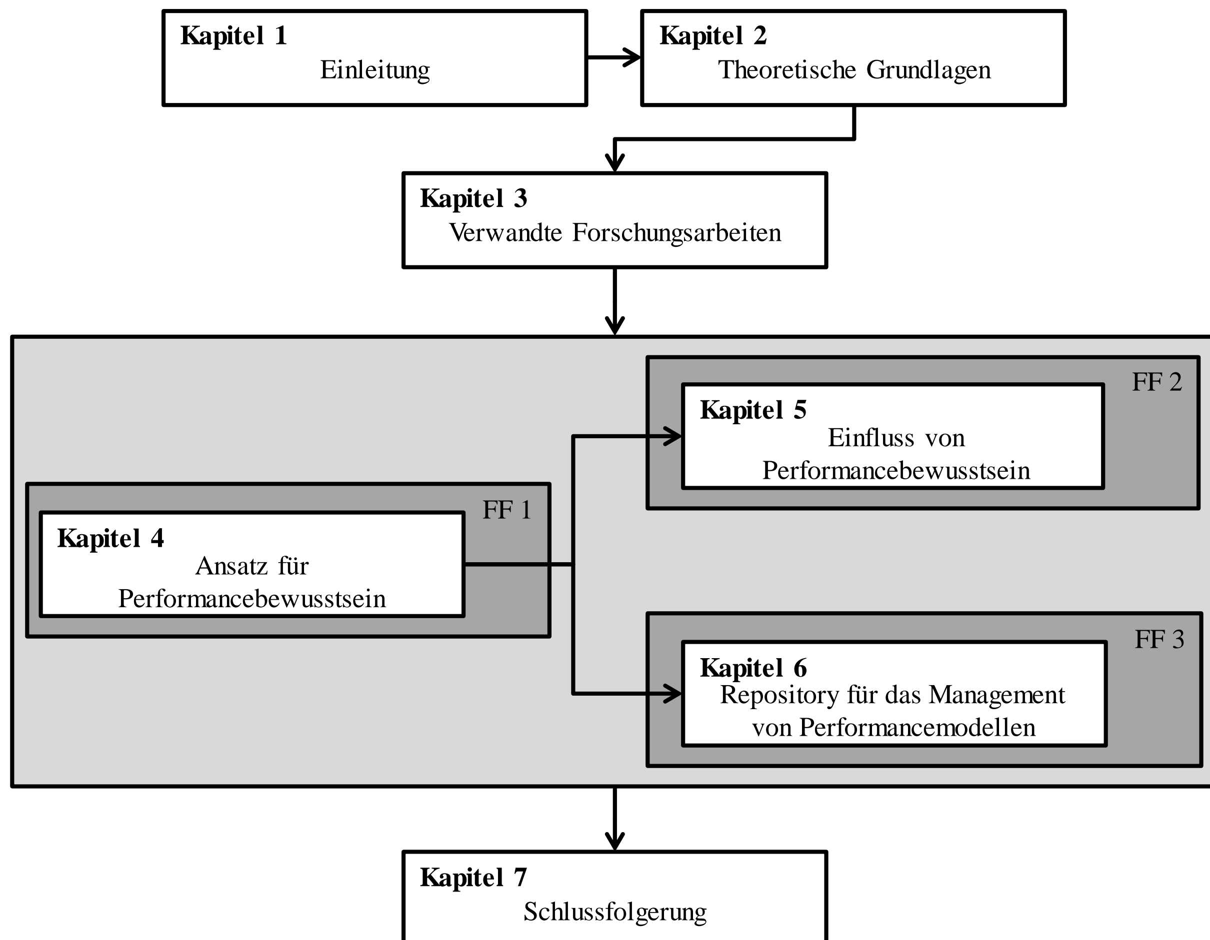


Abbildung 1-3: Struktur der Arbeit und Zuordnung der Kapitel zu Forschungsfragen
Quelle: Eigene Darstellung

In Kapitel 3 werden verwandte Forschungsarbeiten in vier Bereiche unterteilt und beschrieben. Aus dem Forschungsbereich Performancebewusstsein werden modell- und messbasierte Ansätze vorgestellt. Als nächstes werden Ansätze für das Reverse Engineering von Komponenten beschrieben. Danach werden Versionskontrollsysteme mit ähnlichen Zielsetzungen wie das Repository für Performancemodelle beschrieben. Abschließend werden Evaluationen anderer Ansätze für die Bereitstellung von Einsichten für Entwickler innerhalb der IDE vorgestellt. Im Rahmen der Zusammenfassung werden die Aspekte der untersuchten Systeme und der eingesetzten Evaluationsmethoden diskutiert.

Die folgenden drei Kapitel stellen den Hauptteil der Arbeit dar. Kapitel 4 begründet zuerst die Relevanz der Vorhersage von Antwortzeiten für Komponenten auf Basis von wiederverwendeten Diensten. Die übergreifenden Ziele der adressierten Forschungsfrage werden konkretisiert. Der hier beschriebene Kontext definiert grundlegende Rahmenbedingungen und Annahmen bei der Entwicklung des Ansatzes. Das anhand der verfeinerten Ziele strukturierte Konzept des Ansatzes für Performancebewusstsein wird als nächstes beschrieben. Das Konzept adressiert die Abbildung des Quelltextes von Komponenten als Performancemodell. Die Aspekte der Struktur und des Kontrollflusses von Komponenten, sowie die Abbildung der Parametrisierung, von Puffermechanismen und Antwortzeiten externer Dienste werden berücksichtigt. Die Visualisierung der Ergebnisse von Antwortzeitvorhersagen wird skizziert. Die technische

Realisierung des Konzepts wird als nächstes anhand von vier Phasen beschrieben. Das resultierende Artefakt wird hinsichtlich der Genauigkeit der Antwortzeitvorhersagen im Rahmen eines Experiments evaluiert. Abschließend werden die Ergebnisse der ersten Forschungsfrage zusammengefasst.

In Kapitel 5 wird die Evaluation des Ansatzes für Performancebewusstsein im Rahmen eines kontrollierten Experiments beschrieben. Ziel der Evaluation ist es, den Einfluss der Verfügbarkeit von Performancebewusstsein auf die Antwortzeit von Komponenten zu quantifizieren. Zuerst wird die Relevanz der Durchführung der Evaluation begründet. Die übergreifenden Ziele der adressierten Forschungsfrage werden konkretisiert. Der hier beschriebene Kontext definiert grundlegende Rahmenbedingungen und Annahmen bei der Gestaltung des Experiments. Als nächstes wird ein Entwurf des Experiments vorgestellt. Ausgehend von den Hypothesen und Variablen werden der Aufbau des Experiments, die Auswahl von Versuchspersonen sowie eingesetzte Objekte beschrieben. Hinsichtlich der Gestaltung werden auch noch Verfahren für die Erhebung und Auswertung von Daten sowie für die Sicherstellung der Validität beschrieben. Als nächstes werden die Durchführung und der tatsächliche Verlauf des Experiments beschrieben. Die Auswertung anhand der Eigenschaften sowie des Verhaltens von Versuchspersonen, der Lösungen sowie der Benutzerfreundlichkeit strukturiert. Die Ergebnisse des Experiments werden abschließend interpretiert und zusammengefasst.

Kapitel 6 begründet zuerst die Relevanz eines Repository für das Management von Performancemodellen. Die übergreifenden Ziele der dritten Forschungsfrage werden anschließend konkretisiert. Der hier beschriebene Kontext definiert grundlegende Rahmenbedingungen und Annahmen bei der Entwicklung des Repository. Das anhand der verfeinerten Ziele strukturierte Konzept für die Verwaltung von Performancemodellen wird als nächstes beschrieben. Die Versionierung von Spezifikationen, die Verwaltung von hardwarespezifischen Ressourcenbedarfen und die Synchronisation von Änderungen mit Modellierungsumgebungen werden hier adressiert. Die technische Realisierung des Repository wird als nächstes beschrieben. Das resultierende Artefakt wird mit dem Ansatz für Performancebewusstsein aus dem vorherigen Kapitel integriert. Es wird beschrieben, wie bei der Berechnung von Antwortzeitvorhersagen Performancemodelle mit Hilfe des Repository zwischengespeichert werden können. Abschließend werden die Ergebnisse der zweiten Forschungsfrage zusammengefasst.

In Kapitel 7 werden zuerst die Ergebnisse der Forschungsfragen zusammengefasst. Als nächstes werden Limitationen der vorliegenden Arbeit diskutiert. Die Vision einer Vernetzung dezentraler Entwicklungsumgebungen für die Erfassung und den Austausch von Optimierungsmaßnahmen wird abschließend als Ausblick vorgestellt.

2 Theoretische Grundlagen

Aus den Bestrebungen zur Zentralisierung von Rechen- und Speicherleistung in Verbindung mit einer gleichzeitigen Kommerzialisierung und Standardisierung der damit zusammenhängenden Dienste ist das Konzept des *Cloud Computing* hervorgegangen. Infrastrukturressourcen, wie den virtualisierten Rechnern, Speichermedien und Netzwerkkomponenten, flexible Applikationsumgebungen oder abgekapselte Anwendungen werden im Rahmen von Cloud Computing als Dienste über das Internet zur Verfügung gestellt. Für die Verarbeitung von variablen Workloads können somit Cloud-Dienste flexibel in Anspruch genommen und in Abhängigkeit der anfallenden Nutzungsintensität bezahlt werden.

Mit dem Aufkommen mobiler Endgeräte und dem *Internet of Things* (IoT) sind neue Paradigmen, wie dem Edge Computing, dem Fog Computing und der Cloudlets entstanden (Satyanarayanan 2017). Eine Abgrenzung dieser Begriffe ist in Abbildung 2-1 dargestellt.

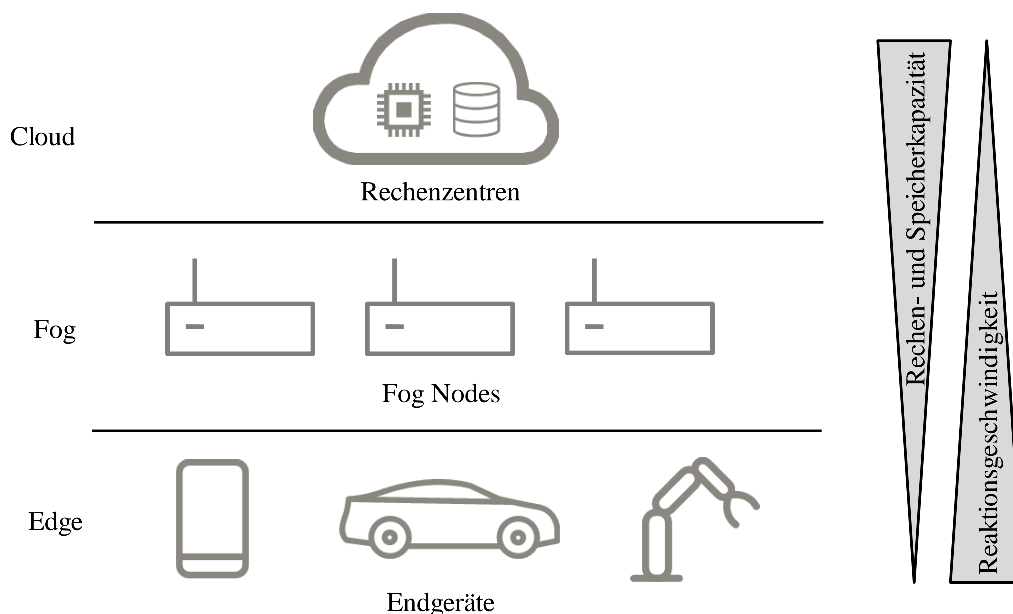


Abbildung 2-1: Cloud, Fog und Edge Computing
Quelle: In Anlehnung an Yi et al. (2015b)

Das IoT beschreibt einen Verbund von Endgeräten in einem Kommunikations- und Steuerungsnetzwerk, in dem Sensoren und Aktoren in der physischen Umgebung nahtlos integriert sind (Gubbi et al. 2013). Im Rahmen seiner Taxonomie spezifiziert Seitz (2019, S. 10) IoT als Oberklasse von Industrial Internet of Things (IIoT) und Cyber-Physical Systems (CPS). Das Konzept IIoT wird dort weiter in die Klassen Industrie 4.0 und Industrial Internet unterteilt. Im Gegensatz zu Cloud-Diensten, stellen die industriellen Anwendungen des IoT Echtzeitsysteme, mit der Anforderung einer garantierten Ausführungszeit, dar.

Cloudlets verfügen über genügend Rechenkapazität, sind am Internet angeschlossen und stellen angrenzenden mobilen Endgeräten Dienste zur Verfügung (Satyanarayanan et al. 2009). Im Rahmen einer solchen Architektur instanziieren mobile Nutzer die kundenspezifische

Implementierung eines Dienstes auf einer vertrauenswürdigen virtuellen Ressource in der Nähe zur Auslagerung von Verarbeitungsschritten. Aus dieser Interaktion gehen beispielsweise Anforderungen hinsichtlich einer schnellen Inbetriebnahme von Diensten, ohne das Vorhandensein einer Einschwingphase, hervor.

Das *Edge Computing* adressiert die Herausforderung der Latenz und des Datendurchsatzes während der mobilen Nutzung von Cloud-Diensten und impliziert eine Verlagerung von Workloads von der zentralen Infrastruktur auf mobile Endgeräte (Varshney/Simmhan 2017). Aufgrund der eingeschränkten Verfügbarkeit von Ressourcen auf den Endgeräten, wird im Rahmen des *Fog Computing* eine Zwischenschicht eingeführt. Fog Computing erweitert das Cloud Computing an der Peripherie des Internet, indem es eine geringe und vorhersagbare Netzwerklatenz unterstützt (Bonomi et al. 2014). Durch die Verlagerung von Workloads an die Peripherie können Echtzeitanwendungen unterstützt werden (Linthicum 2017). Fog Nodes können heterogen sein und in Form von Serverknoten, Netzwerk-Router oder mobilen Geräten vorliegen (Bonomi et al. 2014). Die Nodes und das umgebende Netzwerk können dabei für den parallelen Betrieb unterschiedlicher Anwendungen in Tenants aufgeteilt werden.

Im Gegensatz zum Cloud Computing müssen beim Deployment von Anwendungen im Fog und Edge Computing spezielle Anforderungen hinsichtlich Latenz und Datendurchsatz berücksichtigt werden (Brogi/Forti 2017). Wöbker et al. (2018) beschreiben eine Infrastruktur für die Verteilung und Verwaltung derartiger Anwendungen. Neben der Eignung der Knoten muss auch die Geschwindigkeit der Verteilung und der Inbetriebnahme der Anwendungen berücksichtigt werden.

Im Fokus der vorliegenden Arbeit stehen Unternehmensanwendungen, die auf zentralen Infrastrukturen, ohne Echtzeitanforderungen, betrieben werden. Anforderungen an die Performance der Inbetriebnahme von Diensten aus dem Kontext der Cloudlets werden im Rahmen dieser Arbeit nicht betrachtet.

Weitere theoretische Grundlagen dieser Arbeit werden in den folgenden Abschnitten beschrieben.

2.1 Softwarequalitätsmodelle

In Anlehnung an der Definition für Produktqualität nach Garvin (1984) definieren Kitchenham/Pfleeger (1996) den Begriff Softwarequalität anhand von fünf Sichten. Die transzendente Sicht beschreibt Qualität als eine erkennbare, jedoch nicht definierbare Eigenschaft. Aus Nutzersicht beschreibt Qualität die Zweckmäßigkeit im Kontext der Bearbeitung einer Aufgabe durch einen Benutzer. Softwarequalität repräsentiert aus Herstellersicht den Grad der Erfüllung einer Spezifikation während der Umsetzung und nach der Auslieferung vor dem Hintergrund der Vermeidung von Kosten für Nacharbeiten. Aus Produktsicht wird die Softwarequalität durch die Messung der internen Produkteigenschaften bewertet. Die Abwägung zwischen den Kosten für die Herstellung und dem Kundennutzen wird von der wertbasierten Sicht, in Form des Preises, den Kunden zu bezahlen bereit sind, adressiert.

Seit über vier Jahrzehnten stellen Qualitätsmodelle ein anerkanntes Mittel zur Unterstützung des Qualitätsmanagements von Software dar (Deissenboeck et al. 2009). Qualitätsmodelle

strukturieren Eigenschaften von Software in Taxonomien und beschreiben Metriken für deren Messung. Einige der wichtigsten Softwarequalitätsmodelle werden im Folgenden erläutert.

Softwarequalitätsmodell nach Boehm (1973)

Boehm (1973) stellt ein hierarchisches Modell für Softwarequalität vor. Das Modell adressiert die Fragestellungen, wie nützlich eine Software aktuell ist, wie einfach diese angepasst werden kann und ob diese nach einer Änderung der Umgebung noch nutzbar sein wird. Die Effizienz ist eines der Eigenschaften, welche den aktuellen Nutzen bestimmen und wird weiter untergliedert in *Messbarkeit* (engl. *accountability*), *Ressourceneffizienz* (engl. *device efficiency*) und *Ergonomie* (engl. *human engineering*). Die Messbarkeit beschreibt inwieweit Quelltext für die Bestimmung der Laufzeit instrumentiert werden kann. Ergonomie beschreibt die Effizienz der unterstützen Arbeitsabläufe aus Nutzersicht. Hinsichtlich der Ressourceneffizienz wird festgestellt, dass diese in Konkurrenz mit anderen Eigenschaften, wie Portabilität und Wartbarkeit, stehen kann (Boehm et al. 1976).

Softwarequalitätsmodell nach McCall et al. (1977)

McCall et al. (1977) entwickeln Richtlinien für eine objektive Spezifikation bzw. Quantifizierung des Qualitätsanspruchs an einem Softwaresystem während der Phase der Anforderungsdefinition. Konkret sollen Anforderung für den Entwickler formuliert und deren Einhaltung während der Entstehung des Softwaresystems geprüft werden. Mit ihrem Modell wollen McCall et al. (1977) die erste ganzheitliche Betrachtung von Softwarequalität ermöglichen.

Der übergreifende Begriff der Softwarequalität wird basierend auf den Ergebnissen einer Literaturstudie in Faktoren aufgeteilt. Faktoren stellen Eigenschaften, welche aktiv zur Qualität einer Software beitragen, dar. Für jeden Qualitätsfaktor wird eine Menge von Kriterien spezifiziert. Kriterien stellen Attribute von Software dar und helfen dabei Faktoren zu beurteilen. Im Gegensatz zu den Faktoren, welche die Sicht der Benutzer beschreiben, sind Kriterien an der Software orientiert. Ein einzelnes Kriterium kann dabei mehreren Faktoren zugeordnet werden. Dadurch können Beziehungen zwischen zwei Faktoren kenntlich gemacht werden. Für die Quantifizierung von Kriterien werden Metriken definiert. Metriken können sowohl objektiv als auch subjektiv sein. Diese werden als Verhältnis der beobachteten zu der maximalen Ausprägung dargestellt. Die Gesamtbewertung eines Faktors erfolgt über Normalisierung und Integration aller relevanten Metriken. Zusätzlich werden auch Empfehlungen für die Erhebung von Metriken beschrieben. Insgesamt wurden elf Faktoren definiert. Den Faktor Effizienz definieren die Autoren beispielsweise als den für eine Verarbeitung notwendigen Umfang an Quelltext und an Ressourcen. Anhand einer zeitlichen Einordnung soll der Faktor Effizienz während der Entwurfs- und der Implementierungsphase gemessen bzw. adressiert werden. Effizienz wird weiterhin in die Kriterien Verarbeitungs- und Speichereffizient unterteilt. Das Kriterium der Speichereffizienz beeinflusst den Faktor der Testbarkeit negativ. Auf den Faktor der Portabilität haben beide Kriterien für Effizienz einen negativen Einfluss. Die Effizienz wird wiederum von Kriterien, wie Zuverlässigkeit, Fehlertoleranz und Modularität, negativ beeinflusst.

FURPS+

FURPS+[1] ist ein von Hewlett-Packard eingesetztes Modell für die Festlegung von messbaren Zielen in der Produktentwicklung zur Erreichung einer höheren Kundenzufriedenheit und ist ein Akronym für *Functionality*, *Usability*, *Reliability*, *Performance* und *Supportability* (Grady 1992, S. 32). Eine Spezifizierung dieser Produkteigenschaften ist in Abbildung 2-2 dargestellt.

FURPS+				
Functionality	Usability	Reliability	Performance	Supportability
Feature Set Capabilities Generality Security	Human Factors Aesthetics Consistency Documentation	Frequency/ Severity of Failure Recoverability Predictability Accuracy Mean Time to Failure	Speed Efficiency Resource Consumption Thruput Response Time	Testability Extensibility Adaptability Maintainability Compatibility Configurability Serviceability Installability Localizability

Abbildung 2-2: Produkteigenschaften des FURPS-Qualitätsmodells
Quelle: Grady (1992, S. 32)

Eingebettet in einem übergeordneten Framework für den operativen Einsatz von Softwaremetriken von der Projektplanung bis zur Produktfreigabe, adressiert FURPS+ den Aspekt der Schätzung von Projekten und die Überwachung des Fortschritts (Grady 1992, S. 3). FURPS+ wird von den Autoren als Metriken für die Erfassung der Produkteigenschaften, welche zur Kundenzufriedenheit beitragen, beschrieben. Der Fokus des Modells liegt jedoch eher auf die ganzheitliche Betrachtung der Produkteigenschaften, als auf die detaillierte Spezifikation der einzelnen Metriken. Dem gegenüber stehen die Indikatoren für Kundenzufriedenheit, welche mit der Methode des Quality Function Deployment (QFD) quantifiziert werden. Mit Hilfe der QFD-Methode soll das Feedback der Kunden erfasst werden. Konkret werden im Rahmen dieser Methode Kundenanforderungen und Produkteigenschaften gegenübergestellt und Abhängigkeiten zwischen den beiden Dimensionen spezifiziert. Die beiden Ansätze werden dahingehend kombiniert, dass Kundenanforderungen nach den Produkteigenschaften des FURPS-Modells eingeordnet werden, damit alle relevanten Aspekte abgedeckt werden.

[1] Das anfangs als noch FURPS bezeichnete Modell wurde um die Möglichkeit der Erweiterung um zusätzliche Eigenschaften Rechnung zu tragen in FURPS+ umbenannt.

Softwarequalitätsmodell nach Dromey (1995)

Dromey (1995) entwickeln ein Modell zur Unterstützung der Einbettung von Qualität in Software, der Spezifikation von Qualitätsstandards für bestimmte Programmiersprachen, der Klassifizierung von Fehler und der Entwicklung von automatisierten Quelltextanalysen. Im Gengensatz zu den bis dahin bekannten Ansätzen versucht das Modell die Beziehung zwischen Softwareeigenschaften und Qualitätsattributen explizit darzustellen.

Bestandteile des Modells sind Strukturformen, Softwareeigenschaften und Qualitätsattribute. Strukturformen repräsentieren Bausteine einer Programmiersprache, wie z.B. Programme, Statements, Schleifen oder Variablen. Softwareeigenschaften beschreiben Strukturformen hinsichtlich Korrektheit, Struktur, Modularität und Aussagekraft. Unter Korrektheit werden Eigenschaften, wie Berechenbarkeit, Präzision und Konsistenz verstanden. Die Struktur beschreibt die Vollständigkeit und Konsistenz des Kontrollflusses von Programmen. Unter Modularität werden Eigenschaften, wie Kohäsion und Kopplung subsumiert. Das Vorhandensein von Spezifikationen und Dokumentation trägt zur Aussagekraft der Software bei. Die Qualitätsattribute Funktionalität, Zuverlässigkeit, Benutzerfreundlichkeit, Effizienz, Wartbarkeit, Portabilität und Wiederverwendbarkeit werden den Softwareeigenschaften zugeordnet. Die Effizienz wird beispielsweise von den folgenden Softwareeigenschaften beeinflusst:

- **Resolved:** Der Kontrollfluss passt mit den vorliegenden Datenstrukturen zusammen.
- **Effective:** Alle notwendigen, jedoch keine überflüssigen Verarbeitungsschritte für die Implementierung der Strukturform liegen vor.
- **Nonredundant:** Ähnlich zu der vorherigen Eigenschaft, liegen alle notwendigen, jedoch keine überflüssigen logischen Prüfungen vor.
- **Direct:** Die Art und Ausprägung der eingesetzten Strukturformen passen zu den Anforderungen.
- **Utilized:** Alle eingesetzten Strukturformen werden aufgerufen.

Normenreihe ISO/IEC 25000

Die Normenreihe ISO/IEC 25000[2] stellt die Überarbeitung des Standards ISO 9126[3] dar. Die Serie des neuen Standards mit dem Namen *Systems and Software Quality Requirements and Evaluation* (SQuaRE) besteht aus den folgenden Abschnitten (ISO/IEC 2011):

- **ISO/IEC 2500n - Quality Management Division:** Standards dieses Abschnittes definieren grundlegende Modelle und Begriffe, welche innerhalb der Normenreihe referenziert werden.
- **ISO/IEC 2501n - Quality Model Division:** Standards dieses Abschnittes beschreiben Qualitätsmodelle für Computersysteme, Softwareprodukte, Nutzqualität (engl. *quality in use*) und Daten.

---

[2] https://www.iso.org/standard/64764.html
[3] https://www.iso.org/standard/22749.html

- **ISO/IEC 2502n - Quality Measurement Division:** Standards dieses Abschnittes beschreiben ein Referenzmodell für die Messung der Qualität von Softwareprodukten und Definitionen für Qualitätsmetriken.
- **ISO/IEC 2503n - Quality Requirements Division:** Standards dieses Abschnittes sollen die Spezifikation von Qualitätsanforderungen anhand der Qualitätsmodelle und -metriken unterstützen.
- **ISO/IEC 2504n - Quality Evaluation Division:** Standards dieses Abschnittes beschreiben Anforderungen, Empfehlungen und Richtlinien für die Evaluation von Softwareprodukten.
- **ISO/IEC 25050 – 25099 SQuaRE Extension Division:** Standards dieses Abschnittes beinhalten Anforderungen an die Qualität kommerzieller Standardsoftware und Berichtsformate.

Software Product Quality							
Functional Suitability	Performance Efficiency	Compatibility	Usability	Reliability	Security	Maintainability	Portability
Functional Completeness Functional Correctness Functional Appropriateness	Time Behaviour Resource Utilization Capacity	Co-existence Interoperability	Appropriateness Recognizability Learnability Operability User Error Protection User Interface Aesthetics Accessibility	Maturity Availability Fault Tolerance Recoverability	Confidentiality Integrity Non-repudiation Accountability Authenticity	Modularity Reusability Analysability Modifiability Testability	Adaptability Installability Replaceability

Abbildung 2-3: Modell für Produktqualität nach ISO/IEC 25010
Quelle: ISO/IEC (2011)

Ein Modell für die Qualität von Computersystemen und Softwareprodukten wird durch den Standard ISO/IEC 25010 beschrieben. Dabei wird zwischen der Produktqualität - statische Eigenschaften von Software und dynamische Eigenschaften von Computersystemen - und der Nutzqualität - dem Ergebnis der Interaktion mit dem Produkt - unterschieden (ISO/IEC 2011). Das Modell für Produktqualität ist in Abbildung 2-3 dargestellt.

Die Eigenschaft Performance Efficiency beschreibt die Performanz im Verhältnis zu den genutzten Ressourcen und wird von den folgenden Untereigenschaften näher spezifiziert (ISO/IEC 2011):

- **Time Behavior:** Grad, in dem die Antwort- und Verarbeitungszeiten sowie die Durchsatzraten eines Produkts oder Systems bei der Erfüllung seiner Funktionen bestimmten Anforderungen genügen.
- **Resource Utilization:** Grad, in dem die Art und der Umfang der zur Erfüllung bestimmter Funktionen genutzten Ressourcen den Anforderungen entsprechen.

- **Capacity:** Grad, in dem die Höchstgrenzen eines Produkt- oder Systemparameters bestimmten Anforderungen genügen.

Das Modell der Nutzqualität besteht aus den Eigenschaften Effectiveness, Efficiency, Satisfaction, Freedom from Risk und Context Coverage, welche Anforderungen aus der Sicht des Benutzers beschreiben.

Aktuelle Forschungsarbeiten entwickeln neue Qualitätsmodelle für spezielle Architekturmuster oder adressieren deren praktische Anwendbarkeit. Göb (2013) entwickelt ein Modell für Qualität von serviceorientierten Architekturen, welches Qualitätsanalysen auf Basis von Messungen unterstützt. Lochmann (2014) entwickelt ein konkretes Meta-Modell für Softwarequalität, welches die Struktur von Qualitätsmodellen abbildet und basierend auf einem Werkzeug im Rahmen von Beurteilungen praktisch angewendet werden kann.

Nach Kan (2002, S. 5) können sich Qualitätseigenschaften gegenseitig positiv oder negativ beeinflussen. McCall et al. (1977) beschreiben folgende Faktoren, welche die Effizienz einer Software negativ beeinflussen:

- **Integrität:** die zusätzlichen Programm- und Verarbeitungsschritte für die Kontrolle des Zugriffs auf die Software und entsprechende Daten tragen zu einer längeren Laufzeit und einem höheren Speicherbedarf bei.
- **Benutzerfreundlichkeit:** die zusätzlichen Programm- und Verarbeitungsschritte für die Vereinfachung der Bedienabläufe und der Verbesserung der Bildschirmausgaben tragen zu einer längeren Laufzeit und einem höheren Speicherbedarf bei.
- **Wartbarkeit:** Der Einsatz von Modularisierung und Kommentaren zur Vereinfachung von Wartungsarbeiten stellt einen Overhead dar.
- **Testbarkeit:** Der Einsatz von Instrumentierung zur Vereinfachung von Tests stellt einen Overhead dar.
- **Portabilität:** Für die Unterstützung von Portabilität wird maschinenunabhängiger Quelltext anstatt eines für eine bestimmte Hardware optimierten Codes eingesetzt.
- **Flexibilität:** Eine für die Unterstützung von Flexibilität vorausgesetzte Abstrahierung erhöht den Overhead.
- **Wiederverwendbarkeit:** Eine für die Unterstützung von Wiederverwendbarkeit vorausgesetzte Abstrahierung erhöht den Overhead.
- **Interoperabilität:** die zusätzlichen Verarbeitungsschritte für die Transformation von Protokollen tragen zu einer längeren Laufzeit bei.

Eine starke Ausprägung des Qualitätsfaktors Effizienz, beeinflusst nicht implizit andere Faktoren negativ (McCall et al. 1977). Doch auch verschiedene Ausprägungen derselben Qualitätseigenschaft können sich gegenseitig negativ beeinflussen. Beispielsweise können sich unterschiedliche Metriken der Effizienz bzw. Performance aufeinander auswirken.

Mögliche Konflikte zwischen den Metriken der Effizienz sind:

- **Reaktionszeit:** Die Reaktionszeit beschreibt die Dauer zwischen dem Absetzen einer Anfrage durch den Nutzer und dem Anfang deren Bearbeitung (Jain 1991). Für die Verbesserung dieser Metrik muss die Verarbeitung in die Nähe des Nutzers verlagert werden. Dies wiederum kann eine eingeschränkte Ressourcenkapazität und dadurch eine längere Verarbeitungszeit implizieren.
- **Antwortzeit:** Durch die Zwischenspeicherung von Daten können Verarbeitungsschritte vermieden und dadurch die Antwortzeit verbessert werden. Dies impliziert jedoch auch einen eine höhere Auslastung des Hauptspeichers.
- **Anzahl Round Trips:** Bei der Implementierung von mobilen Anwendungen wird zugunsten eines besseren Ansprechverhaltens die Anzahl entfernter Aufrufe über das Internet minimiert. Um dies zu erreichen, werden innerhalb eines Aufrufs mehr Daten als evtl. notwendig abgerufen. Dies führt implizit auch zu einer höheren Auslastung des Hauptspeichers.
- **Garbage Collection:** Der Garbage Collector ist für die Entfernung von nicht mehr benötigten Objekten aus dem Heap bzw. dem Hauptspeicher zuständig und wird beispielsweise in der Java-Laufzeitumgebung eingesetzt. Während einer Garbage Collection müssen laufende Prozesse kurzzeitig angehalten werden, was zu einer Verschlechterung deren Verarbeitungszeit führt. Da in einigen Domänen, wie beispielsweise dem Börsenhandel, keinerlei Ausreißer innerhalb der Verarbeitungszeiten hinnehmbar sind, können Collection-Läufe nur selten durchgeführt werden. Dies wiederum, impliziert einen höheren Ressourcenverbrauch. Für viele andere Domänen spielt der Einfluss der Garbage Collector jedoch eine viel geringe Rolle und wird akzeptiert.
- **Kosten/Performance-Verhältnis:** Diese Metrik stellt einen zu verarbeitenden Workload zu den dadurch verursachten Kosten in Relation (Jain 1991). Durch eine höhere Ressourcenauslastung kann die Verarbeitungszeit von Programmen beschleunigt werden. Im Kontext von Edge und Fog Computing impliziert dies jedoch einen höheren Energieverbrauch, mehr Hardwarekapazität und ggf. eine kürzere Batterielebensdauer.
- **Computation Offloading:** Zur Überwindung von Ressourcenengpässen können Verarbeitungsschritte von mobilen Endgeräten auf Fog Nodes ausgelagert werden (Yi et al. 2015a). Die Ausrichtung einer Applikationsumgebung an die Unterstützung eines flexiblen Computation Offloading kann einen Overhead, der sich auf die Performance auswirkt, implizieren.
- **Speicherauslastung:** Für eine Senkung der Speicherauslastung können Daten, sowohl im Hauptspeicher als auch auf der Festplatte, komprimiert vorgehalten werden. Die impliziert aber eine höhere CPU-Auslastung aufgrund der Ausführung der Komprimierungsalgorithmen.

Die Priorität von sich gegenseitig beeinflussenden Qualitätseigenschaften kann nicht allgemeingültig festgelegt werden und hängt von den Anforderungen des Kunden bzw. der Nutzer ab (Kan 2002, S. 5). Konkurrierende Performancemetriken müssen auch bei der Auswahl einer geeigneten Infrastruktur gegeneinander abgewogen werden. Cloud-Plattformen bieten

umfangreiche und stabile Rechenkapazität bei einer relativ hohen Netzwerklatenz. Diese Infrastruktur ist vor allem für aufwändige Berechnungen und den Zugriff auf großen Datenmengen geeignet. Im Gegensatz dazu unterstützen mobile Endgeräte und Fog Nodes eine geringe Netzwerklatenz und damit eine hohe Reaktionszeit. Seitz et al. (2018) nennen neben einer geringen Latenz auch die minimale Auslastung der Bandbreite als Voraussetzungen für den Einsatz von Fog Computing im IIoT und bechreiben erfolgreiche Anwendungsfälle. Aufgrund der geringen Rechen- und Speicherkapazität können hierbei jedoch nur begrenzte Workloads verarbeitet werden. Das Konzept des Seamless Computing adressiert die Unterstützung der Mobilität von Workloads zwischen Cloud und Edge durch die Anwendung von Virtualisierung und Container-Technologien (Mueller et al. 2017).

In den folgenden Abschnitten der vorliegenden Arbeit wird der Begriff Performance für die Beschreibung der Effizienz von Software verwendet und anhand der Metriken Antwortzeit, Durchsatz und Verbrauch operationalisiert. Der Begriff Antwortzeit wird als für die Verarbeitung von Programmen oder Diensten notwendige Dauer definiert. Unter Durchsatz wird der Umfang der innerhalb einer bestimmten Zeitspanne verarbeiteten Einheiten verstanden. Als Ressourcenverbrauch werden die für eine Verarbeitung von Programmen oder Diensten benötigten Hardwareressourcen definiert. Weiterhin werden Abhängigkeiten zwischen der Performance und anderen Qualitätseigenschaften nicht berücksichtigt.

2.2 Performancebewusstsein

Der Begriff *Performancebewusstsein* (engl. *performance awareness*) beschreibt allgemein die Verfügbarkeit von Einsichten über Performance und die Fähigkeit auf diese Beobachtungen zu reagieren (Tůma 2014).

2.2.1 Schwerpunkte von Performancebewusstsein

Ansätze für Performancebewusstsein können unterschiedliche Inhalte und Zielgruppen adressieren. Nach Tůma (2014) werden die folgenden vier Schwerpunkte von Performancebewusstsein unterschieden:

- **Bewusstsein über performancerelevante Mechanismen:** Komplexe Systeme können auf eine Vielzahl von Schichten, Komponenten und Technologien aufbauen. Diese können wiederum komplexe Mechanismen, die einen großen Einfluss auf die Performance haben, aufweisen. Beispiele hierfür sind Garbage Collection und das Kompilieren bei bestimmten Plattformen. Durch die Abstrahierung der Mechanismen für aufrufende Schichten sind Einsichten über deren Einfluss auf die Performance nicht implizit vorhanden. Performancebewusstsein zielt auf die effiziente erlernen dieser Mechanismen ab.
- **Bewusstsein über Erwartungen an die Performance:** Das Performanceverhalten eines Systems kann sich unter geänderten Bedingungen, wie z.B. bei einer neuen Parametrisierung, verschlechtern. Eine Verschlechterung kann sowohl auf eine bewusste Entwurfsentscheidung als auch auf eine unvorhergesehene Einwirkung zurückzuführen sein. Hinsichtlich der Evolution eines Systems ist es wichtig unterscheiden zu können ob eine Verschlechterung der Performance unter den neuen Bedingungen zu erwarten war, oder ob die Ursachen erst identifiziert werden müssen. Performancebewusstsein zielt auf die Abbildung der Erwartungen an das Verhalten des Systems bzw. einzelner Komponenten ab.

- **Performancebewusstsein der Entwickler:** Während der Softwareentwicklung existieren verschiedene Aktivitäten und Akteure, welche die Performance beeinflussen können. Mit Hilfe von Spezialisten oder durch automatisierte Optimierungen, wie. z.B. während dem Kompilieren von Quelltext, werden diese Aspekte schon adressiert. Darüber hinaus sollten jedoch auch die Softwareentwickler befähigt werden, diese Maßnahmen zu ergänzen. Performancebewusstsein zielt auf die Befähigung der Entwickler, die Performance der entwickelten Software wären der Implementierung wahrzunehmen und zu verbessern.
- **Performancebewusstsein der Anwendungen:** Adaptive Anwendungen sind in der Lage über die eigene Struktur und das eigene Verhalten zu Reflektieren und basierend darauf Anpassungen vorzunehmen. Performancebewusstsein zielt auf die Erweiterung der Fähigkeiten von selbstadaptiven Anwendungen, um das eigenen Performanceverhalten beurteilen und beeinflussen zu können, ab.

Auch eine Verknüpfung dieser Schwerpunkte ist möglich und sinnvoll. Beispielsweise könnte ein Bewusstsein über performancerelevante Mechanismen für Entwickler unterstützt werden. Der Fokus dieser Arbeit liegt auf Performancebewusstsein für Entwickler. Die Ebenen der Mechanismen, Erwartungen und Anwendungen werden nicht isoliert adressiert.

2.2.2 Abgrenzung Performancebewusstsein und Entwicklerrückkopplung

In der Literatur werden Konzepte, die Aspekte von Performancebewusstsein abdecken, auch unter dem Begriff *Entwicklerrückkopplung* (engl. d*eveloper feedback*) eingeordnet (Heger et al. 2013; Weiss et al. 2013). Im Gegensatz zum Performancebewusstsein adressiert Entwicklerrückkopplung nicht die Ebenen der Mechanismen, Erwartungen und Anwendungen. Neben der Performance adressiert Entwicklerrückkopplung zusätzlich andere nichtfunktionale Anforderungen, wie. z.B. Security, sowie auch funktionale Anforderungen (Whalen et al. 2010). Die Schnittmenge zwischen Performancebewusstsein und Entwicklerrückkopplung ist in Abbildung 2-4 dargestellt.

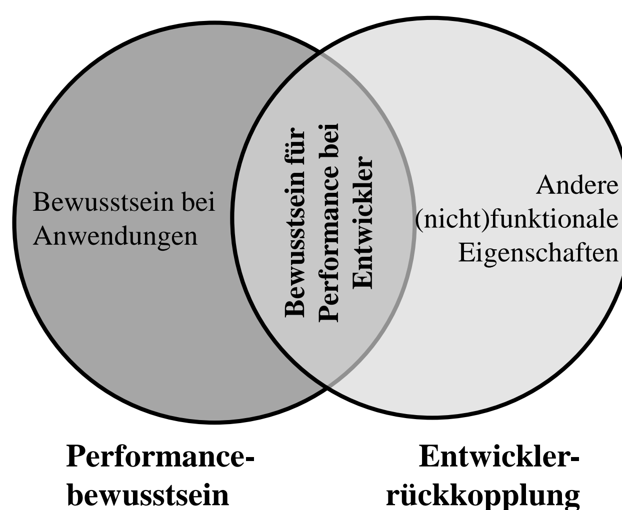


Abbildung 2-4: ***Schnittmenge zwischen Performancebewusstsein und Entwicklerrückkopplung***
Quelle: Eigene Darstellung

Der Ansatz des *Feedback-Driven Development* schlägt einen Softwareentwicklungsprozess für Cloud-Anwendungen vor, bei dem die Entwicklerrückkopplung eine zentrale Rolle darstellt (Bruneo et al. 2014). Hierbei werden Entwickler kontinuierlich mit Echtzeitmessungen aus der laufen Anwendung versorgt. Aufgrund der Einsichten über das Verhalten der Nutzer und der Anwendung soll es Entwicklern möglich sein, Verbesserungsmaßnahmen einzuleiten. Im Folgenden werden die Bereiche des Performancebewusstseins für Entwickler und die Entwicklerrückkopplung gleichgesetzt.

2.2.3 Kriterien für Performancebewusstsein bei Entwicklern

Zur Gewinnung von Einsichten über die Performance einer Anwendung können Entwickler mit Hilfe unterschiedlicher Techniken, wie z.B. Tracing oder Profiling, Messungen durchführen und anschließend analysieren. Dieses manuelle Vorgehen ist zeitaufwändig und setzt Werkzeuge und Spezialwissen voraus. Genau aus dieser Hinsicht grenzt sich Performancebewusstsein von Monitoring, Profiling und Instrumentierung ab. Westermann (2012) definiert sieben Kriterien bzw. Ziele für die Vereinfachung der Performanceevaluation während der Softwareentwicklung und für die Unterstützung von Entwickler mit Performancebewusstsein:

- **Einfachheit:** Die Untersuchung von Performanceeigenschaften sollte mit einem minimalen Aufwand und ohne spezielle Ausbildung möglich sein.
- **Zielorientierung:** Entwickler sollten nicht gezwungen werden umfangreiche Ergebnismengen zu analysieren. Stattdessen sollte die Rückmeldung eine oder zwei Metriken fokussieren.
- **Zuverlässigkeit:** Damit Evaluationsergebnisse von Entwickler akzeptiert werden, müssen diese zuverlässig und klar sein.
- **Anpassbarkeit:** Zur Adressierung der Mehrheit der Entwickler einer Organisation, sollten Ansätze für Performancebewusstsein unterschiedliche Technologien, Werkzeuge und Abstraktionsebenen bzw. Systemschichten unterstützen.
- **Genauigkeit:** Vorhersagen sollen präzise Ergebnisse liefern. Alle für den vorhergesagten Wert relevanten Einflussgrößen sollen dafür berücksichtigt werden.
- **Verallgemeinerbarkeit:** Zur Unterstützung mehrerer Szenarien, sollen Vorhersagen keine domänenspezifischen Annahmen über die Struktur der Beziehungen zwischen Einflussgrößen treffen.
- **Effizienz:** Durch einen möglichst hohen Automatisierungsgrad sollen Vorhersagen mühelos angefordert werden können. Für eine schnelle Berechnung der Ergebnisse sollen möglichst wenige Messläufe durchgeführt werden.

2.2.4 Aspekte von Performancebewusstsein

Der Begriff Performancebewusstsein spezifiziert nicht die Art und Weise, wie Beobachtungen gewonnen und darauf reagiert wird. Für beide Phasen können entsprechende Aktivitäten manuell oder automatisiert durchgeführt werden. Unternehmen stellen unter anderem Einsichten über Mechanismen und Antwortzeitverhalten von Systemen beispielsweise in Form von Dokumentation oder Service Level Agreements (SLA) bereit und ermöglichen dadurch Performancebewusstsein über manuelle Aktivitäten. Andere Verfahren automatisieren den Prozess der

Erhebung, Verarbeitung und Darstellung von Performancemessungen vollständig (Horký et al. 2015). Hinsichtlich des zeitlichen Bezugs unterstützen Ansätze für Performancebewusstsein sowohl historische Messungen (Bulej et al. 2015) als auch Vorhersagen über das zukünftige Verhalten (Weiss et al. 2013). Für die Verarbeitung von Beobachtungen werden sowohl mess- als auch modellbasierte Verfahren eingesetzt (Brunnert et al. 2015b). Bei messbasierten Verfahren werden Performancekennzahlen während der Laufzeit einer Anwendung im produktiven Betrieb oder aus Testumgebungen erhoben (Bulej et al. 2012b). Modellbasierte Verfahren können auch dann eingesetzt werden, wenn noch keine lauffähige Anwendung existiert und unterstützen dadurch Vorhersagen über die Performance (Danciu et al. 2014).

Einsichten können in unterschiedlichen Formen bereitgestellt werden. Einerseits können Performancemetriken, wie beispielsweise Antwortzeit oder Speicherverbrauch, dargestellt werden (Horký et al. 2015). Andere Ansätze liefern auch Hinweise auf problematische Programme oder textuelle Anweisungen, wie Probleme gelöst werden können (Parsons 2007). Werkzeuge für die Unterstützung von Performancebewusstsein können in den Entwicklungsumgebungen integriert sein, oder jedoch separat installiert und betrieben werden. Ausgehend von den existierenden Ansätzen kann die Klassifizierung aus der Abbildung 2-5 abgeleitet werden. Ansätze, die mehrere Kombinationen dieser Ausprägungen aufweisen, können auch vorkommen.

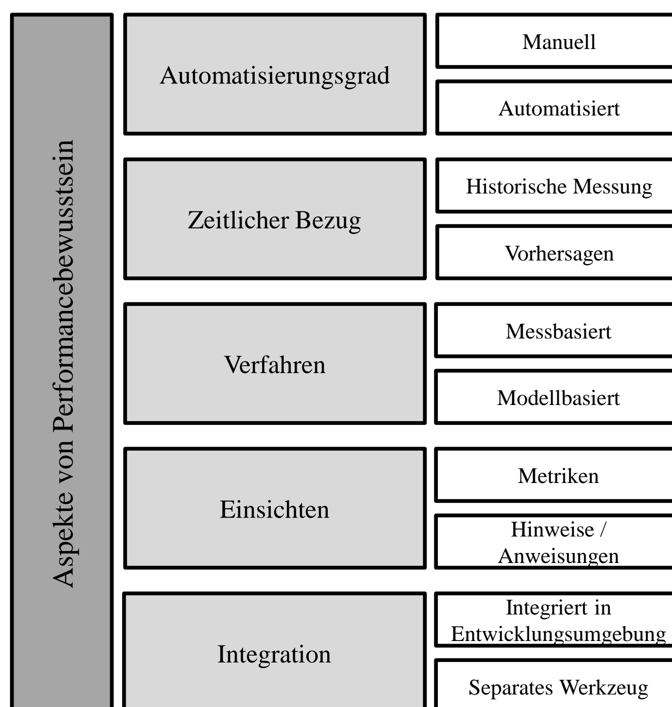


Abbildung 2-5: Aspekte von Performancebewusstsein
Quelle: Eigene Darstellung

2.2.5 Andere Arten von Bewusstsein während der Softwareentwicklung

Die Begriffe Bewusstsein bzw. Awareness werden auch in anderen Forschungsbereichen der Informatik gebraucht. In der rechnergestützten Gruppenarbeit steht oft das Bewusstsein über die Aktivität anderer Gruppenmitglieder im Fokus. Dourish/Bellotti (1992) untersuchen den Einfluss von Awareness auf die Effizienz der kollaborativen Textbearbeitung. Das Bewusstsein über die Änderungen an gemeinsam bearbeiteten Dokumenten wird in (Tam/Greenberg 2006) untersucht. Dourish/Bly (1992) Adressieren das Bewusstsein von räumlich entfernten Gruppenmitgliedern.

Storey et al. (2005) beschrieben die Ergebnisse einer Literaturstudie über Ansätze für Bewusstsein über die Aktivität anderer Gruppenmitglieder speziell während der Softwareentwicklung. Einige der identifizierten Arbeiten nutzen Informationen aus Versionskontrollsystemen, um beispielsweise die Evolution des Quelltextes zu visualisieren, oder um Änderungskonflikte zu vermeiden.

2.2.6 Continuous Software Engineering

Das Konzept des *Continuous Delivery* zielt darauf ab, eine Anwendung permanent in einem auslieferbarem Zustand vorzuhalten (Humble/Farley 2010, S. 347), und stellt eine Erweiterung des *Continuous Integration* und des *Continuous Deployment* dar. Unter Continuous Integration wird das häufige und daher auch stark automatisierte Testen von Software verstanden. Fehler sollen möglichst zeitnah identifiziert und den Entwicklern zurückgemeldet werden. Continuous Deployment adressiert hingegen den nachgelagerten Schritt der Verteilung von Software. Bei der kontinuierlichen Auslieferung (engl. delivery) geht es um die zeitnahe Erlangung und Rückmeldung von Kundenfeedback. Der Begriff des *Continuous Software Engineering* wurde von Bosch (2014) eingeführt. Krusche/Bruegge (2017) entwickelten ein Meta-Modell für die Beschreibung des kontinuierlichen Charakters des Software Engineering durch eine Abbildung von Prozessketten, welche von Ereignissen unterbrochen werden können.

Das Konzept des Continuous Software Engineering kann aus den Perspektiven der Frequenz und der Richtung von Maßnahmen betrachtet werden. Die Durchführung der Tests und der Verteilung soll nach jeder Änderung an der Software erfolgen können. Die daraus resultierenden Ergebnisse und Beobachtungen sollen zurück zum Entwickler, dem Ursprung der Änderung, zurückgemeldet werden. Die Verknüpfung des Continuous Software Engineering mit dem Konzept des Performancebewusstseins stellt eine sinnvolle Erweiterung dar und ergibt die kontinuierliche und zeitnahe Rückmeldung von Einsichten über die Performance einer Software zum Entwickler.

2.3 Komponentenbasierte Softwareentwicklung

Der Begriff der *Softwarekomponente* wurde erstmalig durch (McIlroy et al. 1968) geprägt. Durch die Spezifikation von lose gekoppelten Komponenten soll die Trennung von Aufgaben unterstützen und die Wiederverwendung fördern. Die vorliegende Arbeit orientiert sich an der Definition für Softwarekomponenten nach Szyperski (2002). Dieser beschreibt folgende Eigenschaften von Softwarekomponenten (Szyperski 2002, S. 41):

- Softwarekomponenten stellen Einheiten der Zusammensetzung dar,
- weisen vertraglich spezifizierte Schnittstellen auf,
- weisen nur explizite Abhängigkeiten zu einem Kontext auf,
- können unabhängig deployed werden und
- werden durch Dritte zusammengesetzt.

Schnittstellen beschreiben einerseits wie bestimmte Dienste durch Nutzer aufgerufen werden können und andererseits was durch eine Komponente implementiert werden soll (Szyperski 2002, S. 53). Im Gegensatz zu dem Blackbox-Ansatz, legen Whitebox-Komponenten nicht nur ihre Schnittstellen offen, sondern auch ihre Implementierung (Szyperski 2002, S. 40).

Komponentenmodelle liefern eine Definition für Komponenten und beschreiben wie diese erstellt, abgebildet, zusammengesetzt und deployed werden (Lau 2006). In diesem Sinne stellen Komponentenmodelle Meta-Modelle dar (Becker 2008, S. 23). Ein Beispiel für ein solches Meta-modell ist die Unified Modeling Language (UML) (OMG 2015).

2.3.1 Phasen der Entwicklung und Wiederverwendung von Komponenten

Im Rahmen der komponentenbasierten Softwareentwicklung sollen Komponenten aufgrund ihrer Unabhängigkeit wiederverwendet werden (Basili/Caldiera 1988). Sharp/Ryan (2010) beschreiben ein Framework für komponentenbasierte Softwareentwicklung und unterscheiden dabei zwischen der Entwicklung von Komponenten und von Systemen. Die Phasen dieses Frameworks sind in Tabelle 2-1 aufgelistet.

Während der Entwicklung von Komponenten werden bestimmte Problemstellungen und Anwendungsdomänen identifiziert und adressiert. Während der Entwurfsphase wird die Granularität von Komponenten spezifiziert. Die Architektur soll eine Abkapselung der Funktionen und die Wiederverwendbarkeit gewährleisten. Während der Auswahlphase werden von den Komponenten angebotene Dienste spezifiziert. Darauf aufbauend müssen externe Schnittstellen abgeleitet werden. Abschließend werden Komponenten implementiert.

Phase	**Aktivität**	
	Entwicklung von Komponenten	**Entwicklung von Systemen**
Analyse	Problemidentifikation und Auswahl der Domäne	Anforderungsanalyse
Entwurf	Erstellung der Komponentenarchitektur	Auswahl eines Komponentenmodells
Auswahl	Spezifikation der Dienste und externen Schnittstellen	Auswahl von wiederverwendbaren Komponenten
Implementierung	Erstellung der Komponente	Zusammenstellung der Komponenten

Tabelle 2-1: Phasen der Entwicklung von Komponenten und Systemen
Quelle: In Anlehnung an Sharp/Ryan (2010)

Während der Entwicklung von Systemen werden Komponenten wiederverwendet. Nach der Anforderungsanalyse soll ein geeignetes Komponentenmodell ausgewählt werden. Als Beispiel hierfür nennen die Autoren Enterprise Java Beans der Java Enterprise Edition (vgl. Abschnitt 2.5). Ausgehend von den spezifizierten Anforderungen sollen anschließend wiederverwendbare Komponenten identifiziert werden. Die Implementierung des Systems besteht aus der Zusammensetzung (engl. assembly) der ausgewählten Komponenten.

2.3.2 Einflussfaktoren auf die Performance einer Komponente

Die Performance einer Komponente im Sinne von Antwortzeit, Durchsatz und Ressourcenauslastung wird durch mehrere Faktoren unterschiedlich beeinflusst. Diese Einflussfaktoren sind in Abbildung 2-6 dargestellt.

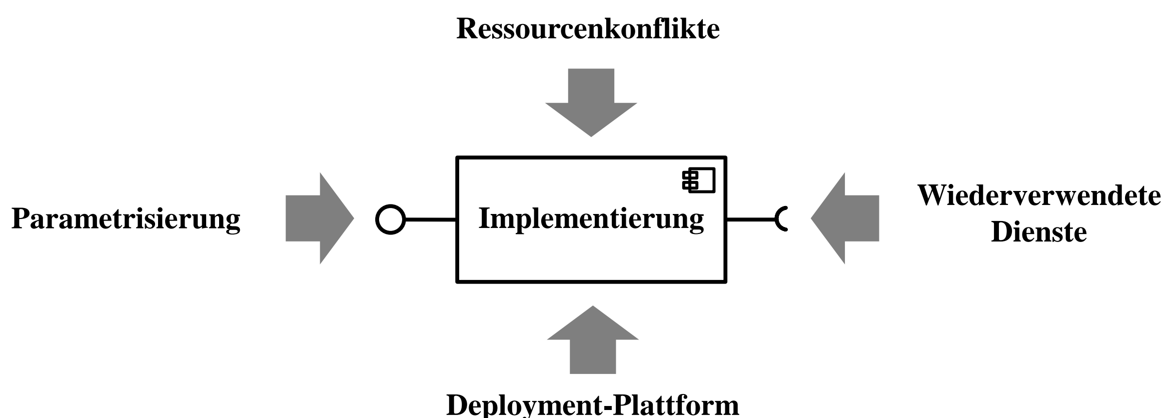


Abbildung 2-6: Einflussfaktoren auf die Performance einer Komponente
Quelle: In Anlehnung an Koziolek (2010)

Nach Koziolek (2010) wird die Performance einer Komponente durch folgende Faktoren beeinflusst:

- **Implementierung der Komponente:** Die Funktionalität einer Komponente kann auf verschiedene Weisen umgesetzt werden. Zwei Implementierungen desselben Dienstes können dabei unter identischen Bedingungen eine unterschiedliche Ausprägung der Antwortzeit, des Durchsatzes und der Ressourcenauslastung aufweisen.
- **Wiederverwendete Dienste:** Die Dauer des Wartens auf die Rückgabe eines wiederverwendeten Dienstes trägt zur Gesamtantwortzeit des aufrufenden Dienstes bei.
- **Parametrisierung:** Das Laufzeitverhalten wird von der Parametrisierung eines Dienstes durch den Aufrufer beeinflusst. Darüber hinaus kann der Kontrollfluss von den Rückgabewerten von wiederverwendeten Diensten abhängen. Die Parametrisierungen kann eine unterschiedliche Ausprägung der Antwortzeit, des Durchsatzes und der Ressourcenauslastung bewirken.
- **Deployment-Plattform:** Komponenten können auf unterschiedliche Plattformen deployed werden. Diese können aus mehreren Schichten, wie beispielsweise Container, Middleware, Datenbank, Betriebssystem und Hardware, bestehen und das Laufzeitverhalten der Komponenten beeinflussen.
- **Ressourcenkonflikte:** Durch die gemeinsame Nutzung der Ressourcen einer Plattform durch mehrere Prozesse können Konflikte und dadurch zusätzliche Wartezeiten entstehen. Diese Wartezeiten tragen zur Gesamtantwortzeit des betroffenen Dienstes bei.

Relevante Einflussfaktoren auf die Performance von Systemen im Allgemeinen stellen laut Jain (1991) Hardwareeigenschaften, wie beispielsweise der CPU-Typ, die Hauptspeichergröße, oder die Anzahl der verfügbaren Festplatten, dar.

2.3.3 Arten der Anbindung von wiederverwendeten Diensten

Die Anbindung eines wiederverwendeten Dienstes an die aufrufende Komponente kann sowohl innerhalb desselben Rechnerprozesses als auch über Interprozesskommunikation erfolgen (van Steen/Tanenbaum 2017, S. 106). Wird ein Dienst innerhalb desselben Prozesses, der die aufrufende Komponente ausführt, aufgerufen, stellt dies ein Library-Aufruf dar. Komponenten können jedoch auch Dienste wiederverwenden, die auf einem anderen Prozess oder einer anderen Maschine ausgeführt werden. Beispiele für dieses Muster sind Remote Procedure Calls und Message-Oriented Communication (van Steen/Tanenbaum 2017, S. 163). Darüber hinaus können Dienste synchron und asynchron aufgerufen werden. Im Gegensatz zu der asynchronen Kommunikation, blockiert die Ausführung der Komponente bei einem synchronen Aufruf bis zum Eintreffen der Rückmeldung von dem wiederverwendeten Dienst (van Steen/Tanenbaum 2017, S. 173). Dabei hat die synchrone Kommunikation mehr Einfluss auf die Performance der aufrufenden Komponente. Verteilte komponentenbasierte Systeme repräsentieren eine weit verbreitete Technologie für die Umsetzung von modernen Unternehmensanwendungen (Kounev 2006). Architekturmuster für die Realisierung solcher Systeme werden im folgenden Abschnitt beschrieben.

2.4 Serviceorientierte Architekturen

Serviceorientierte Architekturen (SOA) stellen ein Architekturmuster für die Implementierung von verteilten komponentenbasierten Systemen dar. SOA ist als Ansatz für die Bewältigung der Herausforderungen großer monolithischer Anwendungen hervorgegangen (Newman 2015, S. 8). Nach Fowler (2005) werden unter SOA sehr unterschiedliche Konzepte, wie die Bereitstellung von Software als Web Services, dem Aufbau einer standardisierten und unternehmensweiten Integrationsplattform oder dem Austausch von asynchronen Nachrichten zwischen Anwendungen, verstanden. Das W3C (2004) definiert SOA als eine Menge von Komponenten, welche aufgerufen werden können und deren Schnittstellenbeschreibungen veröffentlicht und entdeckt werden können. Eng verwandt mit SOA ist das *Microservice-Architekturmuster*, welches den Entwurf von Anwendungen als voneinander unabhängig einsetzbare Services beschreibt (Lewis/Fowler 2014). Richards (2015, S. 1) beschreibt SOA und Microservices als servicebasierte Architekturen bestehend aus verteilten Diensten, auf die mit Hilfe von Protokollen für entfernte Aufrufe zugegriffen wird. Eine Gegenüberstellung der beiden Architekturmuster ist in Abbildung 2-7 dargestellt.

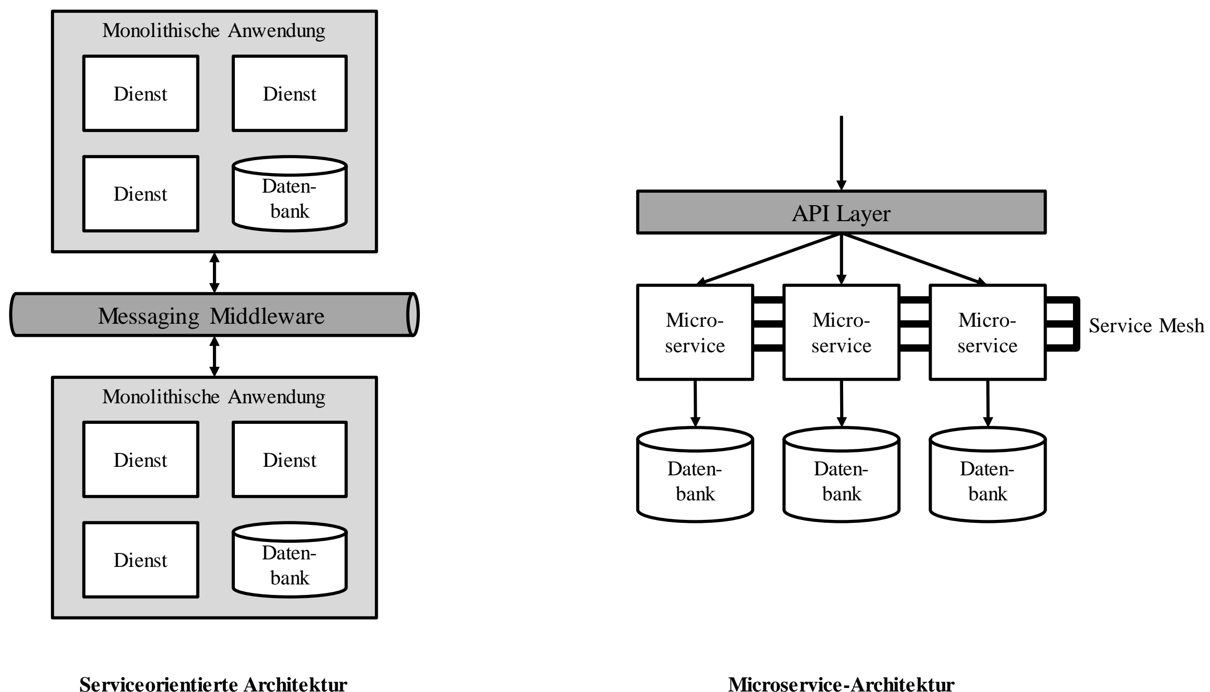


Abbildung 2-7: Serviceorientierte Architekturen
Quelle: In Anlehnung an Richards (2015, S. 26ff)

Bei einer klassischen serviceorientierten Architektur werden Dienste bzw. Services über eine Middleware miteinander integriert. Im Kontext von SOA unterstützen Services umfangreiche Funktionalitäten auf Unternehmensebene und werden von einer oder mehreren Komponenten implementiert. Mehrere zusammenhängende Services werden meistens als eine monolithische Anwendung zusammengefasst (Kounev 2006). Zur Bearbeitung mehrerer paralleler Aufrufe, können Services, abhängig von der Technologie, horizontal und vertikal skaliert werden. Bei der vertikalen Skalierung werden innerhalb einer Anwendungsinstanz mehrere Instanzen eines Service erstellt. Für die Hinzunahme zusätzlicher Hardwareressourcen, können neue Instanzen

der Anwendung bereitgestellt werden. Alle Services der Instanzen einer Anwendung nutzen dieselbe Datenbank für die Ablage von Informationen.

Monolithische Anwendungen verfolgen das Ziel einer hohen Integration und Wiederverwendung von Komponenten und Datenstrukturen. In der Praxis kann sich die Zusammenfassung einer hohen Anzahl von Komponenten innerhalb einer Anwendungsinstanz bzw. vieler Tabellen innerhalb derselben Datenbank als kontraproduktiv für den Betrieb und die Wartbarkeit des Gesamtsystems erweisen. Bei der Skalierung eines intensiv genutzten Service über mehrere Anwendungsinstanzen werden evtl. auch wenig genutzte Services mehrfach instanziiert. Auch einzelne Änderungen an einer Komponente oder einer Datenstruktur betreffen die Anwendung als Ganzes und bedingen umfangreiche Tests und Wartungsfenster für alle Anwendungsinstanzen.

Das Ziel des Microservice-Architekturmusters ist es die Herausforderungen der Test- und Wartbarkeit von monolithischen Anwendungen zu adressieren (Lewis/Fowler 2014). Der Umfang eines Microservice soll dafür so klein wie möglich gehalten werden. Während Services im Kontext von SOA zusammenhängende Funktionen bündeln und unternehmensweit zur Verfügung stellen sollen, implementieren Microservices spezialisierte Funktionen für bestimmte Anwendungsfälle. Durch die Vermeidung einer gemeinsamen Datenbasis können Microservices unabhängig voneinander entwickelt und deployed werden (Nadareishvili et al. 2016, S. 64). Durch das Entfallen eines gemeinsamen Applikationsservers ist ein Microservice für grundlegende Aspekte, wie beispielsweise Protokollierung und Authentifizierung, selbst zuständig. Neben deren Gestaltung, weicht auch die Integration der Services zwischen den beiden Architekturmustern stark voneinander ab. Nach Lewis/Fowler (2014) soll Anwendungslogik, die bei SOA von der Messaging Middleware umgesetzt wird, in die Microservice-Endpunkte verlagert werden. Einige der wichtigsten Techniken der Integration von verteilten Diensten werden im Folgenden beschrieben.

Messaging Middleware

Die Messaging Middleware übernimmt im Kontext einer SOA Aufgaben, wie die Ermittlung von Empfängern, die Erweiterung bzw. Transformation von Nachrichten oder die Übersetzung von Protokollen, und wird häufig als *Enterprise Service Bus* (ESB) instanziiert (Nadareishvili et al. 2016, S. 30). Nachrichten von angeschlossenen Unternehmensanwendungen werden von Service-Endpunkten des ESB über ein bestimmtes Protokoll entgegengenommen, in ein internes Format umgewandelt, ggf. anhand vordefinierter Regeln umgewandelt, und dann basierend auf dem Inhalt oder einem Prozessmodell an empfangende Anwendungen über ggf. abweichende Protokolle versendet (Chappell 2004). Die Implementierung komplexer Transformationsregeln führt zu einer Verlagerung von Geschäftslogik in dem ESB und erschwert somit die Wartbarkeit und das Testen von Anwendungen. Quelltextänderungen müssen dann sowohl auf der Ebene des Service als auch auf der Ebene des ESB getestet werden. Bei der Analyse von Fehlern im produktiven Einsatz von Anwendungen, müssen auch die Transformationsregeln im ESB untersucht werden. Nach Anpassungen am ESB, wie z.B. die Implementierung von Patches oder die Durchführung von Upgrades, muss auch die Interoperabilität mit den abhängigen Anwendungen getestet werden.

API Layer

Endpunkte bzw. Application Programming Interfaces (API) von Microservices werden in der Praxis meistens mit Hilfe von *API Gateways* abgesichert (Nadareishvili et al. 2016, S. 97). Neben der Authentifizierung von Clients, der Analyse des Aufrufverhaltens zur Abwehr einer Überlastung und einer Protokollierung für Abrechnungszwecke unterstützten API Gateways auch Lastverteilung und die Abstrahierung der technischen Anbindung der Endpunkte. Der API Layer kann neben der Funktion eines Gateways auch Werkzeuge für den Entwurf, die Dokumentation und die Vermarktung von API bereitstellen. Die von den Konzepten des ESB und API Gateway unterstützten Integrationsmuster sind in der Tabelle 2-2 gegenübergestellt.

Obwohl die Verlagerung von Geschäftslogik in Form von komplexen Transformationsregeln zu einer schlechteren Wart- und Testbarkeit von Anwendungen führt, ist dieser Ansatz für einige Anwendungsfälle hilfreich. Beim Austausch von Nachrichten zwischen Unternehmen, wie beispielsweise dem Versenden von Rechnungen von einem Zulieferer an eine große Menge Kunden, müssen viele individuelle Anforderungen hinsichtlich der Struktur der Belege berücksichtigt werden. Die Abbildung solcher Anforderungen als ausgelagerte Transformationsregeln in einem ESB kann dabei die Komplexität der Implementierung des Service reduzieren.

Integration Pattern	Enterprise Service Bus	API Gateway
Value Mapping	Ja	Nein
Aggregator	Ja	Nein
Splitter	Ja	Nein
Routing	Ja	Nein
Protocol Adaption	Ja	Ja
Structure Mapping	Ja	Ja
API Analytics	Nein	Ja
API Security	Nein	Ja
API Traffic Management	Nein	Ja

Tabelle 2-2: ***Abgrenzung zwischen Enterprise Service Bus und API Gateway***
Quelle: SAP (2018, S. 41)

Service Mesh

Ähnlich zum Ansatz des API Gateway, der eine vertikale Integration von Microservices mit ggf. externen Aufrufern unterstützt, können Microservices mit Hilfe eines *Service Mesh* horizontal untereinander integriert werden. Implementierungen wie Istio[4] und Linkerd[5] adressieren neben Sicherheit, Lastverteilung, Hochverfügbarkeit und Protokollierung auch Aspekte der

[4] https://istio.io/
[5] https://linkerd.io/

Zuverlässigkeit. Durch die Bereitstellung eines Circuit Breaker, kann beispielsweise verhindert werden, dass sich der Ausfall eines Service auf dessen Aufrufer ausweitet (Montesi/Weber 2016).

Dependency Injection

Dependency Injection (DI) ist eine Technik zur Anbindung von wiederverwendeten Komponenten innerhalb monolithischer Anwendungen. Im Gegensatz zum expliziten Aufruf eines Konstruktors, wird bei DI die Abhängigkeit zwischen Komponenten als Konfiguration abgebildet. Neben dem direkten Aufruf von Konstruktoren stellen auch die Entwurfsmuster Factory und Service Locator Vorläufer von DI dar (Prasanna 2009, S. 20). Die verschiedenen Arten der Konfiguration von Abhängigkeiten sind in Abbildung 2-8 dargestellt.

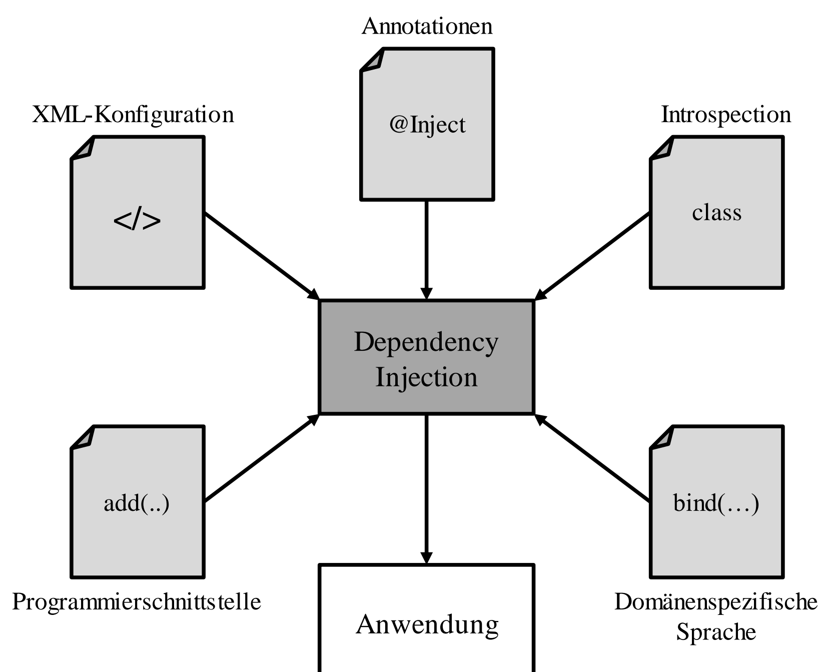


Abbildung 2-8: Arten der Konfiguration von Dependency Injection
Quelle: Prasanna (2009, S. 27)

Die Konfiguration mit Hilfe von Annotationen stellt eine aktuell weit verbreitete Art der Konfiguration von Abhängigkeiten. Dabei werden im Quelltext auf Ebene von Attributen, Methoden oder Parametern Referenzen auf wiederverwendete Komponenten mit Annotationen versehen. Bei der XML-Konfiguration werden die Spezifikationen der Abhängigkeiten in eine zentrale Datei ausgelagert. Bei der Konfiguration mit Hilfe einer domänenspezifischen Sprache oder einer Programmierschnittstelle werden Konfigurationen in Quelltextdateien ausgelagert (Prasanna 2009, S. 30ff). Unabhängig von der Art der Konfiguration können anzubindende Komponenten anhand des Typs oder anhand von Schlüsselwerten identifiziert werden (Prasanna 2009, S. 36ff). DI wird durch Frameworks für Programmiersprachen wie Java, C# und .Net unterstützt (Prasanna 2009, S. 19).

Auch die Auswahl eines Architkturmusters für den Entwurf und die Integration von Diensten beeinflusst Performanceeigenschaften unterschiedlich. Der Einsatz eines ESB mit

umfangreichen Transformationen kann die Verarbeitung von Nachrichten signifikant verzögern. Im Falle eines hohen Nachrichtenaufkommens kann die Messaging Middleware auch einen Bottleneck darstellen. Die Überlastung eines Dienstes durch bestimmte Clients kann mit Hilfe eines API Gateways identifiziert und verhindert werden. Beim Einsatz eines Service Mesh können Microservices direkt, ohne dabei eine zentrale Komponente zu durchlaufen, miteinander kommunizieren. Dadurch können geringere Raktions- und Antwortzeiten erzielt werden.

2.5 Java Enterprise Edition

Die Java EE spezifiziert eine Standardplattform für die Entwicklung und Ausführung von Unternehmensanwendungen auf Basis von Java (DeMichiel/Shannon 2013, S. 1) und repräsentiert einen etablierten Industriestandard für die Umsetzung von verteilten komponentenbasierten Systemen (Kounev 2006). Dabei wird Java EE eher zur Umsetzung von monolithischen Anwendungen im Kontext einer SOA eingesetzt. Eclipse MicroProfile[6] liefert eine Optimierung von Java EE für die Implementierung von Microservices. Die Java-EE-Plattform beschreibt grundlegende Dienste, wie beispielsweise Sicherheit, Kommunikation oder Persistenz von Anwendungen. Der Standard wird durch unterschiedliche Hersteller in Form von Anwendungsservern umgesetzt.

2.5.1 Architektur und Komponenten

Die Architekturelemente der Java-EE-Plattform sind in Abbildung 2-9 dargestellt. Container repräsentieren Laufzeitumgebungen und stellen Dienste für Anwendungskomponenten bereit (DeMichiel/Shannon 2013, S. 5). Der Web und der EJB Container bilden den Anwendungsserver (Müller-Hofmann et al. 2015, S. 5). Der Anwendungsserver nimmt Anfragen von Nutzern entgegen, bearbeitet diese und verschickt anschließend eine Rückmeldung. Die Java-EE-Laufzeitumgebung definiert folgende Komponententypen (DeMichiel/Shannon 2013, S. 8):

- *Application Clients* sind Java-Programme, welche auf Desktop-Rechnern laufen, typischerweise eine native Benutzeroberfläche anbieten und über Zugriff auf die Java-EE-Middleware verfügen.
- *Applets* sind Komponenten, welche typischerweise im Browser ausgeführt werden und eine umfangreiche Benutzerschnittstelle für Java-EE-Anwendungen implementieren.
- *Servlets, JSP-Seiten, JSF-Anwendungen, Filter* und *Web Event Listeners* werden typischerweise im *Web Container* ausgeführt und beantworten Anfragen über das Hypertext Transfer Protocol (HTTP). Mit Hilfe dieser Komponenten können beispielsweise Benutzeroberflächen in Form von Hypertext Markup Language (HTML) oder Rückgabewerte in Form von Extensible Markup Language (XML) erzeugt werden.
- *Enterprise JavaBeans (EJB)* implementieren die Geschäftslogik von Java-EE-Anwendungen und werden in einem *EJB Container* mit Unterstützung für Transaktionen ausgeführt.

[6] https://microprofile.io/

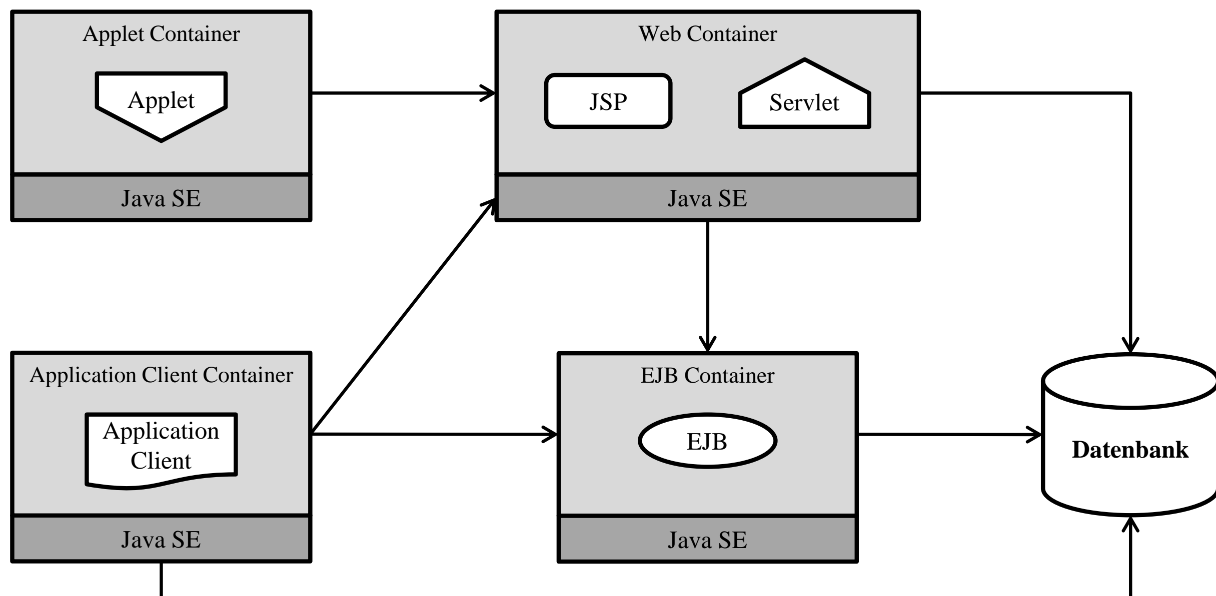


Abbildung 2-9: Architektur der Java EE
Quelle: In Anlehnung an DeMichiel/Shannon (2013, S. 6)

Die EJB-Technologie unterstützt die Implementierung von EJB-Komponenten bzw. -Instanzen durch die Annotation einfacher Java-Klassen. EJB-Instanzen verfügen entweder implizit, oder aufgrund der expliziten Deklaration, über Dienste wie beispielsweise Zustandsmanagement, Speichermanagement oder Sicherheit (Panda et al. 2014, S. 6). Es werden folgende Typen von EJB unterschieden (Saks 2009, S. 39f):

- *Session Beans* werden zur Bearbeitung von Anfragen eines einzelnen Nutzers ausgeführt, können Transaktionen unterstützen und Objekte in einer darunterliegenden Datenbank manipulieren. Nach dem Herunterfahren oder einem Absturz des EJB Container geht eine entsprechende Instanz verloren.
- *Message-driven Beans* werden nicht direkt von Nutzern, sondern von einzelnen Nachrichten ausgelöst und asynchron ausgeführt. Diese können Transaktionen unterstützen und Objekte in einer darunterliegenden Datenbank manipulieren. Message-driven Beans haben keinen Zustand und entsprechende Instanzen gehen nach dem Herunterfahren oder einem Absturz des EJB Container verloren. Entsprechende Komponenten werden mit Hilfe der Annotation @*MessageDriven* deklariert.
- *Entity Beans* sind Teil eines Domänenmodells und bieten eine objektbasierte Sicht auf die Inhalte der Datenbank. Diese existieren auch nach dem Herunterfahren oder einem Absturz des EJB Container. Entsprechende Komponenten werden mit Hilfe der Annotation @*Entity* deklariert.

Bei der Anmeldung am Anwendungsserver wird jedem Nutzer eine Session zugewiesen (DeMichiel/Shannon 2013, S. 42). Session Beans können weiter in folgende Kategorien unterteilt werden (Panda et al. 2014, S. 12):

- *Stateful Session Beans* speichern ihren Zustand zwischen zwei Aufrufen desselben Nutzers. Entsprechende Komponenten werden mit Hilfe der Annotation @*Stateful* deklariert.

- *Stateless Session Beans* speichern keinen Zustand und repräsentieren Dienste, welche durch eine Ausführung abgeschlossen werden. Entsprechende Komponenten werden mit Hilfe der Annotation @*Stateless* deklariert.
- *Singleton Session Beans* speichern ihren Zustand, werden jedoch von allen Nutzern geteilt und existieren über die Lebensdauer der Anwendung. Entsprechende Komponenten werden mit Hilfe der Annotation @*Singleton* deklariert.

2.5.2 Speicherung von Entity-Instanzen

Die Unterstützung der Persistenz und die objektrelationale Abbildung (engl. object/relational mapping) wird durch die *Java Persistence API (JPA)* spezifiziert (DeMichiel 2013, S. 21). JPA wird durch unterschiedliche Hersteller in Form von Persistenzframeworks umgesetzt. Wichtige Implementierungen der JPA-Spezifikation sind Hybernate[7], EclipseLink[8] und OpenJPA[9].

Mit dem *EntityManager* definiert JPA eine Schnittstelle für die Verwaltung von Entity Beans. Jede Instanz des EntityManager ist einem *Persistence Context* zugeordnet. Der Persistence Context stellt eine Menge von Entity-Instanzen, in der für eine Entity-Identität genau eine Instanz existiert, dar (DeMichiel 2013, S. 63). Der EntityManager spezifiziert Funktionen für die Interaktion mit dem Persistence Context bzw. für das Suchen, Speichern und Sperren von Entity-Instanzen sowie für die Erstellung von Datenbankabfragen.

Hinsichtlich der Verwaltung von Entity Beans spezifiziert JPA auch die Unterstützung von Mechanismen für die Pufferung (engl. caching) von Instanzen. Bauer/King (2006) unterscheiden folgende Stufen der Pufferung:

- Pufferung auf Transaktionsebene: Daten können innerhalb einer Verarbeitungseinheit bzw. einer Datenbanktransaktion zwischengespeichert werden. Dabei verfügt jede Verarbeitungseinheit über einen eigenen Puffer. Dieser wird nach der Ausführung der Verarbeitungseinheit aufgelöst.
- Pufferung auf Prozessebene: Mehrere, evtl. parallellaufende, Verarbeitungseinheiten greifen auf einem gemeinsamen Puffer zu.
- Pufferung auf Cluster-Ebene: Mehrere Verarbeitungseinheiten, die auf unterschiedlichen Maschinen laufen können, greifen auf einem gemeinsamen Puffer zu.

Die JPA-Spezifikation adressiert die Unterstützung eines Second-Level Cache für die Verbesserung der Performance von Anwendungen (DeMichiel 2013, S. 124). Die Pufferung auf Transaktionsebene entspricht einem First-Level Cache und die Pufferung auf Prozess- sowie Cluster-Ebene einem Second-Level Cache (Bauer/King 2006, S. 596). Der Persistence Context dient implizit als ein First-Level Cache.

Unterschiedliche EntityManager-Instanzen können dieselbe Entity-Identität in ihrem Persistence Context laden und gleichzeitig bearbeiten. Zur Sicherstellung der Konsistenz bei

[7] http://hibernate.org/orm/
[8] http://www.eclipse.org/eclipselink/
[9] http://openjpa.apache.org/

konkurrierenden Zugriffen auf Entity-Instanzen spezifiziert JPA auch einen Sperrmechanismus (DeMichiel 2013, S. 89).

2.5.3 Rollen und Lebenszyklus

Die Java-EE-Plattform unterscheidet im Zusammenhang mit der Entwicklung und dem Betrieb von Java-EE-Produkten und -Anwendungen und folgende Rollen (DeMichiel/Shannon 2013, S. 19ff):

- Java EE Product Provider sind Hersteller von Anwendungsservern und stellen Implementierungen der Java-EE-Plattform bereit.
- Application Component Provider sind beispielsweise Gestalter von HTML-Dokumenten oder Entwickler von EJB.
- Application Assembler stellen eine Menge von Java-EE-Komponenten zu Anwendungen zusammen.
- Deployer setzten Werkzeuge des Product Provider für die Einrichtung von Komponenten und Anwendungen in einer Laufzeitumgebung ein. Die Einrichtung erfolgt in den Phasen Installation, Konfiguration und Ausführung.

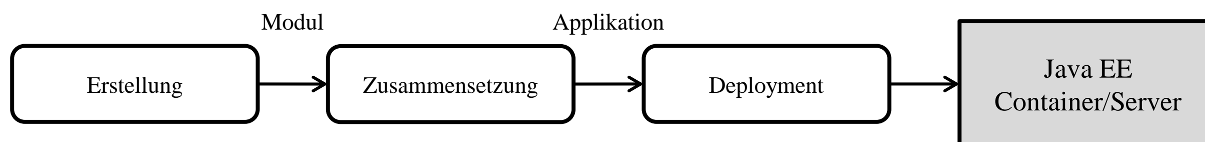


Abbildung 2-10: Lebenszyklus von Java-EE-Applikationen
Quelle: In Anlehnung an DeMichiel/Shannon (2013, S. 197)

Der Lebenszyklus von Java-EE-Applikationen ist in Abbildung 2-10 dargestellt. Einzelne Java-EE-Komponenten werden zuerst durch einen Provider erstellet. Mehrere Komponenten können ein Java-EE-Modul bilden. Der Assembler setzt mehrere Module zu einer Java-EE-Anwendung zusammen. Anwendungen werden durch den Deployer in einem Container deployed. Einzelne Java-EE-Module können jedoch auch direkt deployed werden. Eine Deployment Descriptor stellt eine Vereinbarung zwischen dem Provider bzw. Assembler und dem Deployer dar und beschreibt die Anforderungen der Komponenten an ihre Laufzeitumgebung (DeMichiel/Shannon 2013, S. 23).

2.6 Modellbasierte Performanceevaluation

Die Performance von Computersystemen kann durch den Einsatz von Messung, analytischer Modellierung und Simulation evaluiert werden (Jain 1991). Dabei basieren sowohl analytische Verfahren als auch die Simulation auf Performancemodellen. Nach Jain (1991) eignen sich modellbasierte Verfahren auch dann, wenn das zu untersuchende System noch nicht existiert.

Formalismen für die Performancemodellierung unterstützten die Vorhersage von Performance-eigenschaften. Nach Jonkers (1994) werden deterministische und probabilistische Formalismen unterschieden. Deterministische Modelle spezifizieren Quantitäten, wie Zeiten oder die Anzahl

von Iterationen, als feste Werte. Im Gegensatz dazu, nehmen diese Quantitäten bei probabilistischen Modellen mit einer Wahrscheinlichkeit einen bestimmten Wert an. Einige der wichtigsten Formalismen werden im Folgenden beschrieben.

2.6.1 Warteschlangennetze

Für die Evaluierung der Performance von Computersystemen werden oft Warteschlagennetze (Bolch et al. 2006) eingesetzt. Ein Beispiel für ein Warteschlangennetz bestehend aus zwei Warteschlangensystemen ist in Abbildung 2-11 dargestellt. Diese werden basierend auf der Arbeit von Bolch et al. (2006) im Folgenden beschrieben.

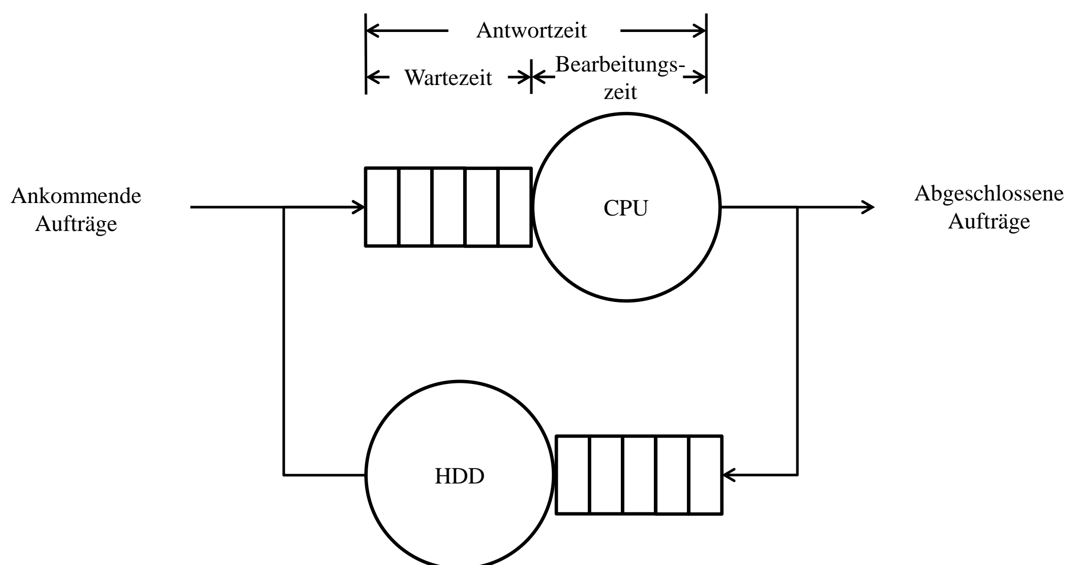


Abbildung 2-11: Beispiel für ein Warteschlangennetz
Quelle: In Anlehnung an Menascé et al. (2004, S. 39f)

Ein Warteschlangensystem besteht aus einer Warteschlange mit begrenzter oder unbegrenzter Kapazität und mehrere identischen Bediener (engl. service station), welche Ressourcen aus der realen Welt darstellen. Bediener können nur einen Auftrag (engl. job) zur selben Zeit abarbeiten. Bediener können dadurch den Status *besetzt* oder *inaktiv* aufweisen. Falls zu einem Zeitpunkt alle Bediener besetzt sind, werden ankommende Aufträge in die Warteschlange abgelegt. Nach der Abarbeitung eines Auftrags durch einen Bediener wird anhand einer vorgegebenen Scheduling-Strategie ein neuer Auftrag aus der Warteschlange für die Bearbeitung ausgewählt. Folgende Scheduling-Strategien werden unterschieden:

- First-Come-First-Served: Aufträge werden in der Reihenfolge bearbeitet, in der diese angekommen sind.
- Last-Come-First-Served: Aufträge, welche zuletzt angekommen sind, werden als nächstes bearbeitet.
- Service-In-Random-Order: Aufträge werden in zufälliger Reihenfolge für die Bearbeitung ausgewählt.

- Round Robin: Aufträge, die innerhalb eines bestimmten Zeitrahmens nicht bearbeitet werden konnten, werden zurück in die Warteschlange abgelegt. Aufträge werden dabei nach First-Come-First-Served für die Bearbeitung ausgewählt. Das Verfahren wird solange wiederholt, bis die Bearbeitung abgeschlossen ist.
- Processor Sharing: Entspricht einer Round-Robin-Strategie mit unendlich kleinem Zeitrahmen und stellt die gleichzeitige Bearbeitung aller Aufträge dar.
- Inifinite Server: Es werden so viele Bediener angenommen, dass keine Warteschlangen entstehen.
- Static Priorities: Die Auswahl wird anhand einer festen Priorität getroffen.
- Dynamic Priorities: Die Auswahl wird anhand einer sich mit der Zeit ändernder Priorität getroffen.
- Preemtion: Die Bearbeitung eines Auftrags wird unterbrochen, falls in der Warteschlange ein anderer Auftrag mit höherer Priorität existiert.

Mit der Ankunftsrate wird die Anzahl der Aufträge, welche in einer bestimmten Zeit ankommen, beschrieben. Wie in Abbildung 2-11 dargestellt, besteht die Antwortzeit eines Warteschlangensystems aus der Wartezeit und der Bearbeitungszeit.

Warteschlangennetze bestehen aus mindestens zwei miteinander verbundenen Bedienern. Aufträge können dabei beliebig zwischen den Bediener übertragen werden. Bei offenen Warteschlangennetzen können Aufträge im Netzwerk ankommen und dieses auch verlassen. Bei geschlossenen Warteschlangennetzen können Aufträge weder ins Netzwerk ankommen, noch dieses verlassen. Die Anzahl der Aufträge in einem geschlossenen Warteschlangennetz bleibt dadurch konstant. Das Beispiel aus Abbildung 2-11 zeigt ein offenes Warteschlangennetz und besteht aus zwei Bedienern.

2.6.2 Layered Queueing Networks

Layered Queueing Networks (LQN) stellen eine Weiterentwicklung von Queueing Networks dar und unterstützen das gegenseitige Aufrufen von Bedienern (Woodside et al. 1995). Ein Beispiel für ein LQN ist in Abbildung 2-12 dargestellt. Das Konzept der LQN wird basierend auf der Arbeit von Woodside (2013) im Folgenden beschrieben.

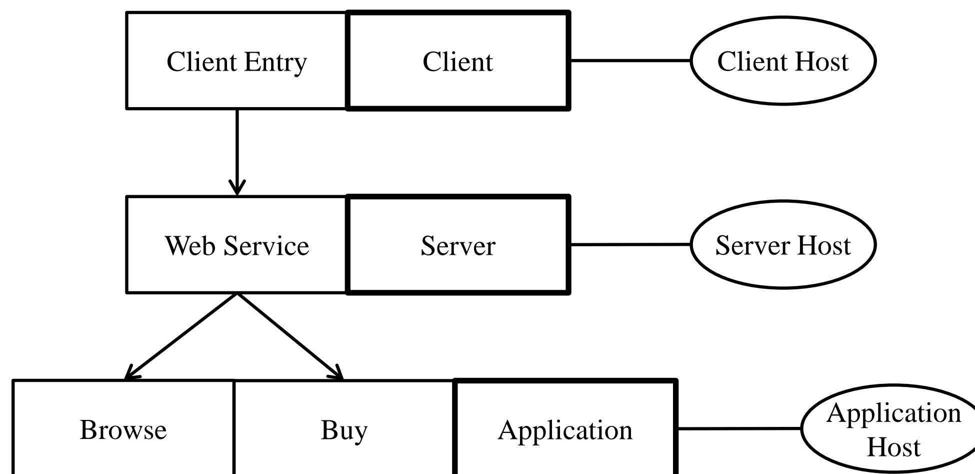


Abbildung 2-12: Beispiel für ein Layered Queueing Network
Quelle: In Anlehnung an Woodside (2013, S. 3)

Während der Bearbeitung eines Auftrags können Bediener andere Bediener auf einer darunterliegenden Schicht aufrufen. Zur Modellierung von LQN werden folgende Elemente unterschieden:

- *Tasks* repräsentieren logische Bediener und führen durch Entities spezifizierte Operationen aus. Tasks verfügen über Ressourcen, wie beispielsweise Warteschlangen, eine Scheduling-Strategie und eine Multiplizität. Die Multiplizität gibt die Anzahl der verfügbaren Bediener an. Tasks, welche nicht aufgerufen werden, dienen zur Generierung von Last.
- *Host Processors* sind Tasks zugeordnet und repräsentieren physische Ressourcen, mit deren Hilfe Operationen ausgeführt werden. Processors verfügen über eine Warteschlange und eine Scheduling-Strategie. Der Zugriff von Tasks auf Processor wird anhand einer Priorität gesteuert.
- *Entries* repräsentieren Operationen, die von Tasks angeboten werden. Tasks können über ein oder mehrere Entries verfügen. Workload-Parameter spezifizieren die Ressourcenauslastung, die Bedenkzeit und Aufrufe anderer Entries. Innerhalb desselben Tasks verfügen Entries weder über Prioritäten, noch können sie einander aufrufen.
- *Calls* stellen Aufrufe zwischen Entries unterschiedlicher Tasks dar. Calls können synchron, asynchron oder weitergeleitet sein. Bei weitergeleiteten Calls wartet der Aufrufer auf eine Rückmeldung, die Empfänger können Aufrufe jedoch asynchron weiterleiten.
- Demands beschreiben die Auslastung von Processors und die Anzahl der Aufrufe von Operationen, welche für die Bearbeitung eines Entries notwendig sind.

In dem Beispiel aus Abbildung 2-12 stellen die Elemente *Client*, *Server* und *Application* Tasks dar. Physische Ressourcen werden durch die Elemente *Client Host*, *Server Host* und *Application Host* abgebildet und sind jeweils einem Task zugeordnet. Tasks verfügen über Entries, die einander aufrufen.

2.7 Das Palladio Component Model

Das Palladio Component Model (PCM) beschreibt einen Ansatz für die Modellierung der Performance und die Evaluierung der Quality of Service (QoS) komponentenbasierter Software (Becker et al. 2009). PCM besteht aus einem Meta-Modell, einem Vorgehen für die Performancemodellierung sowie der Modellierungsumgebung Palladio-Bench[10]. Im Folgenden werden die Meta-Ebenen sowie die Teilmodelle des PCM beschrieben.

2.7.1 Meta-Ebenen des Palladio Component Model

Der Modellbegriff wird im Zusammenhang mit PCM auf unterschiedliche Abstraktionsebenen gebraucht. Ein Meta-Modell definiert die Struktur von Modellen und spezifiziert die Konstrukte einer Modellierungssprache sowie deren Beziehungen (Stahl et al. 2006, S. 85). Modelle bilden Elemente der realen Welt ab. Im Folgenden wird die Einordnung von PCM in den unterschiedlichen Meta-Ebenen beschrieben.

Der PCM-Ansatz ist mit Hilfe des Eclipse Modeling Framework[11] (EMF) implementiert. EMF ist ein Framework für modellgetriebene Softwareentwicklung basierend auf Eclipse[12] und unterstützt die Spezifikation von Meta-Modellen sowie die Generierung von Quelltext auf Basis dieser Meta-Modelle (Stahl et al. 2006, S. 230). EMF stellt eine Implementierung der Meta Object Facility (MOF) der Object Management Group (OMG) dar (Moore et al. 2004, S.4). Das PCM-Meta-Modell wird mit Hilfe von Ecore spezifiziert (Becker et al. 2009). Ecore wiederum repräsentiert das Meta-Modell des EMF. Komponenten der PalladioBench, wie z.B. graphische Editoren, werden auf Basis des PCM-Meta-Modells mit Hilfe von EMF automatisch generiert. Der Zusammenhang zwischen EMF, PCM und der modellierten Software ist in Abbildung 2-13 dargestellt.

[10] http://www.palladio-simulator.com
[11] https://eclipse.org/modeling/emf/
[12] https://eclipse.org

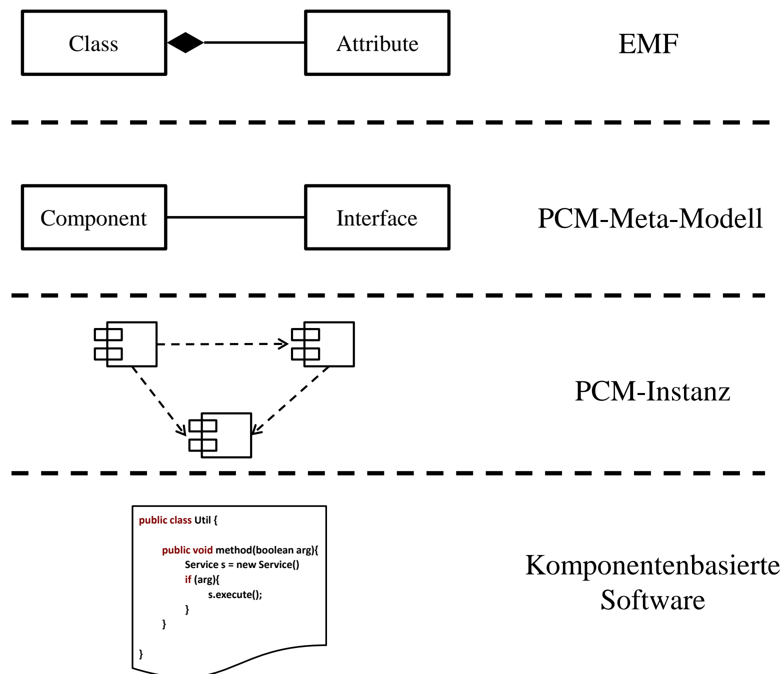


Abbildung 2-13: Meta-Ebenen des Palladio Component Model
Quelle: Eigene Darstellung

Unterschiedliche Aspekte komponentenbasierter Software, wie beispielsweise die Struktur und die Ressourcenauslastung zu Laufzeit, werden als PCM-Instanz modelliert. Das PCM-Meta-Modell definiert die Struktur einer PCM-Instanz und spezifiziert die Konstrukte der PCM-Modellierungssprache sowie deren Beziehungen untereinander. Mit Hilfe von EMF bzw. Ecore wird das PCM-Meta-Model spezifiziert.

2.7.2 Modellartefakte einer PCM-Instanz

Im Gegensatz zu den Warteschlangennetzen und LQN werden bei PCM Aspekte der Architektur separat von Hardwareaspekten modelliert. Der PCM-Ansatz unterschiedet mehrere Entwicklerrollen, die separate Modellartefakte erzeugen (Becker et al. 2009). Eine Zuordnung von Rollen zu Modellartefakte ist in Abbildung 2-14 dargestellt. Diese werden im Folgenden basierend auf der Arbeit von Becker (2008) beschrieben.

Modellartefakte einer PCM-Instanz werden durch folgende Entwicklerrollen erstellt bzw. verwendet:

- Der *Domänenexperte* modelliert das Verhalten von Benutzer.
- Der *Komponentenentwickler* modelliert Komponenten anhand ihrer Schnittstellen, Implementierung und Datentypen. Die Implementierung beschreibt die Parametrisierung und das Verhalten von Komponenten.
- Der *Systemarchitekt* nimmt die Komponenten des Komponentenentwicklers entgegen und setzt diese zu einem System zusammen.
- Der *System-Deployer* modelliert Ressourcen der Laufzeitumgebung und ordnet diese Komponenten zu.
- Der *QoS-Analyst* verwendet die Modellartefakte der PCM-Instanz, um Performancemetriken zu berechnen und um herauszufinden, ob die gewählte Softwarearchitektur die nichtfunktionalen Anforderungen erfüllt.

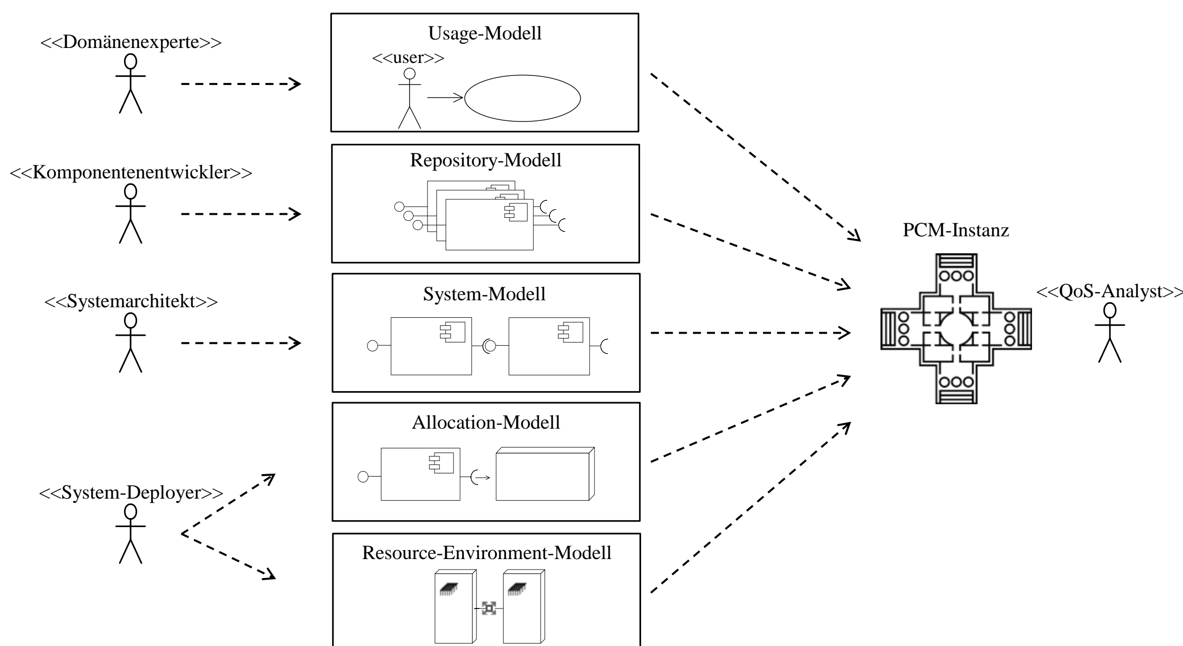


Abbildung 2-14: Modellartefakte einer PCM-Instanz
Quelle: In Anlehnung an Becker et al. (2009)

Wichtige Elemente der unterschiedlichen Modellartefakte werden im Folgenden beschrieben.

Usage-Modell

Die Spezifizierung des Nutzerverhaltens wird dem *UsageModel*-Wurzelknoten untergeordnet. Interaktionen mit dem System werden als *UsageScenario*-Elemente spezifiziert. Jedes UsageScenario verfügt über eine WorkloadSpecification, welche einen geschlossenen oder einen offenen Workload, eine Anzahl Nutzer sowie eine Bedenkzeit spezifiziert. Innerhalb eines UsageScenario führen Nutzer einzelne Schritte, welche als *UsageBehavior* modelliert werden, aus.

Repository-Modell

Spezifizierungen von Komponenten und Schnittstellen werden dem *Repository*-Wurzelknoten untergeordnet. Komponenten werden mit Hilfe des Elements *BasicComponent* abgebildet. Die Funktionalität mehrerer Komponenten kann als *CompositeComponent* zusammengefasst werden. Die Implementierung einzelner Dienste einer Komponente wird innerhalb einer Resource Demanding Service Effect Specification (RDSEFF) modelliert. RDSEFF werden als *ResourceDemandingSEFF*-Element dargestellt. Interne Verarbeitungsschritte werden als *InternalAction* abgebildet. Externe Aufrufe anderer Dienste werden als *ExternalCallAction* modelliert. Schnittstellen beschreiben von Komponenten angebotene bzw. benötigte Dienste und werden als *OperationInterface* spezifiziert. Parameter und Rückgabewerte von Schnittstellen spezifizieren ein *DataType*-Element.

System-Modell

Die Spezifizierung der Zusammensetzung eines Systems wird dem *System*-Wurzelknoten untergeordnet. Ein System besteht aus einer Menge *AssemblyContext*-Elementen, in denen Komponenten eingebettet werden. Innerhalb eines Systems werden Komponenten mit Hilfe von *AssemblyConnector*-Elementen miteinander verbunden. Systeme bieten über *ProvidedRole*-Elemente Dienste nach außen an. Mit Hilfe von *RequiredRole*-Elementen kann der Aufruf externer Dienste modelliert werden.

Allocation-Modell

Das Allocation-Modell verbindet ein System mit einer Laufzeitumgebung und beschreibt die Zuordnung einzelnen Komponenten zu Ressourcen. Hierfür wird für jeden AssemblyContext ein *AllocationContext*-Element angelegt. Alle Ressourcenarten, welche durch InternalAction-Elemente adressiert werden, müssen hier zugeordnet werden.

Resource-Environment-Modell

Die Spezifizierung der Laufzeitumgebung wird dem *ResourceEnvironment*-Wurzelknoten untergeordnet. Eine Umgebung besteht aus *ResourceContainer*- und *LinkingResource*-Elementen. Ein ResourceContainer modelliert eine physische Maschine, wie beispielsweise einen Server oder einen Client-Rechner. ResourceContainer bestehen aus *ProcessingResource*-Elementen, die Hardwarekomponenten darstellen. LinkingResource-Elemente spezifizieren Verbindungen zwischen ResourceContainer.

Modelle werden als separate Dateien abgespeichert. Das Repository-Modell stellt den Kern der PCM-Instanz dar und wird als erstes erstellt. Andere Modelle, wie z.B. das System- und das Usage-Modell referenzieren anschließend Komponenten des Repository-Modells. Im Rahmen einer Performancesimulation wird die PCM-Instanz zu Java-Quelltext, der die Struktur und das Verhalten des Modells abbildet, transformiert.

3 Verwandte Forschungsarbeiten

Die vorliegende Arbeit liefert Beiträge für unterschiedliche Forschungsbereiche und setzt gleichzeitig auf existierende Forschungsergebnisse auf. Ein Überblick über diese Forschungsbereiche ist in Abbildung 3-1 dargestellt.

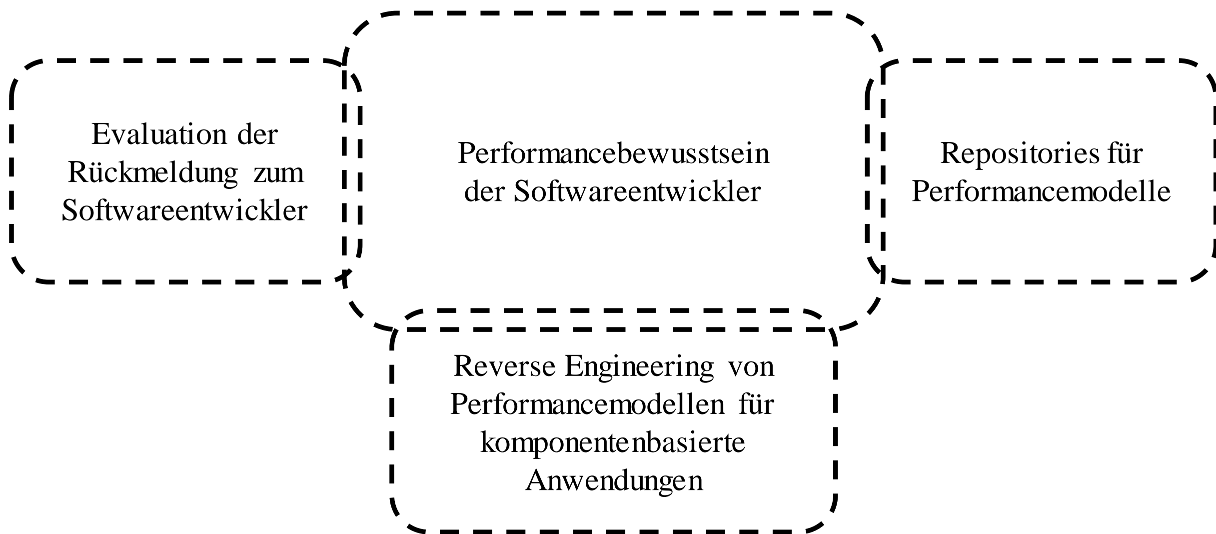


Abbildung 3-1: Teilbereiche verwandter Forschungsarbeiten
Quelle: Eigene Darstellung

Der Fokus der Arbeit liegt in der Entwicklung eines Ansatzes für Performancebewusstsein in Java-EE-Entwicklungsumgebungen. Durch die Anwendung von Reverse Engineering soll der Quelltext einer Java-EE-Anwendung als Performancemodell abgebildet werden. Auf dieser Grundlage sollen anschließend Antwortzeitvorhersagen durchgeführt werden. Der Einfluss des Ansatzes auf das Antwortzeiterhalten von Java-EE-Komponenten wird im Rahmen eines Experiments untersucht. Das im Rahmen dieser Arbeit entwickelte Modellrepository adressiert die Versionierung und Verwaltung von Performancemodellen und soll mit dem Ansatz für Performancebewusstsein integriert werden. In diesem Kapitel werden verwandte Forschungsarbeiten aus diesen Teilbereichen und die in dieser Arbeit adressierten Forschungslücken beschrieben.

3.1 Unterstützung von Performancebewusstsein der Entwickler

Das Konzept des Performancebewusstseins adressiert, wie in Kapitel 2.2 beschrieben, unterschiedliche Ebenen. Der Fokus dieser Arbeit liegt einzig auf der Ebene des Entwicklers. Dieser Abschnitt beschränkt sich daher auf die Beschreibung existierender Forschungsarbeiten mit einem ähnlichen Fokus. Diese werden im Folgenden in mess- und modellbasierte Ansätze unterteilt (siehe Abschnitt 3.1.1 und 3.1.2). Weiterhin werden die Ansätze anhand des Automatisierungsgrads, des zeitlichen Bezugs, des Softwaretyps und der unterstützen Metriken klassifiziert. Der Grad der Automatisierung beschreibt, ob Einsichten über die Performance von Anwendungen automatisiert bereitgestellt wird, oder ob das manuelle Eingreifen des Entwicklers notwendig ist. Der zeitliche Bezug beschreibt, ob sich die bereitgestellten Einsichten auf einen vergangenen, den aktuellen oder den zukünftigen Zustand einer Anwendung bezieht. Existierende

Ansätze adressieren unterschiedliche Arten von Software. Diese reichen von einfachen Java-Programmen bis hin zu Cloud-basierten Anwendungen. Performancemetriken können beispielsweise die Antwortzeit, den Ressourcenverbrauch oder den Durchsatz einer Software abbilden.

3.1.1 Messbasierte Ansätze für Performancebewusstsein

Existierende messbasierte Ansätze für die Unterstützung von Performancebewusstsein der Softwareentwickler sind in Tabelle 3-1 aufgelistet und werden im Folgenden beschrieben.

		Zeitlicher Bezug				
Ansatz	Automatisiert	Historische Messung	Vorhersagen	Softwaretyp	Einsichten	IDE-Integration
Miller et al. (1995)	X	X	-	Parallele Programme	Ressourcen-auslastung, Gleitkomma-operationen	-
Espinosa et al. (1998)	X	X	-	Parallele Programme	Änderungs-vorschläge	-
Vetter (2000)	(X)	X	-	Massage Passaging Interface	Klassifizierung	-
Parsons (2007)	X	X	-	Komponenten-basierte Anwendungen	Beschreibung Antipattern	-
Bulej et al. (2012b)	X	X	-	Java-Programme	Antwortzeit	X
Lee et al. (2012)	X	X	-	Datenbank-managmenet-systeme	Hinweis über Anomalie, Bericht über Ursachen	-
Yan et al. (2012)	X	X	-	Java-Programme	Potenzielle Ineffizienzen	-
Heger et al. (2013)	X	X	-	Java-Programme	Antwortzeit, Aufrufbaum	X
Nistor et al. (2013b)	X	X	-	Java-Programme	Hinweise auf Schleifen	-

Ansatz	Automatisiert	Zeitlicher Bezug: Historische Messung	Zeitlicher Bezug: Vorhersagen	Softwaretyp	Einsichten	IDE-Integration
Bureš et al. (2014)	X	X	-	Autonome Komponenten	Antwortzeit	X
Pradel et al. (2014)	X	X	-	Java-Klassen	Antwortzeit	-
Horký et al. (2015)	X	X	-	Java-Programme	Antwortzeit	X
Waller et al. (2015)	X	X	-	Java-Programme	Antwortzeit	-
Wert (2015)	X	X	-	Unternehmens-anwendungen	Performance-probleme	-

Tabelle 3-1: ***Messbasierte Ansätze für die Unterstützung von Performancebewusstsein der Softwareentwickler***
Quelle: Eigene Tabelle

Miller et al. (1995) beschreiben ein Werkzeug für die Messung der Performance von großen parallelen Programmen mit Hilfe einer dynamischen Instrumentierung. Das Werkzeug entscheidet während der Laufzeit des Programms selbständig den Zeitpunkt und den Umfang dieser Instrumentierung. Eine Assistenzfunktion unterstützt den Nutzer bei der Identifizierung von Performanceproblemen. Zur Identifizierung der Ursache eines Problems werden vordefinierte Hypothesen über bekannte Muster in parallelen Programmen evaluiert. Zur Lokalisierung des Problems identifiziert das Werkzeug Programmteile, welche mit der Problemursache im Zusammenhang stehen. Nutzer haben auch die Möglichkeit eine Analyse über den Zeitpunkt des Auftretens von Performanceproblemen abzurufen. Die Autoren beschreiben exemplarische Analyseergebnisse des Werkzeugs, jedoch keine Evaluierung mit Nutzer bzw. Entwickler. Dieses Werkzeug erfordert lauffähige Programme, die in repräsentative Hardwareumgebungen ausgeführt werden. Entwickler müssen das Werkzeug erlernen und separat betreiben.

Espinosa et al. (1998) beschreiben ein Werkzeug für die automatische Analyse der Performance von parallelen Programmen auf Basis von Ausführungsprotokollen und Quelltext. Eine Wissensbasis spezifiziert bekannte Muster für ineffiziente Programme, wie beispielsweise ungenutzte Prozessoren, in Form einer hierarchischen Struktur. In einem ersten Schritt wird ein Ausführungsprotokoll hinsichtlich der allgemeinen Performance des Programms analysiert. Falls die Ausführung zu lange gedauert hat, werden anhand der Wissensbasis einzelne Performanceprobleme identifiziert und priorisiert. Zu den wichtigsten Problemen werden Ursachen gesucht. Abschließend generiert das Werkzeug einen Bericht mit Lösungsvorschlägen für Entwickler. Eine Evaluierung des Werkzeugs wird nicht beschrieben. Das Werkzeug erfordert lauffähige Programme, die in repräsentative Hardwareumgebungen ausgeführt werden. Analysen erfordern die Durchführung von Tests und die Erhebung von Messungen durch den Entwickler.

Vetter (2000) beschreibt ein Werkzeug für die automatische Analyse der Effizienz von einzelnen Kommunikationen zwischen Message-Passing-Interface-Applikationen. Der Verlauf von Kommunikationen wird protokolliert und mit Hilfe von Entscheidungsbäumen klassifiziert. Durch den Einsatz von beispielhaften Implementierungen werden sowohl effiziente als auch ineffiziente Verläufe simuliert und für den Aufbau des Klassifikationsverfahrens verwendet. Ziel des Ansatzes ist es den Entwickler auf ineffiziente Implementierungen hinzuweisen. Der Autor beschreibt exemplarische Analyseergebnisse des Werkzeugs, jedoch keine Evaluierung mit Nutzer bzw. Entwickler. Damit bestimmte ineffiziente Muster erkannt werden können, müssen diese zuvor bekannt sein und beispielhaft implementiert werden.

Parsons (2007) beschreibt einen Ansatz für die automatische Identifikation von Performance-Antipatterns in Java-EE-Anwendungen. Mit Hilfe von Interceptors werden Aufrufe zwischen Komponenten instrumentiert und der Kontrollfluss der Anwendung erhoben. Das interne Verhalten von Komponenten wird dadurch vernachlässigt. Ausgehend von dem beobachteten Verhalten wird ein Modell der Anwendung abgeleitet. Durch den Einsatz von Data Mining werden bestimmte Muster in der Kommunikation zwischen Komponenten identifiziert. Eine Wissensbasis beschreibt vordefinierte Antipatterns. Als Ergebnis liefert der Ansatz eine Beschreibung der identifizierten Antipatterns und vordefinierte Lösungsvorschläge. Es werden nur Antipattern bzgl. des Entwurfs und des Deployments der Anwendung adressiert (Parsons 2007, S. 52), die Implementierung der Komponenten wird ignoriert. Der Ansatz kann nur Muster erkennen, die während der Evaluation aufgetreten sind, und setzt dadurch die Verfügbarkeit eines realistischen Testszenarios voraus (Parsons 2007, S. 155). Zusätzlich setzt der Ansatz das Deployment der gesamten Anwendung voraus und ist dadurch ist es erst ab der Testphase anwendbar. Eine Evaluation des Ansatzes hinsichtlich des Einflusses auf Softwareentwickler und deren Implementierungen wird nicht durchgeführt. Entwickler müssen die Nutzung des Werkzeugs erlernen und dieses separat betreiben.

Mit der Stochastic Performance Language (SPL) stellen Bulej et al. (2012b) einen Ansatz für die Spezifikation von Erwartungen an die Performance einzelner Methoden vor. Anstatt diese Erwartungen als absolute Zeitangaben zu formulieren, werden Erwartungen an einer Methode dabei relativ zu einer anderen Methode spezifiziert. Diese Spezifikationen sind dadurch unabhängig von der Plattform, auf der Messungen durchgeführt werden. Die SPL wird als Annotation für Java-Quelltext implementiert. Durch eine automatisierte Ausführung der annotierten Methoden können die spezifizierten Erwartungen evaluiert werden. Entwickler haben dadurch die Möglichkeit, darüber informiert zu werden, sobald diese Erwartungen nicht mehr eingehalten werden. Der Ansatz adressiert vor allem das Bewusstsein über Erwartungen an die Performance. Aufgrund der starken Integration mit dem Quelltext und der Entwicklungsumgebung wird implizit auch das Performancebewusstsein der Entwickler adressiert. Die SPL liefert keine Vorhersagen. Aufgrund der Abhängigkeiten zwischen Java-EE-Komponenten ist deren automatisierte Ausführung auch schwerer umzusetzen. Eine mögliche Anwendung der SPL wird in (Bulej et al. 2012a) beschrieben.

Lee et al. (2012) beschreiben einen Ansatz für die Erkennung von Anomalien in der Performance von Datenbankmanagementsystemen durch den Einsatz von Regressionstests und die anschließende Identifikation der Ursachen. Der Ansatz setzt eine lauffähige Implementierung voraus. Mit Hilfe einer Continuous-Build-Infrastruktur wird der Quelltext nach jeder Änderung zusammengebaut und ausgeführt. Anschließend werden Performancetests in Form von

vordefinierten Datenbankabfragen ausgeführt. Die Ergebnisse dieser Tests werden hinsichtlich der Existenz von Anomalien analysiert. Ggf. werden dann Hinweismeldungen verschickt. Für die Identifikation der Ursache einer Anomalie wird die betroffene Komponente mit einem Profiler untersucht. Die Ergebnisse werden mit der Analyse derselben Komponente, vor der Einführung der neuen Änderung, verglichen. Darauf basierend werden potenzielle Ursachen in Form von Quelltextänderungen in einem Bericht zusammengefasst. Vorhersagen werden nicht unterstützt. Die Aussagekraft der Performancemessungen hängt stark davon ab, ob die Testumgebung repräsentativ für die Produktionsumgebung ist.

Yan et al. (2012) beschreiben ein Profiling-Werkzeug für die Analyse der Verwaltung von Objektreferenzen in Java-Programmen. Die Erzeugung, Übergabe und die Nutzung von Objektreferenzen werden zur Laufzeit eines Programms überwacht. Die Ergebnisse werden als Graphstruktur abgebildet und analysiert. Abschließend wird eine priorisierte Liste von potenziell ineffizienten Quelltextsequenzen, welche Objekte erzeugen, erstellt. Nach Yan et al. (2012) kann somit die Hauptspeichernutzung und die Antwortzeit von Programmen reduziert werden. Der Ansatz wird anhand mehrerer Beispielanwendungen evaluiert. Hierbei wenden die Autoren selbst den Ansatz ein, ohne Versuchspersonen einzusetzen. Für die Implementierung des Ansatzes wurde eine spezielle JVM und der entsprechende Compiler erweitert, wodurch der Einsatz des Werkzeugs in der Praxis erschwert wird. Entwickler müssen die Nutzung des Werkzeugs erlernen und dieses separat betreiben.

Heger et al. (2013) beschreiben einen Ansatz für die automatische Erkennung von Performanceverschlechterungen und die Identifizierung der Ursachen. Dieser setzt die Verfügbarkeit von Unit-Tests für relevante Funktionen der Anwendung und den Einsatz eines Quelltextrepositories voraus. Der Ansatz ruft den Quelltext periodisch ab, instrumentiert die verfügbaren Unit-Test und führt diese anschließend mehrfach aus. Dabei werden Performancemessungen erhoben und abgespeichert. Durch den Vergleich mit vorherigen Ergebnissen können Verschlechterungen identifiziert werden. Im Falle einer Verschlechterung werden alle Methoden, welche von dem betroffenen Unit-Test aufgerufen werden, instrumentiert und erneut ausgeführt. Die dabei erhobenen Messungen werden in einem Aufrufbaum eingefügt und analysiert. Das Ergebnis der Analyse wird dem Entwickler innerhalb der IDE angezeigt. Anhand einer Java-Library wird evaluiert, ob Verschlechterungen erkannt werden. Der Einfluss des Ansatzes auf die Implementierungsergebnisse von Softwareentwicklern wird jedoch nicht evaluiert.

Nistor et al. (2013b) beschreiben ein Werkzeug für die automatisierte Erkennung von Performancebugs in Java-Programmen. Gesucht werden dabei Schleifen, die über mehrere Iterationen ähnliche Werte auslesen. In einem ersten Schritt werden sowohl das untersuchte Programm als auch die aufgerufenen Bibliotheken instrumentiert sodass die Ereignisse des Beginns und Beendens einer Schleife sowie des Auslesens von Werten protokolliert werden. Mit Hilfe von Testspezifikationen werden die Programme ausgeführt. Schleifen, die zur Laufzeit mehrfach durchlaufen werden und dabei einen bestimmten Anteil identischer Werte auslesen, werden als potenzielle Performancebugs zurückgemeldet. Entwickler müssen anschließend den betroffenen Quelltext untersuchen und interpretieren. Der Umfang der identifizierten Performancebugs hängt dabei stark von den Testspezifikationen ab. Mit Hilfe von Versuchspersonen wurden Spezifikationen implementiert und deren Einfluss auf die identifizierten Bugs untersucht. Der Einsatz des Werkzeugs durch Versuchspersonen wurde jedoch nicht untersucht. Entwickler müssen die Nutzung des Werkzeugs erlernen und dieses separat betreiben.

Bureš et al. (2014) beschreiben die Integration von Performancebewusstsein in dem Lebenszyklus von autonomen Komponenten. Während dem Entwurf werden Performanceziele spezifiziert. Diese können anschließend mit Hilfe der SPL formal dargestellt werden. Performancemessungen werden während der Laufzeit von Komponenten erhoben und aufbereitet. Diese werden dem Entwickler innerhalb der IDE als grafische Funktionen über Pop-Up-Fenster zurückgemeldet. Entwickler erhalten hierbei keine Hinweise auf ineffiziente Programmteile, sondern Messungen über die aktuell betrachteten Funktionen. Der Ansatz wird als Vision vorgestellt und nicht evaluiert.

Pradel et al. (2014) beschreiben ein Ansatz für die Erkennung von Performanceregressionen bei thread-sicheren Klassen. Der Ansatz erstellt für zwei Versionen derselben thread-sicheren Klasse parallellaufende Performancetests automatisch und führt diese aus. Dabei wird sichergestellt, dass diese hinreichend oft wiederholt werden. Der Ansatz wird für Java-Programme umgesetzt und anhand mehrerer Beispiele aus verschiedenen Java-Projekten evaluiert. Die Autoren stellen fest, dass die generierten Tests evtl. wenig realistische Nutzungsszenarien abbilden. Entwickler müssen die Nutzung des Werkzeugs erlernen und dieses separat betreiben.

Horký et al. (2015) beschreiben einen Ansatz für die Unterstützung von Performancebewusstsein durch die Erweiterung der Quelltextdokumentation von Methoden durch eine Beschreibung des Antwortzeitverhaltens. Antwortzeiten werden mit Hilfe von automatisierten Performance-Unit-Tests erhoben. Das Konzept der Performance Unit Tests basiert dabei auf den Ergebnissen von Horký et al. (2013). In einem ersten Schritt werden der Workload des Tests und die getestete Methode vorbereitet. Anschließend wird der Workload ausgeführt und die Ausführungszeit gemessen. Die Ergebnisse werden dann evaluiert. Zur Vermeidung eines umfangreichen Tests aller Methoden einer Bibliothek werden Messungen nur bei Bedarf ausgeführt. Sobald der Entwickler innerhalb der IDE eine Methode selektiert werden im Hintergrund Performancetests automatisch generiert. Um schnell Ergebnisse rückmelden zu können, werden diese in einem limitierten Umfang hinsichtlich der Parametrisierung und der Anzahl der Wiederholungen ausgeführt. Nach der Rückmeldung der ersten Messungen werden im Hintergrund weitere Evaluationen durchgeführt und die Ergebnisse schrittweise verfeinert. Messungen können sowohl auf dem lokalen Entwicklerrechner als auch auf einer entfernten Maschine durchgeführt werden. Das gemessene Antwortzeitverhalten wird grafisch aufbereitet und in der existierenden Quelltextdokumentation der Methode integriert. Aufgrund der Implementierung für Java-Programme wird die Dokumentation über ein Pop-Up-Fenster im Quelltexteditor angezeigt. Die Evaluierung des Ansatzes wird im Abschnitt 3.3 im Detail beschrieben. Der Ansatz ist nur für Java-Programme ohne Laufzeitabhängigkeiten anwendbar. Für Java-EE-Komponenten, welche externe Dienste wie beispielsweise Datenbanken aufrufen, können nicht ohne Weiteres Performance Unit Tests automatisch generiert und im Hintergrund ausgeführt werden.

Waller et al. (2015) beschreiben einen Ansatz für die Integration von automatisierten Benchmarks in Continuous-Integration-Infrastrukturen zur frühzeitigen Erkennung von Performanceregressionen. Der Microbenchmark von Waller/Hasselbring (2013) wird hierfür periodisch auf einer entfernten Maschine ausgeführt. Ergebnisse werden anschließend grafisch aufbereitet und

in Jenkins[13] angezeigt. Eine Rückmeldung der Ergebnisse zum Entwickler wird von Waller et al. (2015) jedoch nur als Vision beschrieben.

Wert (2015) stellt einen Ansatz für eine automatisierte Diagnose von Performanceproblemen in betrieblichen Softwaresystemen auf Basis von Experimenten vor. Anwendungen werden selektiv instrumentiert und ausgeführt. Der Overhead, der durch die Messung entsteht, wird dadurch minimalisiert. Ergebnisse mehrerer Experimente werden korreliert und zusammengefasst. Ausgehend von einer Taxonomie von Performance-Antipatterns werden die erhobenen Messdaten analysiert und Performanceprobleme identifiziert. Die Existenz von Antipatterns wird durch die Evaluierung von vordefinierten Heuristiken identifiziert. Erkannte Performanceprobleme und deren Ursachen werden in Form eines Berichts zurückgemeldet. Der Ansatz setzt das Deployment der gesamten Anwendung voraus und ist dadurch erst ab der Testphase anwendbar. Eine Evaluation des Ansatzes hinsichtlich des Einflusses auf Softwareentwickler und deren Implementierungen wird nicht durchgeführt. Entwickler müssen die Nutzung des Werkzeugs erlernen und dieses separat betreiben.

3.1.2 Modellbasierte Ansätze für Performancebewusstsein

Existierende modellbasierte Ansätze für die Unterstützung von Performancebewusstsein der Softwareentwickler sind in Tabelle 3-2 aufgelistet und werden im Folgenden beschrieben.

Cortellessa/Frittella (2007) beschreiben ein Framework für die Interpretierung von Performanceanalysen und die Rückmeldung von alternativen Entwürfen an Softwareentwickler. Der Ansatz basiert auf der Erkennung von Performance-Antipatterns in Softwarearchitekturen und setzt die Existenz eines Performancemodells und einer Spezifikation von Performanceanforderungen voraus. Durch die Gegenüberstellung von Performancemetriken des Systems mit den Anforderungen werden Beschreibungen verschiedener Szenarien für weitere Untersuchungen angeboten. Den Autoren zufolge sollen diese Ergebnisse Entwickler dabei unterstützen, die Architektur ihrer Implementierung zu verbessern. Diese geben jedoch auch an, dass der Ansatz nur teilweise automatisiert ist und Expertenerfahrung voraussetzt. Woher die Eingabewerte des Ansatzes hergenommen werden können wird hier nicht beschrieben. Die hier angedeuteten Lösungsvorschläge weisen einen höheren Abstraktionsgrad auf, als die verarbeiteten Performancemodelle.

[13] http://jenkins-ci.org/

Ansatz	Automatisiert	Zeitlicher Bezug: Historische Messung	Zeitlicher Bezug: Vorhersagen	Softwaretyp	Einsichten	IDE-Integration
Cortellessa/Frittella (2007)	-	-	-	-	Beschreibung Szenarien	-
Killian et al. (2010)	X	X	-	Ereignis-gesteuerte Systeme	Hinweise auf Ereignisse	-
Weiss et al. (2013)	X		X	JPA	Antwortzeit	X
Costa et al. (2014a), Costa et al. (2014b)	X	X	-	Storage-Systeme	Antwortzeit	-
Westermann (2014)	X	X	X	-	Antwortzeit, Ressourcen-auslastung, Durchsatz	-
Cito et al. (2015)	X	X	X	Cloud-Anwendung	Metriken für Performance, Workload, Kosten	X
Heger (2015)	-	X	X	-	Lösungs-vorschläge	-

Tabelle 3-2: Modellbasierte Ansätze für die Unterstützung von Performancebewusstsein der Softwareentwickler
Quelle: Eigene Tabelle

Killian et al. (2010) beschreiben einen Ansatz für die Erkennung von Performancebugs, welche bei ereignisgesteuerten Systemen durch Recovery-Mechanismen verschleiert werden. Als Eingabe dienen Informationen über die Ausführung von Ereignissen. Während einer Lernphase wird das Verhalten eines Systems analysiert und Verteilungen über die Dauer der Ereignisse gebildet. Anschließend werden, basierend auf diesen Verteilungen und neuen Eingabedaten, Anomalien erkannt. Zu einer Anomalie werden ähnliche Ausführungen gesucht, die jedoch eine normale Dauer aufweisen. Der erste Schritt, in dem sich die beiden Ausführungen unterscheiden wird dem Entwickler gemeldet. Der Fokus dieses Ansatzes liegt nur auf Bugs, welche unter bestimmten Umgebungsbedingungen auftreten. Der Ansatz wird hinsichtlich des Umfangs der identifizierten Bugs anhand verschiedener Systeme evaluiert. Aufgrund der Dauer der Analysen von bis zu einem Tag eignet sich der Ansatz eher für die Phasen des Testens und des Betriebs. Analysen setzten auch repräsentative Umgebungen voraus. Inwieweit Entwickler dabei unterstützt werden, Bugs zu beheben, wird nicht untersucht. Den Autoren zufolge hängt die

Anzahl der identifizierten Bugs stark von dem Workload während der Datenerfassung ab. Die Arbeit gibt keine Hinweise darauf, welche Fähigkeiten für die Nutzer des Werkzeugs vorausgesetzt werden.

Weiss et al. (2013) beschreiben einen Ansatz für die Vorhersage der Antwortzeit von Diensten der Persistenzschicht auf Basis von maßgeschneiderten Benchmarks während der Implementierung. Der Ansatz adressiert sowohl die Perspektive eines Providers als auch die eines Konsumenten von Persistenzdiensten und wird für die Unterstützung der JPA umgesetzt. Eine Integration mit der Entwicklungsumgebung wird hier als Vision dargestellt. Ausgehend von dem Quelltext einer EntityBean soll im Hintergrund automatisch ein Benchmark erstellt und in einer Testumgebung ausgeführt werden. Die Antwortzeit der einzelnen Methoden der EntityBean sollen grafisch innerhalb der Eclipse IDE angezeigt werden. Entwickler könnten dadurch den Einfluss von Quelltextänderungen evaluieren oder unterschiedliche Implementierungen gegenüberstellen. Der Fokus der vorliegenden Arbeit ähnelt mit dem von Weiss et al. (2013) in verschiedenen Punkten. Der Kontext des Entwicklers dient als Ausgangspunkt einer Antwortzeitvorhersage. Mit Hilfe einer statischen Analyse wird ausgehend von dem betrachteten Quelltext lokal ein Benchmarkmodell erstellt. Anstatt existierende Messungen in dem Modell zu integrieren, werden neue Messungen auf einer Middleware erhoben. Die Ergebnisse werden abschließend im Kontext des Entwicklers zurückgemeldet. Der Ansatz von Weiss et al. (2013) ist auf die JPA limitiert. Der Einfluss des Ansatzes auf die Implementierung der Entwickler wird von Weiss et al. (2013) nicht evaluiert.

Costa et al. (2014a) beschreiben einen Ansatz für die Vorhersage der Antwortzeit von Workflowanwendungen unter bestimmten Konfigurationen. Unter Workflowanwendungen verstehen die Autoren verteilte Prozesse, welche über Dateien in einem gemeinsamen Storage-System kommunizieren und orchestriert werden. Passende Konfigurationen sollen durch die Wiederholung von Vorhersagen identifiziert werden. Costa et al. (2014b) beschreiben die Anwendung dieses Ansatzes während der Entwicklung eines verteilten Storage-Systems. Eine Vorhersage der Antwortzeit des Systems wird unter einer konstanten Parametrisierung nach Quelltextänderungen durchgeführt. Aufgrund der Ergebnisse soll entschieden werden, ob Änderungen angenommen werden, oder korrigiert werden sollen. Die Autoren berichten, dass Entwickler nach einem Monat mit der Anwendung des Ansatzes aufgehört haben. Während der initialen Anwendung wurden relevante Probleme erkannt und behoben. Ab einem bestimmten Punkt lieferte der Ansatz weniger relevante Hinweise. Der Ansatz setzt eine lauffähige Implementierung, die auf einer Zielumgebung deployed wird, voraus.

Westermann (2014) beschreibt ein Framework für das automatisierte Ausführen und Auswerten von Performanceexperimenten für spezifische Zielsetzungen. Mit Hilfe der Experimente werden Messungen erhoben und als Grundlage für die Erstellung von Performancemodellen verwendet. Die resultierenden Modelle sollen dabei die funktionale Abhängigkeit zwischen Eingabeparameter und beobachteten Metriken abbilden. Basierend auf den Performancemodellen kann anschließend das Verhalten der Anwendungen vorhergesagt werden. Entwickler haben dann die Möglichkeit den Einfluss unterschiedlicher Parametrisierungen zu untersuchen. Der Ansatz wird für die Vorhersage der Antwortzeit der Client-Schicht von Web-Anwendungen evaluiert. Die Durchführung von Experimenten setzt eine lauffähige Anwendung, eine Testumgebung sowie Lasttreiber und Monitoring-Werkzeuge voraus. Nach der Änderung der Anwendung muss das Performancemodell auch angepasst oder neu erstellt werden. Dies impliziert die

permanente Verfügbarkeit von Expertenwissen. Entwickler müssen die zugrundeliegenden Werkzeuge erlernen und separat betreiben.

Cito et al. (2015) beschreiben einen Ansatz für Feedback-Driven Development auf Basis der Integration von Monitoringdaten über Cloud-Anwendungen in der IDE. Der Ansatz unterscheidet dabei zwischen Werkzeugen für Analyse und für die Vorhersage des Verhaltens der betrachteten Anwendung. Im Rahmen der Analyse werden Laufzeitinformationen über die Anwendung gesammelt und grafisch in der IDE angezeigt. Im Rahmen der Vorhersage soll der Entwickler über den Einfluss von Quelltextänderungen informiert werden, noch bevor diese produktiv gesetzt werden. Als Grundlage für die Analyse und die Vorhersage dient eine Darstellung von Quelltext als Abhängigkeitsgraph, der mit Hilfe einer statischen Analyse erstellt wird. Jeder Knoten dieses Graphen stellt dabei ein Quelltextartefakt, wie beispielsweise eine Methode oder eine Datenstruktur, dar. Monitoringdaten werden aus der Produktionsumgebung gesammelt, aufbereitet, und auf die Knoten des Abhängigkeitsgraphen abgebildet. Basierend auf dieser Zuordnung werden Monitoringdaten anschließend in der IDE dem entsprechenden Quelltextartefakt zugeordnet und grafisch dargestellt. Für die Vorhersage des Einflusses von Änderungen wird die neue Version des Quelltextes erneut eingelesen und als Abhängigkeitsgraph dargestellt. Anhand der verfügbaren Messungen zu den einzelnen Graphknoten wird die Antwortzeit der übergreifenden Methode neu berechnet. Der Einfluss von Kontrollstrukturen, wie beispielsweise die Anzahl der Iterationen über eine Schleife, kann als Mittelwert über vorhandene Monitoringdaten berechnet werden. Bei einer negativen Auswirkung der Quelltextänderung auf die Performance sieht der Ansatz die Benachrichtigung des Entwicklers vor. Cito et al. (2015) liefern für ihren Ansatz nur ein Konzept. Die Autoren beschreiben eine erste Umsetzung der Analysekomponente des Ansatzes auf Basis der SAP HANA Cloud Platform[14], jedoch keine Evaluation. Der Fokus von Cito et al. (2015) weist einige Ähnlichkeiten zu der vorliegenden Arbeit auf. Beide Ansätze haben die IDE des Entwicklers als Ausgangspunkt und unterstützen die Vorhersage der Antwortzeit von einzelnen Methoden auf Basis von Performancemodellen. Modelle werden mit Hilfe einer statischen Analyse erstellt und mit Monitoringdaten angereichert. Der Ansatz von Cito et al. (2015) verwendet ein proprietäres Modell in Form eines Abhängigkeitsgraphen.

Heger (2015) beschreibt einen Ansatz für ein Empfehlungssystem für die Unterstützung von Entwickler mit Lösungsvorschlägen für Performance- und Skalierbarkeitsprobleme. Entwickler stellen Anfragen in Form von Problembeschreibungen. Ein Entscheidungsträger definiert Entscheidungskriterien für die Priorisierung von Lösungsalternativen. Tester werden mit der Durchführung von Experimenten beauftragt und melden ihre Ergebnisse an das System zurück. Das System stellt dem Entwickler eine Rangliste mit möglichen Lösungen und Aufwandsschätzungen zur Verfügung. Der Ansatz von Heger (2015) unterstützt keine Antwortzeitvorhersagen und setzt die Interaktion verschiedener Rollen und die Verfügbarkeit von Expertenwissen voraus.

[14] http://hcp.sap.com

3.1.3 Adressierte Forschungslücken

Ausgehend von den existierenden Ansätzen für die Unterstützung von Performancebewusstsein der Entwickler werden von dieser Arbeit adressierte Forschungslücken abgeleitet und im Folgenden beschrieben.

Statische Analyse

Problem: Einige der zuvor beschriebenen Arbeiten setzen die Durchführung von Performancetests voraus. Dadurch muss eine Testumgebung verfügbar sein und Zeit für die Bereitstellung aufgewendet werden. Zur Durchführung von Tests werden entsprechende Kenntnisse benötigt. Darüber hinaus setzen manche Arbeiten die Existenz von Unit-Tests bzw. Testfallspezifikationen voraus.

Zielsetzung dieser Arbeit: Der Quelltext der untersuchten Anwendung bzw. Komponente muss nicht lauffähig sein. Aufgrund des Einsatzes einer statischen Analyse kann der Quelltext unvollständig sein und sogar Fehler aufweisen. Dadurch soll der Ansatz früh in der Implementierungsphase angewendet werden können.

Adressierung des Entwicklers

Problem: Einige der zuvor beschriebenen Arbeiten zielen zwar auf die Unterstützung der Entwickler ab, können aber erst in der Test- oder Betriebsphase angewendet werden. Die Rückmeldungen können zu diesem Zeitpunkt jedoch schon obsolet sein.

Zielsetzung dieser Arbeit: Entwickler benötigen kein Expertenwissen und müssen weder externe Werkzeuge betreiben, noch die IDE verlassen.

Messungen

Problem: Einige der zuvor beschriebenen Arbeiten setzten die Existenz von vordefinierten Regeln in Form einer Wissensbasis voraus.

Zielsetzung dieser Arbeit: Zur Durchführung von Vorhersagen wird nur der Quelltext der Anwendung bzw. Komponente und Antwortzeitmessungen von wiederverwendeten Diensten benötigt.

Implementierung

Problem: Einige der zuvor beschriebenen Arbeiten betrachten nur die Kommunikation zwischen Komponenten.

Zielsetzung dieser Arbeit: Bei der Durchführung von Antwortzeitvorhersagen wird die Implementierung einer Komponente bzw. Methode bis auf der Ebene einzelner Anweisungen berücksichtigt. Dadurch kann das Performanceverhalten viel detaillierter abgebildet und vorhergesagt werden.

Vorhersage und Diagnose

Problem: Einige der zuvor beschriebenen Arbeiten unterstützen nur die Erkennung von Performanceproblemen in laufenden Anwendungen.

Zielsetzung dieser Arbeit: Fokus des Ansatzes ist die Durchführung von Antwortzeitvorhersagen. Methodenaufrufe, deren Antwortzeit einen vordefinierten Schwellenwert überschreitet, werden im Quelltexteditor hervorgehoben. Dadurch wird eine Diagnose von Performanceproblemen unterstützt.

3.2 Reverse Engineering von Komponenten

Zur Durchführung von Antwortzeitvorhersagen soll der Ansatz für Performancebewusstsein aus der vorliegenden Arbeit den Quelltext einer Java-Anwendung einlesen und diesen als Performancemodell abbilden. Dieses Vorgehen ist im Forschungsbereich des Reverse Engineering komponentenbasierter Software angesiedelt.

Ansatz	Statische Analyse	Meta-Modell	Automatisiert	Implementierung für Java EE	Puffer-mechanismen	Entwickler-eingaben
Woodside et al. (2001)	-	LQN	X	-	-	X
Israr et al. (2007)	-	LQN	X	-	-	-
Parsons et al. (2008), Parsons (2007)	-	Runtime Path	X	X	-	-
Krogmann (2010), Becker et al. (2010)	X	PCM	X	(X)	-	-
Brosig et al. (2011), Brosig et al. (2009)	-	PCM	X	X	-	-
Brunnert et al. (2013)	-	PCM	X	X	-	-

Tabelle 3-3: ***Komponentenbasierte Ansätze für das Reverse Engineering von Performancemodellen***
Quelle: Eigene Tabelle

Verwandte Arbeiten aus diesem Forschungsbereich präsentieren Ansätze für das Reverse Engineering von Architekturen und des Kontrollflusses (Krogmann 2010, S. 241). Verschiedene Literaturstudien geben einen Überblick über existierende Ansätze für die Rekonstruktion von Architekturen (O'Brien et al. 2002; Koschke 2005; Cornelissen et al. 2009a). Neben der Ableitung von Modellen auf Basis statischer Softwareartefakte, wenden einige Ansätze eine dynamische Analyse an (Krogmann 2010, S. 245ff).

Existierende komponentenbasierte Ansätze für das Reverse Engineering von Performancemodellen sind in Tabelle 3-3 aufgelistet und werden im Folgenden beschrieben. Für eine Beschreibung des Ansatzes aus (Krogmann 2010; Becker et al. 2010) vgl. Abschnitt 4.3.1.1.

Woodside et al. (2001) beschreiben einen Ansatz für die automatische Erstellung von Performancemodellen auf Basis von Traces. Der Ansatz wird mit einem Werkzeug für die Spezifikation von Softwareentwürfen und die Generierung von Quelltext integriert. Softwareentwickler spezifizieren zuerst den Entwurf einer Software. Ausgehend von dem Entwurf wird Quelltext generiert und ausgeführt. Während der Ausführung werden Traces erstellt und anschließend analysiert. Der Entwickler hat die Möglichkeit eine Zielumgebung und ein Deployment zu konfigurieren. Ausgehend von den Traces und den Eingaben des Entwicklers wird ein Performancemodell auf Basis von LQN erstellt. Der Ansatz adressiert vor allem frühe Phasen der Softwareentwicklung, wenn noch kein Quelltext zur Verfügung steht. Für die Erstellung von Performancemodellen auf Basis von Java-EE-Quelltext ist dieser Ansatz weniger geeignet. Einerseits kann die Existenz von Softwareentwürfen, welche den Quelltext genau abbilden, nicht vorausgesetzt werden. Andererseits muss die Java-EE-Anwendung lauffähig sein, damit Traces erstellt werden können. Puffermechanismen werden von Woodside et al. (2001) nicht adressiert.

Israr et al. (2007) beschreiben einen Ansatz für die automatisierte Abbildung von Komponenten und deren Interaktionen als Performancemodell. Den Ausgangspunkt für die Modellerstellung bilden Traces, die Nachrichten zwischen Komponenten erfassen. Aufgrund des abstrakten Formats eignen sich sowohl Traces von laufenden Systemen als auch von ausführbaren Modellen. Traces werden analysiert und als LQN abgebildet. Für eine vollständige Abbildung der Beziehungen zwischen Komponenten setzt der Ansatz eine lauffähige Anwendung, eine Testumgebung und vor allem einen repräsentativen Workload voraus. Ähnlich wie bei dem Ansatz von Woodside et al. (2001) kann die Verfügbarkeit von ausführbaren Softwareentwürfen, welche den Quelltext genau abbilden, nicht vorausgesetzt werden. Im Gegensatz dazu wird jedoch keine Parametrisierung der Modellerstellung durch den Entwickler unterstützt.

Parsons et al. (2008) beschreibt unterschiedliche Vorgehensweisen für die Instrumentierung von Java-EE-Anwendungen zur Erkennung von Komponenten und deren Beziehungen. Hierzu gehören Interceptors für das Abfangen von Anfragen an Komponenten, AOP und die Instrumentierung von Bytecode. Die Autoren beschreiben zusätzlich Möglichkeiten, wie die erfasste Interaktion zwischen Komponenten abgebildet werden kann. Laufzeitpfade stellen eine der Alternativen dar und dienen Parsons (2007) als Grundlage für die Erkennung von Performance-Antipatterns. Die hier beschriebenen Vorgehensweisen adressieren nicht die Abbildung von Puffermechanismen und erlauben keine Parametrisierung der Modellerstellung durch den Entwickler.

Brosig et al. (2011) beschreibt einen Ansatz für die automatisierte Erstellung von Performancemodellen für verteilte komponentenbasierte Anwendungen auf Basis von Monitoringdaten. In einem ersten Schritt werden Komponenten und deren Beziehungen identifiziert. Am Beispiel des SPECjEnterprise2010-Benchmarks wird die Anwendung als zusammengesetzte Komponente identifiziert. Diese besteht aus Komponenten, welche einzelne Klassen darstellen. Die Implementierung des Ansatzes verwendet für das Monitoring herstellerspezifische Funktionen des Anwendungsservers. Für die Abbildung des Kontrollflusses wird die Auslagerung relevanter Anweisungen in separate Methoden vorausgesetzt (Brosig et al. 2009). Durch die Einhaltung einer bestimmten Namenskonvention können die resultierenden Monitoringeinträge während der Analyse interpretiert werden. Zusätzlich berücksichtigt die Modellerstellung auch

Ressourcenverbräuche und die Parametrisierung von Methoden. Als Ergebnis erstellt der Ansatz eine PCM-Instanz.

Brunnert et al. (2013) beschreiben einen Ansatz für die automatische Erstellung von Performancemodellen für laufende Java-EE-Anwendungen. Durch den Einsatz von Servlet Filter werden Nutzeranfragen protokolliert und Gesamtantwortzeiten gemessen. Mit Hilfe von EJB Interceptors werden Aufrufe zwischen den Komponenten protokolliert. Auf Basis dieser Daten werden Komponenten und deren Beziehungen identifiziert. Die gesammelten Informationen werden zuerst aggregiert und dann für die Erstellung einer PCM-Instanz verwendet.

Die meisten der hier beschriebenen verwandten Forschungsarbeiten basieren auf dynamische Analysen. Neben lauffähigen Anwendungen setzten diese implizit die Verfügbarkeit von Testumgebungen und repräsentativen Workload voraus. Allgemein unterstützen diese weder die Abbildung von Puffermechanismen, noch eine Parametrisierung der Modellerstellung durch den Entwickler. Bei dem Ansatz von Woodside et al. (2001) basiert die Modellerstellung vollständig auf Eingaben des Entwicklers und berücksichtigt keinen Quelltext. Nur der Ansatz aus (Krogmann 2010; Becker et al. 2010) unterstützt die Erstellung eines Performancemodells auf Basis einer statischen Analyse.

Andere verwandte Arbeiten, wie beispielsweise (Hrischuk et al. 1999; Courtois/Woodside 2000; Dufour et al. 2003; Canfora et al. 2005; Ross 2006; Zheng et al. 2008), adressieren auch Performanceeigenschaften, unterstützten jedoch nicht die Abbildung von komponentenbasierten Anwendungen (Krogmann 2010, S. 244). Die Ansätze aus (Ross 2006; Dufour et al. 2003; Canfora et al. 2005) basieren auf statischen Analysen.

3.3 Evaluation der Rückmeldung zum Softwareentwickler

Durch die Bereitstellung von Einsichten zielen Ansätze für Performancebewusstsein darauf ab, Entwickler bei der Verbesserung der Performance von Anwendungen zu unterstützen. Wie aus Abschnitt 3.1 hervorgeht, wird der Einfluss der Rückmeldung auf die Performance der Implementierungen nur in wenigen Fällen evaluiert. Nistor et al. (2013b) berichten, dass aufgrund ihrer Analyse mehrerer Open-Source-Projekte acht der 35 identifizierten Bugs durch Entwickler behoben werden konnten (vgl. Abschnitt 3.1.1). Weitere sechs wurden durch die Entwickler als echte Bugs anerkannt, jedoch zu dem Zeitpunkt noch nicht behoben. Costa et al. (2014b) berichten über den Einsatz ihres Werkzeugs in einem Entwicklungsprojekt. Dabei beschreiben sie einige Beispiele, wie aufgrund der Rückmeldungen des Werkzeugs der Entwickler veranlasst wurde, das Verhalten des Systems näher zu untersuchen.

Horký et al. (2015) evaluieren die Rückmeldung zum Softwareentwickler im Rahmen eines Experiments. Die Autoren berichten, dass nur zwölf der 39 Versuchspersonen eine Implementierung abgegeben haben. Dabei haben alle drei Teilnehmergruppen ähnliche Abbrecherquoten aufgewiesen. Die eingereichten Implementierungen haben darüber hinaus stark abweichende Ausführungszeiten aufgewiesen. Diese reichten von wenigen Sekunden bis zu über einem Tag. Aufgrund der hohen Varianz konnte die zugrundeliegende Nullhypothese nicht verworfen werden. Als Alternative zu dem Experiment wurde der Nutzen des Ansatzes anhand von Szenarien erklärt.

Andere verwandte Forschungsarbeiten beschreiben Ansätze für die Bereitstellung von Einsichten für Entwickler innerhalb der IDE, adressieren jedoch andere Aspekte, als Performance. Einige davon führen Evaluationen im Rahmen von Experimenten mit Versuchspersonen durch und werden im Folgenden beschrieben. Zahlreiche andere Forschungsarbeiten evaluieren Ansätze für die Bereitstellung von Einsichten für Entwickler, integrieren diese jedoch nicht in der IDE (Bennett et al. 2008; Quante 2008; Cornelissen et al. 2009b; Wettel et al. 2011).

Rothlisberger et al. (2012) beschreiben einen Ansatz für die Integration von Informationen über das Laufzeitverhalten von Systemen in eine IDE. Damit sollen Wartungsaktivitäten, wie beispielsweise die Lokalisierung einer bestimmen Funktionalität im Quelltext, unterstützt werden. Der Einfluss des Ansatzes wird anhand eines Experiments mit 30 Java-Entwickler, die über durchschnittlich 4,8 Jahren Berufserfahrung verfügt haben, untersucht. Die unabhängige Variable des Experiments ist die Verfügbarkeit des Ansatzes in der IDE. Die Abhängige Variable ist die Bearbeitungszeit der einzelnen Aufgaben und die Korrektheit der Ergebnisse. Versuchspersonen mussten im Rahmen des Experiments fünf Aufgaben, bestehend aus jeweils zwei Wartungsaktivitäten, im Zusammenhang mit einem Open-Source-Werkzeug durchführen. Während der Bearbeitung wurden diese überwacht. Versuchspersonen aus der Testgruppe haben durchschnittlich 17,5 % weniger Zeit für die Bearbeitung der Aufgaben benötigt und die Korrektheit ihrer Ergebnisse war durchschnittlich um 33 % höher.

Ponzanelli et al. (2014) beschreiben einen Ansatz für Integration einer Suchfunktion für Inhalte der Onlineplattform Stackoverflow[15] in eine IDE. Der Ansatz evaluiert den aktuellen Kontext des Entwicklers, sucht nach entsprechenden Einträgen auf der Stackoverflow-Plattform, bewertet die Relevanz der Ergebnisse und benachrichtigt den Entwickler über die verfügbaren Informationen. Mit Hilfe einer Umfrage wird zuerst die Relevanz der abgerufenen Inhalte evaluiert. Im Rahmen eines Experiments wird der Einfluss auf Entwickler während der Bearbeitung einer Entwicklungs- und einer Wartungsaufgabe evaluiert. Die unabhängige Variable des Experiments ist die Verfügbarkeit des Ansatzes in der IDE. Als unabhängige Variablen wurden die Vollständigkeit und Korrektheit der Ergebnisse festgelegt. Zur Bearbeitung der Aufgaben hatten die Versuchspersonen jeweils 90 Minuten Zeit. Am Experiment nahmen 12 Versuchspersonen teil. Jede Versuchsperson hat jeweils eine Aufgabe mit und eine ohne Unterstützung durchgeführt. Zur Vermeidung von Lerneffekten, mussten dabei unterschiedliche Aufgabentypen bearbeiten. Der Median über die Vollständigkeit der Ergebnisse war bei Verfügbarkeit der Unterstützung während der Entwicklungsaufgabe um 40% höher. Bei der Bearbeitung der Wartungsaufgabe war dieser um 10% höher.

[15] http://stackoverflow.com/

3.4 Versionskontrollsysteme für Performancemodelle

Einen Überblick über existierende Ansätze für die Versionierung und Verwaltung von Modellen geben Altmanninger et al. (2009) im Rahmen ihrer Literaturstudie. Die hier beschriebenen Ansätze können auf eine der folgenden Schichten eingeordnet werden:

- **Daten:** Versionskontrollsysteme wie beispielsweise Subversion verwalten Dateien und versionieren diese auf Zeilenebene. Wie in Abschnitt 6.3.1 beschrieben wird, unterstützen diese eine parallele Bearbeitung graphbasierter Modelle nur bedingt.
- **Technologie (Meta-Metamodell):** Modellrepositories für die Versionierung und Verwaltung von Instanzen eines beliebigen Meta-Modells auf der Basis einer bestimmten Technologie. EMFStore und CDO unterstützten beispielsweise EMF-basierte Modelle.
- **Domäne (Meta-Modell):** Werkzeuge für die Versionierung und Verwaltung domänenspezifischer Modelle unter Einsatz eines Modellrepositories. Das Werkzeug Unicase[16] verwaltet beispielsweise UML-Modelle auf Basis von EMFStore. Das im Rahmen dieser Arbeit entwickelte Modellrepository adressiert Performancemodelle in Form von PCM-Instanzen und kann auf dieser Ebene angesiedelt werden.

Keines der Ansätze von Altmanninger et al. (2009) adressiert die Versionierung und Verwaltung von Performancemodellen. Auch insgesamt existieren nur wenige Forschungsarbeiten mit diesem Fokus. Koziolek (2010) argumentiert, dass Komponentenspezifikationen durch den Einsatz von Modellrepositories wiederverwendbar sein sollten, damit der initiale Aufwand für deren Erstellung amortisiert werden kann. Ein Lösungsvorschlag wird jedoch nicht beschrieben.

Woodside et al. (2007) beschreiben eine *Performance Knowledge Base* für die Integration und Speicherung von Wissen über die Performance eines Systems auf der Grundlage von Modellen. Das Modell eines Systems sowie Performancedaten und Evaluationsergebnisse sollen über die Zeit und über unterschiedliche Versionen des Systems integriert und gespeichert werden. Unter Performancedaten werden sowohl Messungen als auch Workloadspezifikationen und Konfigurationen verstanden. Weiterhin unterscheiden die Autoren zwischen dem Modell als abstrakte Abbildung des Systems und einer parametrisierten Instanz. Nur die abstrakte Abbildung soll verwaltet und versioniert werden. Instanzen sollen auf der Grundlage des Modells und von Performancedaten automatisch erstellt werden. Ähnlich zu diesem Ansatz, soll das Repository für Performancemodelle auch die Speicherung von Modellen und Performancedaten über die Zeit und über verschiedene Systemversionen unterstützen. Im Gegensatz zu der Performance Knowledge Base soll das Repository für Performancemodelle jedoch nicht zur Speicherung von Evaluationsergebnissen dienen. Eine Trennung zwischen dem abstrakten Modell und den Performancedaten ist nicht das Ziel des Repository für Performancemodelle. Komponentenspezifikationen können hier Ressourcenverbräuche spezifizieren. Durch den Einsatz von PCM als Meta-Modell existiert aber die Trennung zwischen den Komponentenspezifikationen und der Workload- und Hardwarespezifikation. Die Vision von Woodside et al. (2007) sieht eine Befüllung der Wissensbasis mit automatisch erzeugten Performancemodellen vor. Das Konzept der

[16] http://unicase-ls1.github.io/unicase/

Performance Knowledge Base adressiert die Aspekte der Organisierung von Modellinhalten und der Unterstützung hardwarespezifischer Ressourcenbedarfe nicht.

Svorobej et al. (2015) beschreiben einen Ansatz für die Simulation des Verhaltens von Cloud-Rechenzentren. Für die Evaluation der Performance der Gesamtinfrastruktur werden Monitoringdaten auf der Ebene einzelner VMs gesammelt. Auf Basis der Monitoringdaten werden EMF-Modelle für die physische und logische Abbildung der Cloud-Infrastruktur erstellt. Ein CDO-basiertes Repository ist für die persistente Speicherung der Modelle zuständig. Das aktuelle Modell der Infrastruktur wird von einem Simulationswerkzeug aus dem Repository ausgelesen. Dieses wandelt die Modelle der physischen und logischen Infrastruktur nach einer PCM-Instanz um. Zur Evaluierung der Performance der VMs wird die PCM-Instanz simuliert. Bei der Implementierung des Simulationswerkzeugs handelt es sich um eine Eclipse IDE. Aus dem CDO-Repository ausgelesene Modelle können lokal mit Hilfe von EMFStore persistiert werden. Eine Rückkopplung von lokalen Änderungen an den Modellen mit dem zentralen Repository wird nicht adressiert. Dadurch können Änderungen nicht ausgetauscht werden. Das Simulationswerkzeug unterstützt die Versionierung der lokalen Modelle auf Basis von EMFStore-Revisionen. Varianten von Modellen oder Modellelemente nach Conradi/Westfechtel (1997) werden nicht unterstützt (vgl. auch Abschnitt 6.2.1). Auch die Verwaltung von hardwarespezifischen Ressourcenbedarfe wird durch den Ansatz nicht adressiert.

Einige andere Ansätze adressieren die Versionierung und Verwaltung von UML-Modellen. Kuryazov/Winter (2015) beschreiben einen Ansatz für eine kollaborative Modellierung auf Basis des Austausches von Änderungen zwischen den Bearbeitern eines Modells. Änderungen werden lokal berechnet und mit Hilfe eines zentralen Synchronisierungsdienstes an andere Nutzer verteilt. Durch den Austausch einzelner Änderungen anstatt ganzer Modelle soll eine Bearbeitung in Echtzeit unterstützt werden. Der Ansatz am Beispiel von UML-Klassendiagrammen demonstriert. Kuryazov/Winter (2014) adressieren auch das Festschreiben von Änderungen auf einem zentralen Repository. Eine Erweiterung von EMFStore für die Unterstützung des Austausches von Änderungen in Echtzeit auf Basis von Peer-to-Peer wird in (Krusche/Bruegge 2014) beschrieben.

Murta et al. (2007) beschreiben ein Software Configuration Management System für UML-Modelle. Für die Verwaltung der Modelle ist eine zentrale Ablage auf Basis des NetBeans Metadata Repository[17] zuständig. Modellinhalte können durch Nutzer über Web Services abgefragt werden. Nutzer bearbeiten Modelle lokal und schreiben Änderungen auf der zentralen Ablage fest. Eine Identifizierung und Auflösung eventueller Konflikte wird durch den Ansatz unterstützt. Modelle werden als hierarchische Struktur, bei der jedes einzelne Unterelement eine eigene Revision besitzt, verwaltet. Die Änderung an einem Unterelement führt dabei auch zu einer neuen Revision des übergeordneten Elements. Im Rahmen einer Evaluation wird die Lese- und Schreibgeschwindigkeit des Systems anderen dateibasierten Versionskontrollsystemen gegenübergestellt.

Eine domänenspezifische Modellierungsumgebung auf Basis von EMFStore wird in (Bruegge et al. 2008) beschrieben. Das Werkzeug unterstützt die Bearbeitung und Verwaltung unterschiedlicher Modelle der Softwareentwicklung, wie z.B. UML-Modelle.

---

[17] https://netbeans.org/

3.5 Zusammenfassung

Der Ansatz für Performancebewusstsein aus der vorliegenden Arbeit soll die Vorhersage der Antwortzeiten von Java-EE-Komponenten unterstützen. Mehrere der relevantesten Systemtypen, die in der Literaturstudie von Danciu et al. (2015c) identifiziert wurden, sind durch den Fokus dieses Ansatzes abgedeckt. Der Einfluss von wiederverwendeten Komponenten stellt die Grundlage für Antwortzeitvorhersagen dar. In erster Linie adressiert der Ansatz dadurch ein komponentenbasiertes System. Java-EE-Anwendungen können aber auch als verteiltes System implementiert werden und sind Web-basiert.

4 Ansatz für die Unterstützung von Performancebewusstsein

In diesem Kapitel wird der Ansatz für die Unterstützung von Performancebewusstsein in Java-EE-Entwicklungsumgebungen vorgestellt. Als erstes wird die Relevanz des adressierten Schwerpunkts motiviert (siehe Abschnitt 4.1). Im Abschnitt 4.2 wird das zugrundeliegende Konzept erläutert. Darauf aufbauend wird im Abschnitt 4.3 die technische Umsetzung des Ansatzes beschrieben. Eine Untersuchung der Genauigkeit von Vorhersagen über die Antwortzeit von Komponenten wird in Abschnitt 4.4 vorgestellt. Abschließend werden die Ergebnisse zusammengefasst, sowie weitere Forschungsrichtungen aufgezeigt (siehe Abschnitt 4.5).

Ein erstes Konzept für die Unterstützung von Performancebewusstsein in Java-EE-Entwicklungsumgebungen wurde in (Danciu et al. 2014) veröffentlicht. Die in diesem Kapitel beschriebene Gestaltung, Umsetzung und Validierung des Ansatzes wurden in (Danciu et al. 2015b) veröffentlicht.

4.1 Motivation

4.1.1 Problemstellung

Aus Nutzersicht stellt die Antwortzeit einer Anwendung die wichtigste Performancemetrik dar (Kohavi/Longbotham 2007). Auch für die Kommunikation zwischen zwei Anwendungen versuchen Unternehmen akzeptable Antwortzeiten mit Hilfe von SLA durchzusetzen. Die Gesamtantwortzeit einer Komponente wird, wie im Abschnitt 2.3.2 beschrieben, von der Implementierung, Parametrisierung, Deployment-Plattform sowie durch Ressourcenkonflikte und wiederverwendete Dienste beeinflusst. Bei der Umsetzung komplexer Szenarien in einer stark integrierten Umgebung, getrieben von Entwurfsmuster, wie Microservices und serviceorientierte Architekturen, stellt der Einfluss der wiederverwendeten Dienste die größte Herausforderung dar. Bei diesen Diensten kann es sich um Schnittstellen zu anderen Komponenten oder zu externen Systemen handeln. Mit steigender Komplexität weisen Unternehmensanwendungen zunehmend mehr Komponenten und Schichten auf. Je stärker Systeme miteinander vernetzt werden, desto mehr Web Services werden aufgerufen. Dies wiederum impliziert mehr wiederverwendete Dienste, welche einen noch höheren Einfluss auf die Gesamtantwortzeit der Komponenten haben.

Während der Implementierung einer Komponente ist es daher wichtig beurteilen zu können, wie hoch der Einfluss von wiederverwendeten Diensten ist. Der Entwickler einer Komponente muss unter anderem folgende Fragen adressieren (Danciu et al. 2014):

- Können die Anforderungen an die Antwortzeit der Komponente mit der aktuellen Implementierung eingehalten werden?
- Werden die Anforderungen an die Antwortzeit durch die Wiederverwendung eines bestimmten Dienstes verletzt?
- Führt eine Änderung an der Implementierung der Komponente zu einer Verletzung der Anforderungen an die Antwortzeit?

Aufgrund von komplexen Architekturen, Lebenszyklen und Governancestrukturen von Unternehmensanwendungen (Brunnert et al. 2014), stellt die Beantwortung dieser Fragen eine große

Herausforderung dar. Die Struktur, Verteilung und Auslastung der Dienste ist für den Entwickler nicht transparent. Somit benötigt der Entwickler aufbereitete Performancemessungen. Dienste und Komponenten unterliegen einer ständigen Evolution. Damit kann sich auch das Antwortzeitverhalten kontinuierlich ändern. Die organisatorische Zuordnung von Diensten erschwert zusätzlich den Zugang zu Informationen, wie z.B. Monitoringergebnissen.

Die notwendigen Aktivitäten zur Bestimmung des Einflusses von wiederverwendeten Diensten sind in Abbildung 4-1 dargestellt. Der Entwickler muss ausgehend vom Quelltext der Komponente den Kontrollfluss und die aufgerufenen Dienste extrahieren. Das Verhalten der Dienste muss mit Hilfe eines Monitoringwerkzeugs erfasst und verdichtet werden. Der Aufrufbaum und die Messungen müssen anschließend zu einem Modell integriert werden. Dieses Modell kann dann für die Evaluation der Gesamtantwortzeit analysiert werden. Für die Bearbeitung dieser Aktivitäten benötigt der Entwickler umfassendes Wissen über Performance Engineering und entsprechende Werkzeuge.

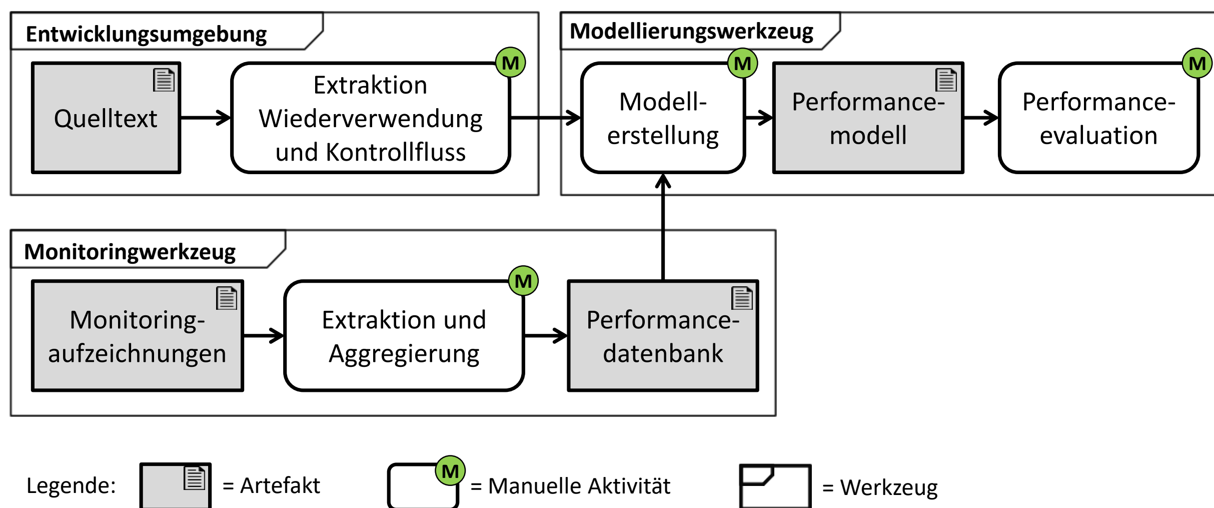


Abbildung 4-1: ***Schritte für die manuelle Abschätzung des Antwortzeitverhalten von Komponenten einer Unternehmensanwendung***
Quelle: In Anlehnung an Danciu et al. (2015b)

Können Anforderungen an die Antwortzeit wegen der wiederverwendeten Dienste nicht eingehalten werden, müssen Maßnahmen, wie z.B.:

- Auswahl eines anderen Dienstes mit ähnlicher Funktionalität und besserer Performance, oder
- Optimierung des wiederverwendeten Dienstes

ergriffen werden. Werden diese Maßnahmen nicht rechtzeitig ergriffen kann es im Kontext komplexer Unternehmenslandschaften (Vögele et al. 2014) zu signifikanten Verzögerungen führen.

4.1.2 Forschungsziele

Für die Unterstützung von Performancebewusstsein bei Entwicklern von komponentenbasierten Unternehmensanwendungen soll ein Ansatz entwickelt werden. Der Ansatz verfolg das Ziel Entwicklern von Java-EE-Anwendungen kontextspezifische Einsichten über die erwartete Antwortzeit von Komponenten automatisiert bereitzustellen. Wenn von hohen Antwortzeiten auszugehen ist, soll der Entwickler auf die möglichen Ursachen des Performanceproblems hingewiesen und beeinflusst werden, entsprechende Optimierungen durchzuführen. Die Gestaltung des Ansatzes soll folgende Aspekte adressieren:

- Struktur und Wiederverwendung von Komponenten: Die Struktur von Komponenten und deren Wiederverwendung sollen auf Basis von Quelltext identifiziert werden. Es soll untersucht werden, welche Zusammenhänge identifiziert und abgebildet werden können.
- Kontrollfluss von Komponenten: Der Kontrollfluss der Komponenten wird im Quelltext mittels Anweisungen spezifiziert. Es soll untersucht werden, wie Anweisungen eingelesen und abgebildet werden können.
- Parametrisierung von Komponenten: Die Parametrisierung von Komponenten beeinflusst deren Antwortzeitverhalten. Diese kann jedoch nur zur Laufzeit beobachtet werden. Es soll untersucht werden, wie Informationen über die Parametrisierung von Komponenten im Quelltext integriert werden können.
- Berücksichtigung von Puffermechanismen: Rückgabewerte können von Komponenten zwischengespeichert werden. Es soll untersucht werden, wie diese Mechanismen bei der Vorhersage von Antwortzeiten berücksichtigt werden können.
- Integration von statischen und dynamischen Informationen: Statische Informationen über die Struktur von Komponenten sollen mit dynamischen Informationen über das Antwortzeitverhalten von wiederverwendeten Diensten integriert werden.
- Visualisierung der Einsichten: Einsichten über die erwartete Antwortzeit von Komponenten sollen dem Entwickler zurückgemeldet werden. Die Verarbeitung der Informationen soll den Arbeitsfluss des Entwicklers nicht stören.

Nach der Entwicklung eines Konzepts soll der Ansatz praktisch umgesetzt und in einer Entwicklungsumgebung integriert werden. Die Korrektheit der bereitgestellten Einsichten soll anhand einer konkreten Unternehmensanwendung evaluiert werden.

4.1.3 Kontext

Der Ansatz für Performancebewusstsein adressiert die Entwicklung von Java-EE-Anwendungen. Die Vorhersage der Antwortzeiten einer neuen Komponente setzt die Wiederverwendung anderer Komponenten, für die Antwortzeitmessungen existieren, voraus. Antwortzeitmessungen werden durch Instrumentierung aus dem produktiven Betrieb oder aus Lasttestumgebungen erhoben. Diese werden in einer zentralen Datenbank gespeichert und aggregiert.

Der Ansatz wird als Erweiterung in eine existierende Java-EE-Entwicklungsumgebungen integriert. Die Evaluierung der Antwortzeit einer neuen Komponente setzt die Verfügbarkeit des entsprechenden Quelltextes voraus.

Die Evaluierung der Korrektheit des Ansatzes wird innerhalb einer Testumgebung unter kontrollierten Bedingungen und unter Verwendung von Standard-Hardware durchgeführt. Vorhersagen werden für eine beispielhafte Implementierung einer Java-EE-Anwendung durchgeführt.

4.2 Konzept für die Unterstützung von Performancebewusstsein

Die wichtigsten Elemente des Ansatzes sind in Abbildung 4-2 dargestellt. Der Kontext des Entwicklers, im Sinne einer fokussierten Komponente, Schwellwerten für akzeptable Antwortzeiten und dem erwarteten Workload, stellt den Ausgangspunkt des Ansatzes dar. Eine fokussierte Komponente befindet sich in der Implementierung und der entsprechende Quelltext wird mit Hilfe eines Editors dargestellt oder bearbeitet. Der aktuelle Kontext wird durch die Entwicklungsumgebung im Hintergrund automatisch auf ein Performancemodell abgebildet. Das Modell beschreibt die Implementierung der Komponente und wird mit Informationen über das Verhalten von wiederverwendeten Diensten angereichert. Mit Hilfe des Modells werden Vorhersagen über die Antwortzeit der fokussierten Komponente durchgeführt und im Kontext des Entwicklers rückgemeldet.

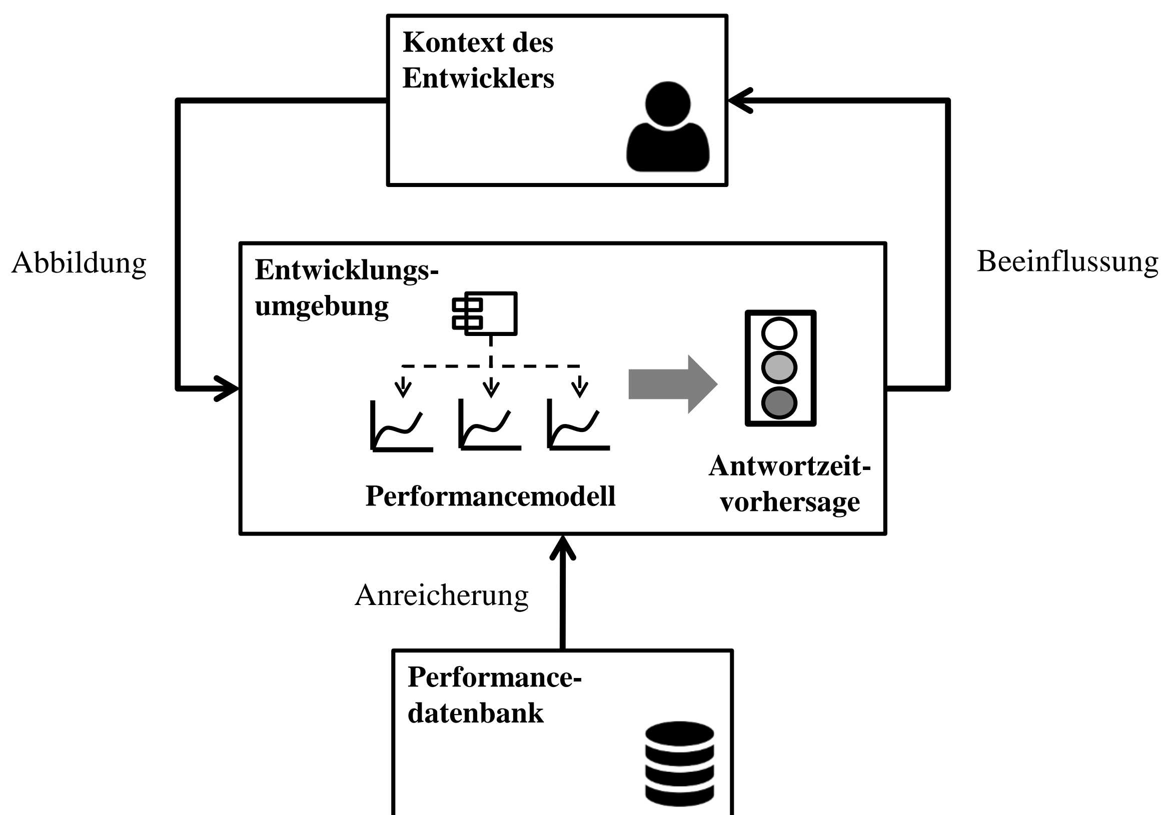


Abbildung 4-2: ***Ansatz für die Vorhersage der Antwortzeit von Komponenten einer Unternehmensanwendung zur Unterstützung von Performancebewusstsein***
Quelle: Eigene Darstellung

Bei der Entwicklung komponentenbasierter Software wird die Realisierung von lose miteinander gekoppelten Einheiten verfolgt. Durch eine Trennung der Belange bzw. Verantwortlichkeiten soll die Wiederverwendung von Komponenten unterstützt werden. Der hier beschriebene Ansatz basiert auf der Annahme, dass für eine Teilmenge der wiederverwendeten Dienste,

jedoch nicht für die fokussierte Komponente Performancemessungen existieren. Der Einfluss von wiederverwendeten Diensten stellt die Grundlage für Vorhersagen dar (siehe Abbildung 4-3). Wiederverwendung im Sinne von Referenzen zu anderen Komponenten wird zuerst im Quelltext identifiziert. Die entsprechenden Aufrufe werden zu einem Kontrollfluss modelliert. Mit Hilfe von Informationen über das Verhalten der wiederverwendeten Komponenten wird der Kontrollfluss parametrisiert und anschließend analysiert.

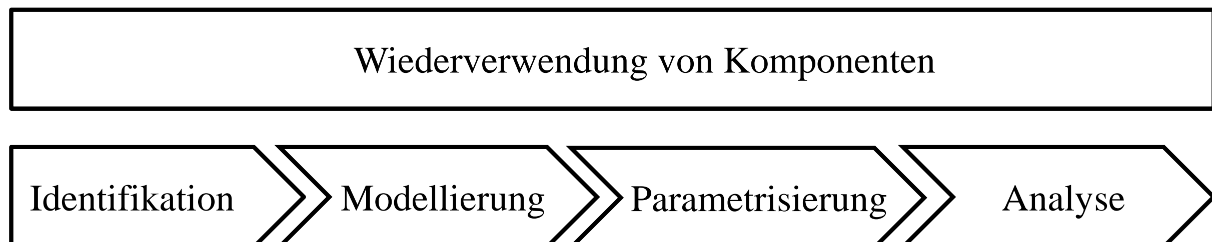


Abbildung 4-3: Idee für die Vorhersage der Antwortzeit von Komponenten
Quelle: Eigene Darstellung

Für die Abschätzung der Antwortzeit werden neben den wiederverwendeten Diensten auch die Implementierung und die Parametrisierung der fokussierten Komponente einbezogen. Wie diese Einflussfaktoren berücksichtigt werden ist in Tabelle 4-1 beschrieben. Die Implementierung von Komponenten wird durch Java-Quelltext repräsentiert. Dieser Quelltext ist in Java-Klassen organisiert, die wiederum eine Menge von Methoden enthalten. Jede Methode besteht aus einer Menge von Anweisungen, z.B. zur Steuerung des Kontrollflusses, Manipulation von Variablen oder zum Aufruf von Methoden. Für die Ausführung von Anweisungen werden Ressourcen, wie z.B. CPU oder Hauptspeicher, benötigt. Das Warten auf freie Ressourcen, sowie deren Nutzung tragen zur Gesamtantwortzeit bei. Aufgrund der Annahme, dass keine Performancemessungen für die fokussierte Komponente vorliegen, ist dieser Ressourcenverbrauch nicht bekannt und wird im Performancemodell nicht abgebildet. Aufrufe von Diensten anderer Komponenten werden als wiederverwendete Dienste betrachtet. Für diese Fälle werden von der Performancedatenbank Antwortzeiten abgerufen und im Performancemodell integriert. Die Parametrisierung der Dienstaufrufe einer Komponente entspricht einer Menge von Werten eines bestimmten Typs. Die Ausprägung dieser Werte wird von einem Aufrufer bestimmt und beeinflusst das Laufzeitverhalten der Komponente. Aufgrund der Annahme, dass die fokussierte Komponente noch nicht ausführbar ist, können mögliche Ausprägungen und deren Wahrscheinlichkeit nicht automatisch erhoben werden. Der Entwickler der Komponente kann unter Umständen über Informationen zur Parametrisierung verfügen. Die Wertebereiche von Parametern sind typischerweise sehr umfangreich und die Spezifikation von möglichen Werten damit sehr aufwändig. Daher unterstützt der Ansatz die Spezifikation von Meta-Informationen über die Parametrisierung. Entwickler können die zu erwartende Anzahl der Elemente in einer Liste, oder die Wahrscheinlichkeit, dass ein boolescher Wert wahr ist, spezifizieren. Diese Angaben werden für die Verfeinerung des Performancemodells herangezogen. Wenn eine Liste als Parameter übergeben und in einer Schleife iteriert wird, so kann die Anzahl der Iterationen aufgrund der Anzahl der Elemente spezifiziert werden. Bei booleschen Werten, die in If-Anweisungen referenziert werden, kann die Wahrscheinlichkeit einer bedingten Anweisung oder einer Verzweigung festgelegt werden. Durch das Ignorieren des Ressourcenverbrauchs von

Anweisungen, können im simulierten Verhalten der Komponente keine Ressourcenkonflikte entstehen und die Spezifikation der Deployment-Plattform ist irrelevant.

Einflussfaktor	Berücksichtigung	Beschreibung
Implementierung	Ja	Berücksichtigt werden: - Struktur der Komponenten - Kontrollfluss der Dienste Der Ressourcenverbrauch von Anweisungen im Kontrollfluss wird vernachlässigt.
Wiederverwendete Dienste	Ja	Die Antwortzeiten der wiederverwendeten Dienste werden berücksichtigt.
Parametrisierung	Ja	Die Parametrisierung der Dienstaufrufe wird auf einer Meta-Ebene berücksichtigt: - Länge von Listen - Wahrscheinlichkeiten für boolesche Variablen
Ressourcen-konflikte	Nein	Dadurch, dass der Ressourcenverbrauch von Anweisungen nicht abgebildet wird, kann es auch nicht zu Ressourcenkonflikten kommen.
Deployment-Plattform	Nein	Dadurch, dass der Ressourcenverbrauch von Anweisungen nicht abgebildet wird, hat die Ausprägung der Plattform keinen Einfluss.

Tabelle 4-1: ***Berücksichtigung der Einflussfaktoren auf die Performance von Komponenten durch den Ansatz für Performancebewusstsein***
Quelle: In Anlehnung an Koziolek (2010)

Die wichtigsten Aspekte hinsichtlich der Abbildung der Struktur, des Kontrollflusses, der Parametrisierung und des Antwortzeitverhaltens werden im Folgenden beschrieben.

4.2.1 Abbildung der Komponentenstruktur

Für die Abbildung von Komponenten und deren Beziehungen in einem Performancemodell wird der Quelltext einer Anwendung eingelesen (siehe Abbildung 4-4). Die Ableitung eines Entwurfs, wie. z.B. das Modell einer Anwendung, aus einer Implementierung, wie z.B. dem Quelltext einer Anwendung, wird als *Reverse Engineering* bezeichnet (Chikofsky/Cross 1990). Hinsichtlich der Strategie welche Teile des Quelltextes zu einer Komponente zusammengefasst und auf welcher Abstraktionsebene Komponenten dargestellt werden, existieren in der Literatur unterschiedliche Ansätze. Anquetil/Lethbridge (1999) bilden Gruppen von Dateien, basierend auf der Ähnlichkeit ihrer Namen. Einige Ansätze untersuchen die Abbildung auf Subsysteme (Müller et al. 1993; Mitchell/Mancoridis 2001). Andere verfolgen die Identifizierung von Klassen, welche bestimmte Beziehungen aufweisen und dadurch bekannte Entwurfsmuster

umsetzen (Keller et al. 1999; Sartipi 2003). Favre et al. (2001) fassen eine oder mehrere Klassen zu einer Komponente zusammen und stellen diese graphisch dar. Strein et al. (2007) betrachten einzelne Klassen als Komponenten. Abhängig von der Zielsetzung bieten diese Ansätze unterschiedliche Vor- und Nachteile. Eine große Menge an Komponenten kann z.B. die Lesbarkeit eines Modells stark beeinträchtigen. Eine informelle Darstellung von Komponenten hindert die Verarbeitung der Modelle durch andere Werkzeuge (Krogmann 2010, S. 48).

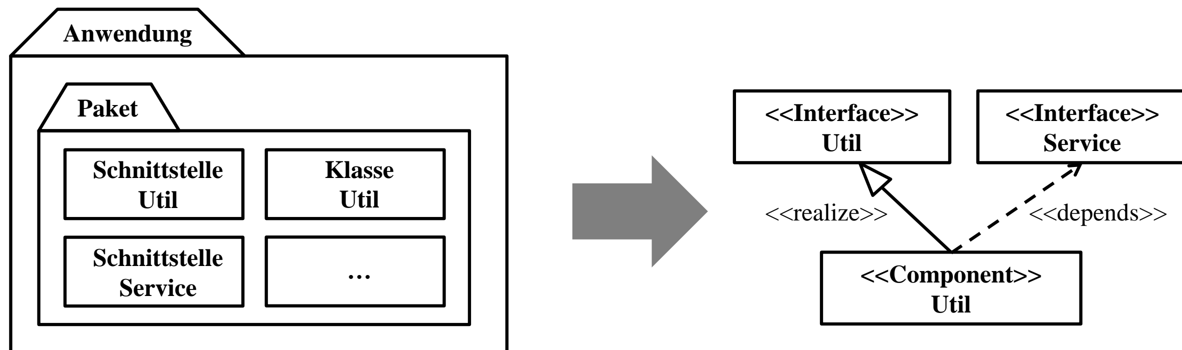


Abbildung 4-4: Beispiel für die Abbildung einer Java-EE-Anwendung auf Komponenten und Schnittstellen
Quelle: Eigene Darstellung

Der Fokus dieses Ansatzes liegt auf Java-EE-Komponenten und die Verwendung von Performancemodellen für automatisierte Performanceevaluationen. Java-Klassen, die Servlets oder Enterprise Beans implementieren, stellen im Kontext von Java EE implizit Komponenten dar (DeMichiel/Shannon 2013). Dadurch, dass die zu erzeugenden Performancemodelle nicht für die Bearbeitung durch Nutzer gedacht sind, stellt eine erhöhte Komplexität keinen Nachteil dar. Für eine möglichst detaillierte Abbildung des Quelltextes betrachtet dieser Ansatz Java-Klassen als einzelne Komponenten. Die abgebildeten Komponenten entsprechen dabei der Definition nach Szyperski (2002).

Java-EE-Komponenten, wie z.B. Enterprise Beans, werden auch als einfache Java-Klassen implementiert und erhalten über Annotationen zusätzliche Eigenschaften. Es ist möglich, dass die Geschäftslogik einer Anwendung sowohl durch Java-EE-Komponenten als durch einfache Klassen umgesetzt wird. Beispielsweise kann eine Enterprise Bean bestimmte Funktionen auf eine Hilfsklasse auslagern. Sogar die Kommunikation zwischen zwei Java-EE-Komponenten kann über Hilfsklassen erfolgen. Bei der Abbildung der Komponentenstruktur werden beide Arten von Klassen berücksichtigt.

Beziehungen zwischen Komponenten werden mit Hilfe von Schnittstellen abgebildet. Wie in Abbildung 4-4 dargestellt, kann eine Komponente Schnittstellen anbieten oder benötigen. Angebotene Schnittstellen können wiederum von anderen Komponenten benötigt werden. Java-Schnittstellen werden als Schnittstelle zwischen Komponenten betrachtet. Hinsichtlich der Abbildung von Schnittstellen existieren verschiedene Spezialfälle, welche im Folgenden beschreiben werden.

Explizite Schnittstellen

Die Java-Programmiersprache sieht die Spezifikation von Methoden, die von einer Klasse angeboten werden, mit Hilfe von Java-Schnittstellen (engl. interface) vor. Jede Java-Schnittstelle einer Klasse wird auf jeweils eine Komponentenschnittstelle abgebildet. Methodendeklarationen werden als Dienstspezifikationen dargestellt.

Implizite Schnittstellen

Im Gegensatz zu der Komponentenorientierten Softwareentwicklung erzwingt die Java-Programmiersprache nicht die Verwendung von Java-Schnittstellen. Klassen können einander auch ohne die Existenz von Schnittstellen referenzieren. Für diese Fälle werden implizite Schnittstellen angenommen. Aus den Methoden einer Klasse wird eine künstliche Schnittstelle abgeleitet.

Hierarchie von Schnittstellen

Java-Schnittstellen können einander erweitern und somit eine Hierarchie von Schnittstellen bilden. Eine Subschnittstelle erbt die Methodendeklarationen der Oberschnittstelle und kann diese um neue Methodendeklarationen erweitern. Implementiert eine Klasse eine Subschnittstelle müssen auch die Methodendeklarationen der Oberschnittstellen implementiert werden. Der Ansatz sieht die genaue Abbildung dieser Hierarchie als Komponentenschnittstellen, die einander erweitern, vor.

Hierarchie von Komponenten

Java-Klassen können einander um neue Attribute und Methoden erweitern. Diese Hierarchie wird abgebildet, indem die resultierende Subkomponente alle Schnittstellen der Oberkomponente und zusätzlich noch die eigenen anbietet.

Treten die weiter oben beschriebenen Fälle gleichzeitig auf, so werden auch die entsprechenden Vorgehensweisen für deren Abbildung kombiniert. Wenn z.B. eine Java-Klasse die Methodendeklarationen einer Java-Schnittstelle implementiert und auch eigene Methoden definiert, werden sowohl eine explizite als auch eine implizite Schnittstelle abgebildet.

4.2.2 Klassifizierung von Komponenten

Der Ansatz unterscheidet zwischen internen und externen Komponenten. Für interne Komponenten werden sowohl Schnittstellen als auch die Implementierung der Dienste abgebildet. Hierfür muss der Quelltext der entsprechenden Klasse verfügbar sein. Interne Komponenten können sich gegenseitig aufrufen. Ein Performancemodell besteht aus mindestens einer internen Komponente - die fokussierte Komponente, deren Antwortzeit vorhergesagt werden soll. Für externe Komponenten können in der Performancedatenbank schon Messungen der Antwortzeit existieren. Statt der Implementierung wird bei externen Komponenten das Antwortzeitverhalten modelliert. Existieren für eine Java-Klasse Quelltext und Performancemessungen, kann diese sowohl als interne als auch als externe Komponente abgebildet werden. Die Möglichkeiten der Klassifizierung von Komponenten sind in Tabelle 4-2 aufgelistet.

		Performancemessungen	
		Verfügbar	**Nicht verfügbar**
Quelltext	**Verfügbar**	Interne / Externe Komponente	Interne Komponente
	Nicht Verfügbar	Externe Komponente	Nicht abgebildet

Tabelle 4-2: ***Klassifizierung von internen und externen Komponenten***
Quelle: Eigene Tabelle

Die Klassifizierung von Komponenten, für die sowohl der Quelltext als auch Performancemessungen verfügbar sind, als intern oder extern impliziert Vor- und Nachteile. Wird die Implementierung einer Komponente modelliert, können zusätzliche wiederverwendete Dienste identifiziert und abgebildet werden. Verwendet diese Komponenten aber keine anderen Dienste wird ihr Antwortzeitverhalten vollständig ignoriert. Wird das Antwortzeitverhalten modelliert und liegen Performancemessungen nur für eine ältere Version vor, wird die aktuelle Implementierung der Komponente ignoriert. Der Ansatz sieht vor, dass diese Entscheidung durch den Entwickler in Form einer Konfigurationseinstellung für alle Komponenten einer Anwendung getroffen werden kann. Existieren für eine Klasse weder der Quelltext, noch Performancemessungen, wird diese im Performancemodell nicht abgebildet.

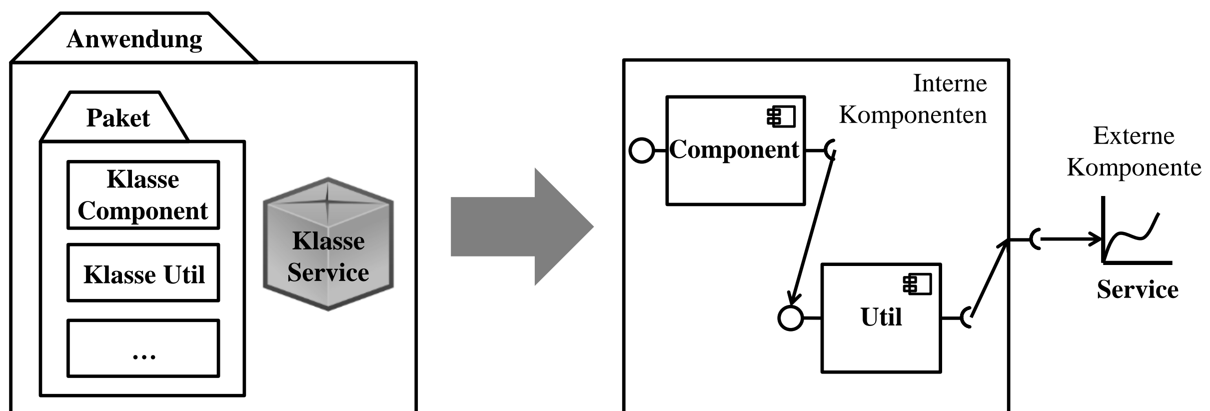


Abbildung 4-5: ***Beispiel für die Abbildung und Abgrenzung von internen und externen Komponenten***
Quelle: Eigene Darstellung

Ein Beispiel für die Abbildung von Klassen als interne oder externe Komponenten ist in Abbildung 4-5 dargestellt. Anwendungen können neben Quelltext auch referenzierte Bibliotheken enthalten. Bibliotheken können in Form von Java-Archiven vorkommen und bestehen aus kompilierten Klassen. Für die Klassen *Component* und *Util* liegt der Quelltext vor. Diese werden als interne Komponenten abgebildet. *Component* verwendet *Service* nicht direkt, sondern über die Klasse *Util*. Die Klasse *Service* wird als externe Komponente abgebildet, da hierfür nur Antwortzeitmessungen vorliegen.

4.2.3 Abbildung des Kontrollflusses

Bei der Abbildung der Komponentenstruktur wird die Existenz von Beziehungen zwischen Komponenten berücksichtigt. Erst mit der Abbildung des Kontrollflusses werden konkrete Dienste, die Komponenten wiederverwenden, spezifiziert. Jede Methode einer Java-Klasse wird als Implementierung eines Dienstes der entsprechenden Komponente untergeordnet. Ein Beispiel für die Abbildung des Kontrollflusses einer Klasse ist in Abbildung 4-6 dargestellt. Die Klasse *Util* besitzt die Methode *method*. In der existierenden Komponentendefinition für *Util* wird die Implementierung des Dienstes *method* angelegt. Dieser Dienst verwendet wiederum den Dienst *execute* der Komponente *Service*. Ein Kontrollfluss besitzt immer einen Start- und einen Endknoten. Für jeden Aufruf des Dienstes *method* wird der Kontrollfluss einmal ausgeführt.

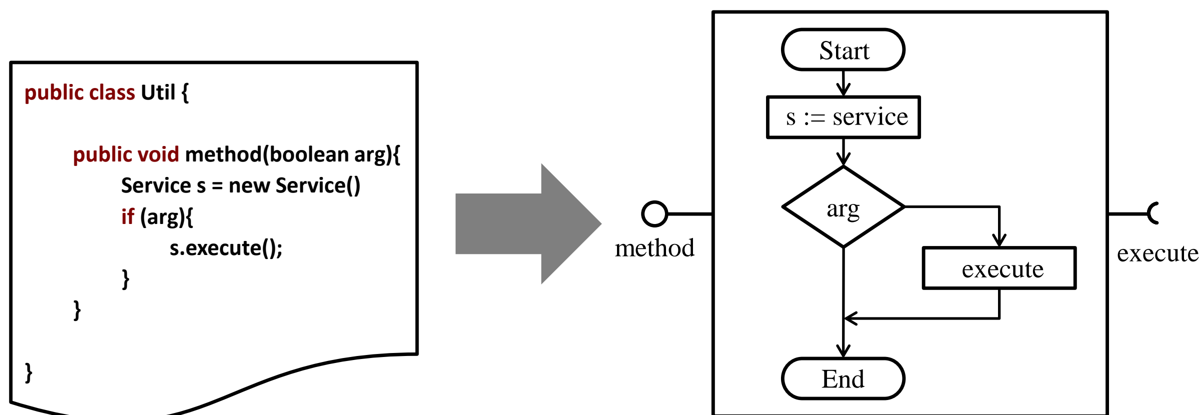


Abbildung 4-6: Beispiel für die Abbildung des Kontrollflusses von Komponenten
Quelle: Eigene Darstellung

Bei der Verarbeitung des Quelltextes einer Methode werden Methodenaufrufe, Kontrollstrukturen und sonstige Anweisungen unterschieden. Die Behandlung dieser Anweisungen wird im Folgenden beschrieben.

Methodenaufrufe

Über die Verarbeitung von Methodenaufrufen wird die Wiederverwendung von Diensten identifiziert. Methoden werden im Zusammenhang mit einer Objektreferenz aufgerufen. Diese Referenz hat als Typ eine Klasse oder eine Schnittstelle. Statische Methoden können ohne Objektreferenz direkt auf einer Klasse angewandt werden. Aufgrund von Polymorphismus und von Referenzen auf Java-Schnittstellen, kann die Ausprägung des entsprechenden Objektes zur Laufzeit nicht immer eindeutig bestimmt werden. Bei der Abbildung des Kontrollflusses wird der Typ der Objektreferenz aus dem Quelltext berücksichtigt. Nach der Abbildung der Komponentenstruktur existiert für diesen Typ eine entsprechende Schnittstelle. Im Kontrollfluss wird der Methodenaufruf als Verwendung dieser Schnittstelle dargestellt. Die Existenz von mehreren Komponenten, welche diese Schnittstelle anbieten, stellt eine Ausnahme dar und wird separat behandelt (siehe Abschnitt 4.2.7).

Im Quelltext können auch Methoden derselben Klasse aufgerufen werden. Abgebildet wird dies dadurch, dass die resultierende Komponente eigene Dienste wiederverwendet. Um dies zu

ermöglichen werden auch private Methoden als öffentliche Dienste von der Komponente angeboten.

Kontrollstrukturen

Mit den wiederverwendeten Diensten werden die Haupttreiber der Gesamtantwortzeit im Performancemodell berücksichtigt. Kontrollstrukturen, wie z.B. If-/Switch-Anweisungen oder For-/While-Schleifen beeinflussen ob und wie oft Dienste im Kontrollfluss aufgerufen werden. Das Ergebnis der Kontrollstrukturen hängt wiederum von Parametern ab und kann nur zur Laufzeit bestimmt werden. Dadurch ist der Ablauf des Kontrollflusses probabilistisch. Folgende Annahmen werden für das Verhalten der Kontrollstrukturen getroffen:

- If-Anweisung: Für die zwei möglichen Ergebnisse wird eine Wahrscheinlichkeit von jeweils 50% angenommen.
- Switch-Anweisung: Für die variable Anzahl der Ergebnisse wird jeweils eine gleichverteilte Wahrscheinlichkeit angenommen.
- For-/While-Schleife: Für Schleifen wird eine Iteration angenommen.

Sonstige Anweisungen

Die Java-Programmiersprache unterstützt die Verschachtelung und Verkettung von Anweisungen. So kann z.B. auch eine Return-Anweisung einen Methodenaufruf enthalten oder eine Methode auf den Rückgabewert eines vorherigen Methodenaufrufs ausgeführt werden. Für die Identifizierung von relevanten Anweisungen werden auch diese Strukturen adressiert.

4.2.4 Abbildung der Parametrisierung

Die Annahme einer Gleichverteilung der Wahrscheinlichkeit für das Eintreten der Bedingung von If-, If-Else- oder Switch-Anweisungen sowie die einmalige Wiederholung bei Schleifen stellt eine starke Vereinfachung dar. Abhängig von der Parametrisierung einer Komponente zur Laufzeit können aufgrund dieser Vereinfachung gravierende Unterschiede zwischen der beobachteten und der vorhergesagten Antwortzeit auftreten. Um dieser potenziellen Fehlerquelle zu entgegnen unterstützt der Ansatz die Spezifikation von Annahmen über die Parametrisierung von Aufrufen durch den Entwickler. Ein Beispiel hierfür ist in Abbildung 4-7 dargestellt. Der Entwickler bearbeitet die Klasse *Util* und spezifiziert die Annahme, dass der Parameter *list* der Methode *method* durchschnittlich einhundert Elemente enthält. Bei der Erstellung des Kontrollflusses wird dieser Wert für die Konfiguration der Anzahl der Wiederholungen aller Schleifen, die auf diesem Parameter basieren, übernommen.

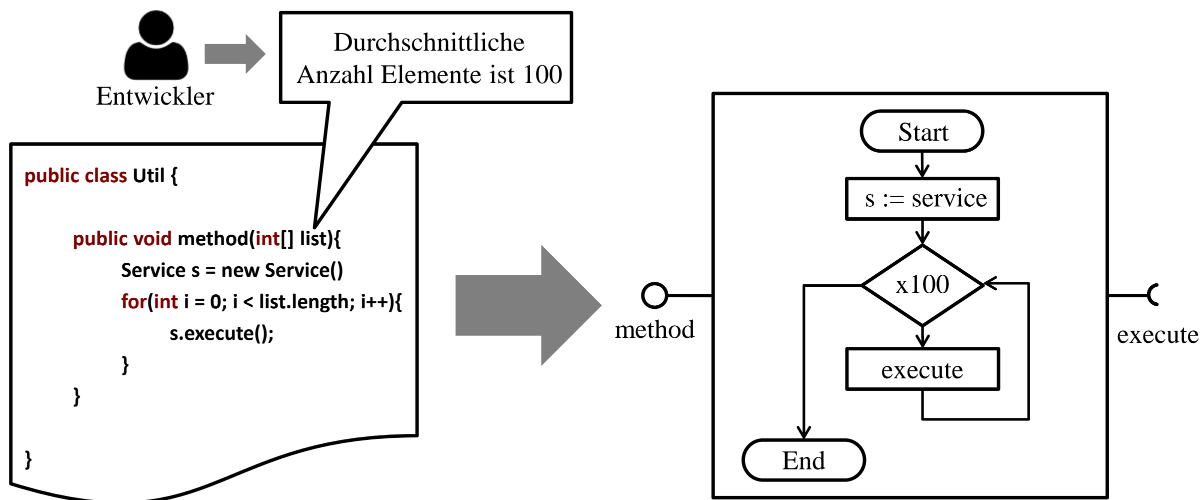


Abbildung 4-7: Beispiel für die Abbildung der Parametrisierung von Dienstaufrufen
Quelle: Eigene Darstellung

Für boolesche Parameter kann die Wahrscheinlichkeit für den Wert *true* spezifiziert werden. Alle If-Anweisungen, welche auf diesem Parameter basieren, werden dementsprechend konfiguriert. Switch-Anweisungen können auf verschiedenen Datentypen basieren und eine beliebige Anzahl an Alternativen aufweisen. Die Wahrscheinlichkeit dieser Alternativen kann durch den Entwickler nicht beeinflusst werden.

4.2.5 Abbildung von Puffermechanismen

Die Implementierung von Komponenten sieht oft die Zwischenspeicherung von Rückgabewerten der wiederverwendeten Dienste vor, wenn diese in einem bestimmten Zeitraum mehrfach gebraucht werden. Daten können sowohl über die ganze Laufzeit einer Anwendung als auch für eine einzelne Nutzersession zwischengespeichert werden. Der Ansatz für die Unterstützung von Performancebewusstsein betrachtet nicht das Verhalten eines Systems über einen Zeitraum, sondern nur die Antwortzeit einzelner Komponenten. Die Existenz von Puffermechanismen auf Ebene der Nutzersession kann daher einen relevanten Einfluss auf die Antwortzeitvorhersagen haben. Die JPA unterstützt beispielsweise die Pufferung von Datenbankobjekten auf Session-Ebene (vgl. Second-Level Cache im Abschnitt 2.5.2). Dieser Mechanismus ist standardmäßig aktiviert und führt dazu, dass unterschiedliche Komponenten auf einen gemeinsamen Puffer zugreifen. Ein Beispiel für die Pufferung von Daten ist in Abbildung 4-8 dargestellt.

Die Komponente *component* verwendet den Dienst *find* der Komponente *BufferedService*, um ein Objekt aus der Datenbank zu laden. Bei dem *BufferedService* handelt es sich um eine externe Komponente, die nur hinsichtlich des Antwortzeitverhaltens abgebildet ist. Der Zugriff auf die Datenbank impliziert eine entsprechende Zeitverzögerung. Die Komponente *Util* wird von *component* aufgerufen und benötigt denselben Datensatz. In der realen Implementierung würde der *BufferedService* feststellen, dass sich der Datensatz bereits im Puffer befindet und diesen zurückliefern. Als Ergebnis würde die wiederholte Ausführung des Dienstes *execute* eine viel geringere Antwortzeit aufweisen. Für eine genauere Vorhersage der Antwortzeit muss dieses Verhalten auch im Performancemodell abgebildet werden.

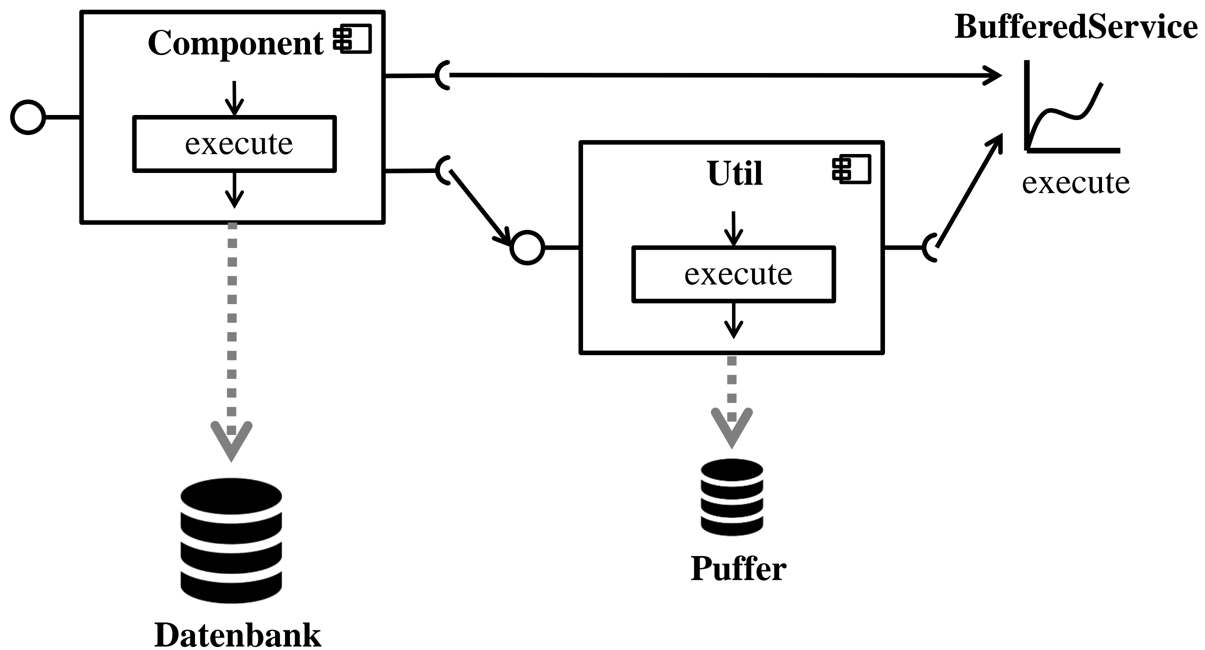


*Abbildung 4-8: **Beispiel für die Pufferung von Daten innerhalb einer Session***
Quelle: Eigene Darstellung

Der Ansatz sieht vor, dass die Existenz von Puffermechanismen bei bestimmten Diensten in der Performancedatenbank spezifiziert werden kann. Bei der Analyse der Gesamtantwortzeit der fokussierten Komponente wird sichergestellt, dass ein Dienst mit Puffermechanismus innerhalb einer Nutzersession maximal ein Mal berücksichtigt wird. Beim wiederholten Aufruf dieses Dienstes bleibt die Gesamtantwortzeit unverändert. Dieses Verfahren berücksichtigt jedoch nicht die Parametrisierung des Aufrufs.

4.2.6 Abbildung des Antwortzeitverhaltens von wiederverwendeten Komponenten

Den Ausgangspunkt für die Abbildung des Antwortzeitverhaltens von wiederverwendeten Diensten bilden Performancemessungen der laufenden Komponenten. Messungen können sowohl in Produktions-, als auch in Testumgebungen durch die Instrumentierung der entsprechenden Java-Klassen durchgeführt werden. Benötigt werden Messungen auf Ebene der Methoden von Klassen. Für jede Methode müssen folgende Informationen erhoben werden:

- Klassenname: Für eine eindeutige Identifizierung muss der voll qualifizierte Name der Klasse protokolliert werden.
- Methodenname und Signatur: Für eine eindeutige Identifizierung der Methode muss die vollständige Signatur hinsichtlich Anzahl, Reihenfolge und Typ der Parameter dokumentiert werden. Die Namen der Parameter sind nicht relevant.
- Ausführungszeit bzw. Ein- und Austrittszeitpunkt: Für jede Methode muss die Dauer zwischen der Ausführung der ersten Anweisung und dem Abschluss der letzten Anweisung erhoben werden.

Die resultierenden Messungen werden in die Performancedatenbank geladen und weiter zu einer aggregierten Darstellung des Antwortzeitverhaltens eines Dienstes verarbeitet. Die verschiedenen Alternativen zur Abbildung der Antwortzeitmessungen werden im Folgenden beschrieben.

Maße der zentralen Tendenz

Mehrere Beobachtungen könne mit Hilfe der Maße der zentralen Tendenz zu einem einzelnen Wert verdichtet werden (Lilja 2000, S. 26). Tabelle 4-3 bietet einen Überblick über diese Mittelwerte.

Maß	Beschreibung	Vorteile	Nachteile
Arithmetisches Mittel	Beschreibt die Summe aller Antwortzeiten, geteilt durch die Anzahl der Beobachtungen.	Berücksichtigt alle Werte aus der Stichprobe.	Empfindlich für Ausreißer.
Median	Beschreibt den mittleren Wert der Liste von aufsteigend sortierten Antwortzeiten.	Weniger empfindlich für Ausreißer.	Berücksichtigt nicht alle Werte aus der Stichprobe.
Modus	Beschreibt den Wert aus der Menge der erhobenen Antwortzeiten, der am häufigsten vorkommt.	Geeignet für nominale Werte, welche nur gezählt werden dürfen. Weniger empfindlich für Ausreißer.	Der Modus kann nur dann bestimmt werden, wenn auch ein Wert häufiger vorkommt als alle anderen.

Tabelle 4-3: ***Maße der zentralen Tendenz***
Quelle: In Anlehnung an Lilja (2000, S. 26-29)

Da es sich bei Antwortzeiten nicht um nominale Werte handelt, und jede Ausprägung potenziell nur einmalig vorkommen könnte, ist der Modus in diesem Fall als Maß der zentralen Tendenz ungeeignet. Das arithmetische Mittel sollte dann eingesetzt werde, wenn alle Werte der Stichprobe berücksichtigt werden sollen (Jain 1991). Bei der Messung der Performance von Unternehmensanwendungen können aus verschiedenen Gründen oszillierende Werte beobachtet werden. Einerseits kommt es während der Einschwingphase zu erhöhten Antwortzeiten aufgrund von Initialisierungen auf Hard- und Softwareebene. Nach dieser Phase können unterschiedliche Systemfehler, Nutzerinteraktionen oder Schwankungen im Workload auftreten, die zu Ausreißern führen (Lilja 2000, S. 45). Wenn durch die Systematik der Messung solche Effekte ausgeschlossen werden, kann das arithmetische Mittel für die Verdichtung der Messwerte angewendet werden. Werden Messungen auf unterschiedlichen Systemen mit stark schwankenden Workloads und während der Einschwingphase erhoben, ist der Median eher geeignet.

Quantile und Perzentile werden ähnlich zum Median berechnet, sie bezeichnen den Wert an einer bestimmten Position einer sortierten Menge. Das 95-Perzentil einer Stichprobe entspricht einem Schwellwert, der beschreibt, dass 95% aller Messungen kleiner oder gleich diesem Wert sind. Abhängig vom gewählten Quantil oder Perzentil wird eine optimistischere oder pessimistischere Interpretation der Stichprobe vorgenommen.

Wahrscheinlichkeitsfunktionen

Wahrscheinlichkeits- (engl. performance mass function) und Wahrscheinlichkeitsdichtefunktionen (engl. performance density function) eignen sich für die Beschreibung der Wahrscheinlichkeit mit der ein Wert oder ein Wertebereich auftreten kann (Becker et al. 2009). Mit Hilfe dieser Funktionen kann das Antwortzeitverhalten eines Dienstes detaillierter abgebildet werden. Anstatt einen konstanten Wert zu verarbeiten werden unterschiedlich Werte abhängig von deren Wahrscheinlichkeit herangezogen. Dies impliziert, dass für die Berechnung der Gesamtantwortzeit mehrere Wiederholungen durchgeführt werden müssen, damit auch Schwankungen bemerkbar werden.

Performancekurven

Performancekurven beschreiben die Performance eines Dienstes abhängig von seiner Konfiguration oder seiner Auslastung (Wert et al. 2012). Mit diesem Konzept können sowohl das Antwortzeitverhalten als auch der Durchsatz und die Ressourcenauslastung abgebildet werden. Die Konfiguration und die Auslastung stellen Inputparameter dar und werden entweder als numerische oder boolesche Werte ausgedrückt (Wert et al. 2012). Mit Hilfe von Performancekurven kann das Antwortzeitverhalten eines Dienstes z.B. abhängig von der Anzahl der parallel bearbeiteten Anfragen dargestellt werden. Performancekurven können entweder als Tabelle oder Funktion spezifiziert werden. Über Tabellen kann eine Menge an Kombinationen von Inputparametern und Performancemetriken spezifiziert werden. Funktionen dienen als Formel für die Berechnung von Metriken.

Der hier beschriebene Ansatz unterstütz die Verarbeitung der Maße der zentralen Tendenz, von Wahrscheinlichkeitsfunktionen und Performancekurven. Um die Wahrscheinlichkeit für unterschiedliche Antwortzeiten von Diensten abzubilden wird die Vorhersage basierend auf mehreren Wiederholungen getroffen.

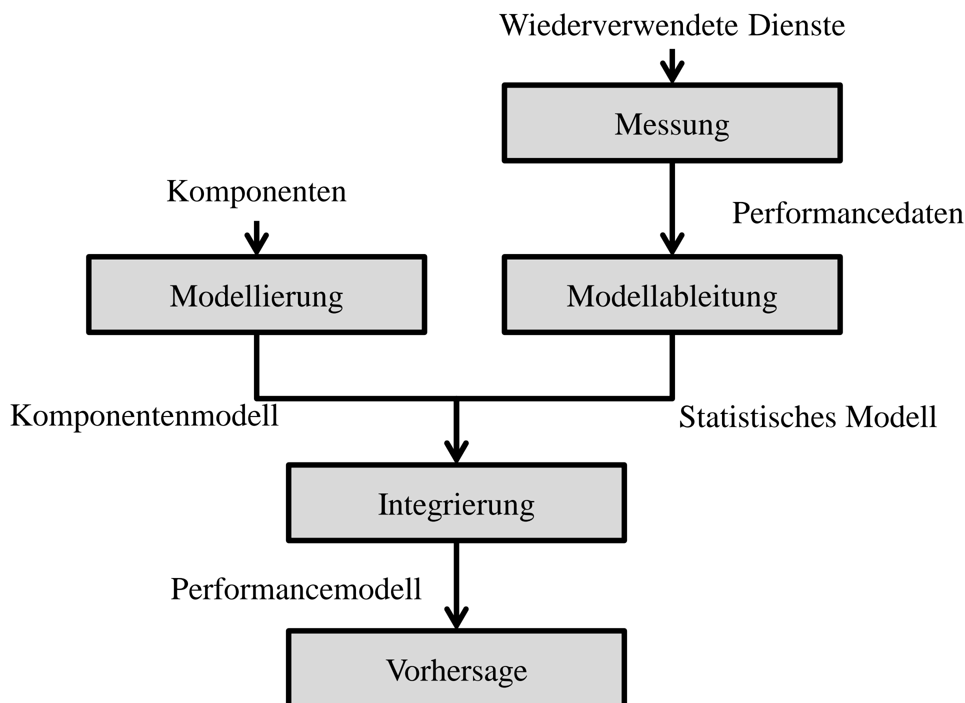


Abbildung 4-9: Integration von modell- und messbasierten Verfahren für die Performanceevaluierung
Quelle: In Anlehnung an Westermann et al. (2011)

Das Vorgehen zur Erhebung, Verdichtung und Integration der Messwerte ist in Abbildung 4-9 dargestellt. Durch die Instrumentierung der wiederverwendeten Dienste werden Antwortzeitmessungen erhoben. Mit Hilfe von statistischen Methoden werden Messungen zu einem statistischen Modell verdichtet. Parallel dazu werden die statische Struktur und der Kontrollfluss der Komponenten modelliert (siehe Abschnitte 4.2.1 und 4.2.3). Das Modell des Antwortzeitverhaltens wird mit dem Komponentenmodell anschließend integriert. Für jeden Aufruf eines wiederverwendeten Dienstes wird das berechnete Antwortzeitverhalten hinterlegt. Über Simulation wird dann die Gesamtantwortzeit der folussierten Komponente evaluiert.

4.2.7 Abbildung von mehrfachen Implementierungen

Sowohl die Implementierung in Java als auch die Abbildung von Komponenten in einem Performancemodell unterstützt die Existenz von mehrfachen Implementierungen derselben Schnittstelle. In der Java-Programmiersprache kann eine Schnittstelle von beliebig vielen Klassen implementiert werden. Erst zur Laufzeit muss eine Beziehung zu einer konkreten Implementierung vorliegen. Die Java EE sieht mit der Context and Dependency Injection (CDI) (DeMichiel/Shannon 2013, S. 149) die Referenzierung von Komponenten über Schnittstellen vor. Die Abhängigkeit zu einer konkreten Implementierung wird durch CDI zur Laufzeit hergestellt. Im Gegensatz zur Resource Injection, bei der wiederverwendete Beans anhand des Namens aufgelöst werden, basiert die DI bei Java EE auf der Auflösung von Typen. Für die Bereitstellung mehrfacher Implementierungen eines Bean-Typs können Qualifier definiert werden. Mit Hilfe der Annotation @*Qualifier* können neue dedizierte Typen von Annotationen definiert werden. Diese Annotationen können dann als Hinweis auf die Bean-Implementierung, welche eingebunden werden soll, angewendet werden.

Für die Vorhersage der Antwortzeit einer Komponente müssen die Abhängigkeiten zu anderen Komponenten eindeutig sein. Die Existenz von mehrfachen Implementierungen für eine Schnittstelle führt zu einem inkonsistenten Zustand des Performancemodells. Dieser Zustand kann sowohl bei internen als auch bei externen Komponenten auftreten.

Interne Komponente

Existieren zwei oder mehrere interne Komponenten, welche dieselbe Schnittstelle implementieren, ist der Kontrollfluss nicht eindeutig und das Performancemodell kann nicht analysiert bzw. simuliert werden. Entsprechende Parametrisierungen aus der Laufzeitumgebung einer Java-EE-Anwendung sind im Performancemodell nicht vorhanden und können auch nicht abgeleitet werden. Der Ansatz wählt in diesem Fall eine beliebige Komponente aus und blendet die restlichen aus.

Externe Komponente

Interne Komponenten können Schnittstellen benötigen, die von mehreren externen Komponenten implementiert werden. Abbildung 4-10 stellt beispielhaft dar, wie die Komponente *Util* den Dienst *execute* der Schnittstelle *ServiceInterface* aufruft. In der Performancedatenbank existieren Antwortzeitmessungen für zwei unterschiedliche Implementierungen dieser Schnittstelle.

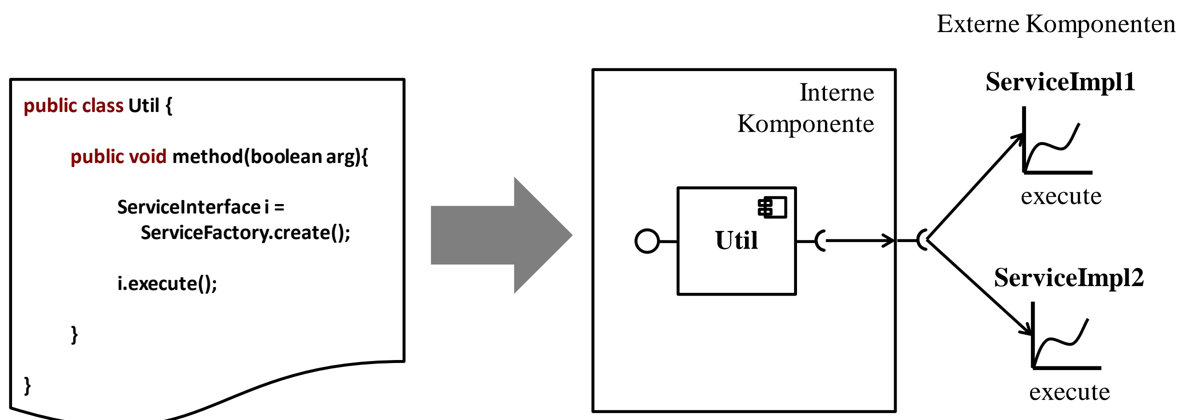


Abbildung 4-10: Beispiel für die mehrfache Implementierung einer Schnittstelle durch externe Komponenten
Quelle: Eigene Darstellung

Komponenten können auch dann als extern betrachtet werden, wenn der Quelltext verfügbar ist (siehe auch Abschnitt 4.2.2). In diesem Fall ist die Existenz der Implementierungen bekannt. Ist der Quelltext nicht verfügbar, können Implementierungen einer Schnittstelle nur anhand der Instrumentierung der Java-EE-Komponenten in der Produktions- oder Testumgebung identifiziert werden. Hierfür sieht der Ansatz die Protokollierung der Quelle des Aufrufs von Methoden vor (Laddad 2009, S. 57). Dadurch wird nicht nur die Ausführung einer Methode, sondern auch der Aufruf, der dazu geführt hat, erfasst.

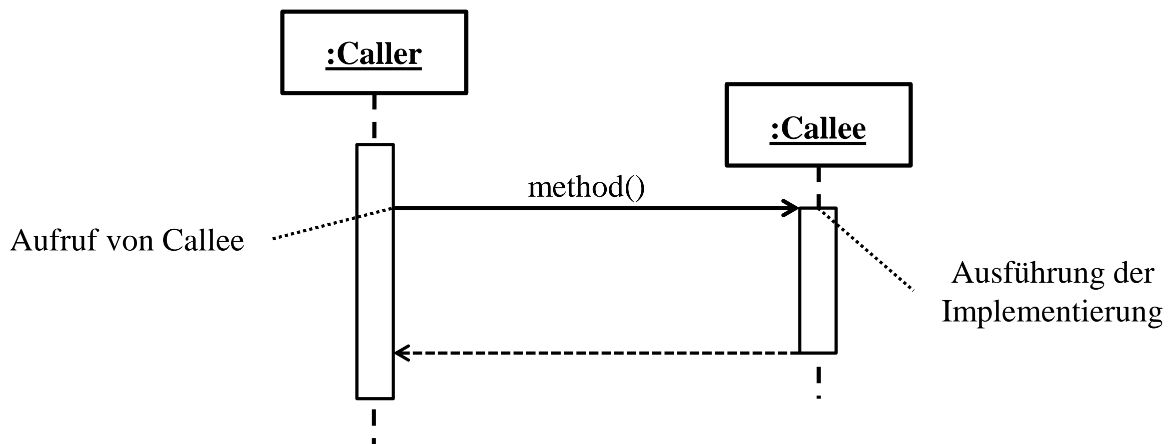


Abbildung 4-11: Beispiel für Protokollierung des Aufrufs und der Ausführung einer Methode
Quelle: In Anlehnung an Laddad (2009, S. 89)

Der Unterschied zwischen der Quelle eines Aufrufs und der Ausführung einer Methode ist in Abbildung 4-11 beispielhaft dargestellt. Die Klasse *Caller* ruft hier eine Methode der Klasse *Callee* auf. Der Aufruf kann sich direkt auf die Klasse *Callee* oder auf eine implementierte Schnittstelle beziehen. Durch die Protokollierung des Aufrufs kann eine Zuordnung von Implementierungen zu Schnittstellen hergestellt werden. Mit dieser zusätzlichen Information ist die Performancedatenbank in der Lage für eine Schnittstelle ein entsprechendes Antwortzeitverhalten zurückzumelden. Der Ansatz sieht vor, dass die Antwortzeit der Implementierung mit der schlechtesten Performance ausgewählt wird.

Existieren mehrere Implementierungen einer Schnittstelle, die jedoch im Quelltext eindeutig referenziert werden, treten die hier beschriebenen Herausforderungen nicht auf.

4.2.8 Beieinflussug des Entwicklers mit Hilfe von Performancebewusstsein

Das Ergebnis der Antwortzeitvorhersage wird dem Entwickler innerhalb des Quelltexteditors, mit dem die fokussierte Komponente dargestellt oder bearbeitet wird, zurückgemeldet (siehe Abbildung 4-12).

```
Component.java                                X
public class Component {

    public void callUtil(){
        Util util = new Util();
        util.method();
    }

}
```

Abbildung 4-12: Rückmeldung der Antwortzeitvorhersage zum Entwickler
Quelle: In Anlehnung an Danciu et al. (2014)

Der Ansatz führt für alle Methoden der angezeigten Klasse eine Vorhersage der Antwortzeit durch. Überschreitet das Ergebnis der Vorhersage den konfigurierten Schwellenwert, wird eine Meldung gegenüber der Methodendeklaration angezeigt. Bei Überschreitung des ersten Schwellenwerts wird eine Warnung in Form eines gelben Tachometers angezeigt. Wird der zweite Schwellenwert überschritten, wird eine Fehlermeldung in Form eines roten Tachometers angezeigt. Einzelne Aufrufe, deren individuelle Antwortzeit die Schwellenwerte überschreiten, werden gelb bzw. rot markiert.

4.3 Technische Umsetzung des Ansatzes

Dieser Abschnitt beschreibt die technische Umsetzung des Konzeptes für die Unterstützung von Performancebewusstsein in Java-EE-Entwicklungsumgebungen. Die Werkzeuge und Aktivitäten des Ansatzes sind in Abbildung 4-13 dargestellt.

Die Java EE Integrated Development Environment (IDE) stellt den Ausgangspunkt des Ansatzes dar. Die IDE basiert auf der Eclipse IDE für Java-EE-Entwickler[18], welche wiederum auf der Eclipse Plattform basiert. Die Eclipse Plattform bietet Basisdienste für Entwicklungswerkzeuge, welche als Plugin implementiert sind (Gallardo et al. 2003, S. 7). Durch die Erweiterung der Plattform um einen spezifischen Satz von Plugins, entsteht eine IDE für Java-EE-Anwendungen. Der hier beschriebene Ansatz stellt auch ein Plugin für die Eclipse IDE dar und erweitert sie um die Unterstützung von Performancebewusstsein.

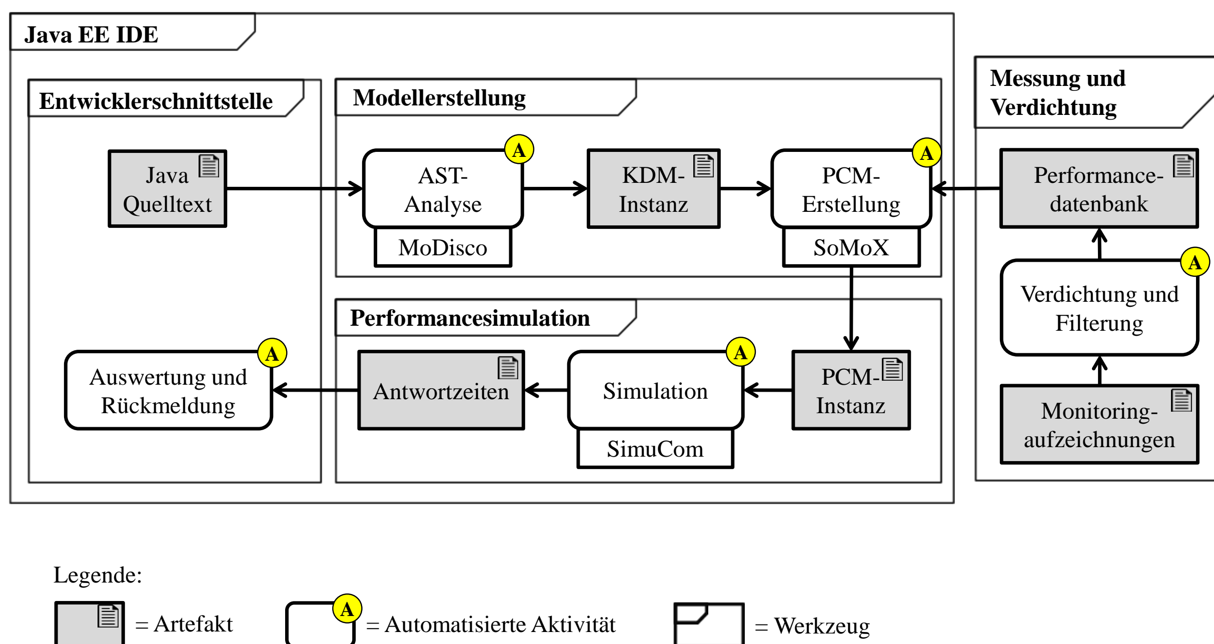


Abbildung 4-13: Ansatz für die Unterstützung von Performancebewusstsein
Quelle: In Anlehnung an Danciu et al. (2015b)

Die existierende IDE wird um drei neue Werkzeuge erweitert. Über die Entwicklerschnittstelle wird der Kontext des Entwicklers adressiert (siehe Abschnitt 4.3.4). Hierüber können Entwickler für eine bestimmte Komponente eine Antwortzeitvorhersage anfordern. Die

[18] https://eclipse.org/

Entwicklerschnittstelle liefert Informationen über den Quelltext bzw. die Komponente, die aktuell vom Entwickler fokussiert wird und zeigt das Ergebnis der Antwortzeitvorhersage an. Während der Modellerstellung wird ausgehend von dem fokussierten Quelltext zuerst mit Hilfe von MoDisco[19] ein abstrakter Syntaxbaum (engl. abstract syntax tree, AST) und anschließend mit Hilfe von SoMoX[20] ein Performancemodell auf Basis von PCM erstellt (siehe Abschnitt 4.3.1). Während der Performancesimulation wird die PCM-Instanz mit Hilfe von SimuCom[21] ausgeführt und die Antwortzeit der fokussierten Komponenten vorhergesagt (siehe Abschnitt 4.3.3).

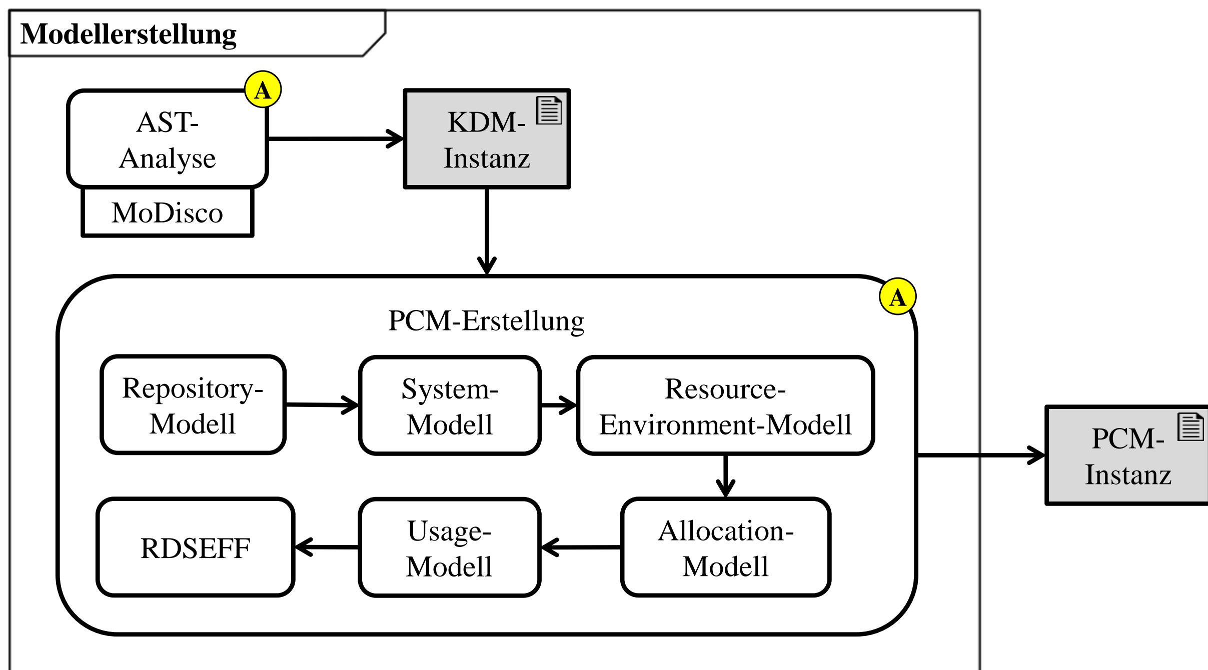
Die Infrastruktur zur Erhebung und Verdichtung von Performancemessungen dient als zentrale Ablage von Antwortzeitmessungen (siehe Abschnitt 4.3.2). Monitoringaufzeichnungen werden durch die Instrumentierung von Anwendungen in der Produktions- oder Testumgebung erhoben. Messungen werden zu einzelnen Werten oder Funktionen verdichtet und der IDE zur Verfügung gestellt. Während der Modellerstellung wird das Antwortzeitverhalten wiederverwendeter Dienste im Performancemodell integriert.

4.3.1 Erstellung des Performancemodells

Zur Modellerstellung werden die Werkzeuge MoDisco und SoMoX nacheinander als Hintergrundprozesse der IDE ausgeführt. Da es sich hierbei um eine rechenintensive Aktivität handelt, würde eine Verarbeitung im Dialogmodus den Arbeitsfluss des Entwicklers stören. Ein zentraler Hintergrundprozess übernimmt die Orchestrierung der zwei Werkzeuge und startet am Ende der Modellerstellung die Performancesimulation.

Die Schritte der Modellerstellung sind in Abbildung 4-14 dargestellt. MoDisco erhält als Eingabe den Pfad eines Java-Projekts und legt die Ergebnisse der Verarbeitung in Form von Dateien in diesem Projekt ab. Sobald die Ergebnisse vorliegen, wird SoMoX gestartet. Die KDM-Instanz wird eingelesen und während der PCM-Erstellung im Hauptspeicher vorgehalten. SoMoX erstellt nacheinander die PCM-Teilmodelle. Das Repository-Modell stellt die Grundlage der PCM-Instanz dar und wird als erstes erstellt. Andere Modelle, wie z.B. das System- und das Usage-Modell referenzieren anschließend Komponenten des Repository-Modells.

[19] https://eclipse.org/MoDisco/
[20] https://sdqweb.ipd.kit.edu/wiki/SoMoX
[21] https://sdqweb.ipd.kit.edu/wiki/Palladio_Solvers_and_Simulation



Legende:

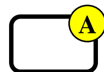
= Artefakt  = Automatisierte Aktivität = Werkzeug

Abbildung 4-14: Ansatz für die Erstellung des Performancemodells
Quelle: In Anlehnung an Danciu et al. (2015b)

Die Schritte der Modellerstellung werden im Folgenden detailliert beschrieben. Listing 4-1 zeigt den vereinfachten Quelltext zweier Java-EE-Komponenten und dient als Beispiel für die Eingabe in die Modellerstellung. Die folgenden Abschnitte stellen Modelle dieser Komponenten dar.

```
1  //Klasse MyBean.java
2  public class MyBean {
3
4      @Inject
5      private DataService dataService;
6
7      public void loadData(List<String> dataIdList){
8             for(String id : dataIdList){
9                    dataService.getData(id);
10            }
11     }
12
13     public void createData(String dataId, boolean doStore){
14            Data data = new Data(dataId);
15            if(doStore){
16                   dataService.storeData(data);
```

```
17          }
18     }
19 }
20
21 //Interface DataService.java
22 public interface DataService{
23     public Data getData(String dataId);
24     public void storeData(Data data);
25 }
26
27 //Klasse DefaultDataService.java
28 public class DefaultDataService implements DataService {
29
30     @PersistenceContext
31     private EntityManager entityManager;
32
33     public Data getData(String dataId){
34          return entityManager.find(Data.class, dataId);
35     }
36
37     public void storeData(Data data) {
38          entityManager.persist(data);
39     }
40 }
```

Listing 4-1: ***Vereinfachtes Beispiel für Komponenten einer Java-EE-Anwendung***
Quelle: Eigene Darstellung

In dem beispielhaften Szenario stellt die Klasse *MyBean* die fokussierte Komponente dar. *MyBean* verwendet die Schnittstelle *DataService*, die von *DefaultDataService* implementiert wird. Die Methode *loadData* erhält eine Liste von IDs als Eingabe und soll die entsprechenden Datensätze mit Hilfe des *DataService* abrufen. Die Methode *createData* erzeugt einen neuen Datensatz und speichert diesen ggf. mit Hilfe des *DataService*. *DefaultDataService* verwendet den *EntityManager* für den Zugriff auf die Datenbank. Die Methode *getData* erhält die ID eines Datensatzes als Eingabe und ruft diesen von der Datenbank ab. Die Methode *storeData* speichert einen Datensatz ab. Für *MyBean* und *DefaultDataService* liegt der Quelltext vor. Der *EntityManager* stellt eine externe Komponente dar.

4.3.1.1 Technische Grundlage der Implementierung

Die Modellerstellung basiert auf dem SOftware MOdel eXtractor (SoMoX), einem Werkzeug für das Reverse Engineering von Komponentenmodellen aus Java-Quelltext (Becker et al. 2010). SoMoX erhält als Eingabe einen AST und erstellt ausgehend davon eine PCM-Instanz, bestehend aus den Repository-, System-, Allocation-, Resource-Environment- und RDSEFF-Modellen. Ein Usage-Modell wird durch SoMoX nicht automatisch erstellt. In einem ersten Schritt erstellt SoMoX eine Menge von Komponentenkandidaten und berechnet für diese unterschiedliche Softwaremetriken (Becker et al. 2010). Ein Clustering-Algorithmus fasst die

Kandidaten basierend auf den Metriken zusammen. Für die Erstellung der AST-Darstellungen kann SoMoX auf eines der folgenden Werkzeuge zurückgreifen (Becker et al. 2010; Leonhardt et al. 2015; Klatt et al. 2013):

- Structural Investigation of Software Systems (SISSy)[22]: Mit Hilfe von SISSy können Generalised Abstract Syntax Trees (GAST) aus Quelltext abgeleitet werden.
- MoDisco: Mit Hilfe von MoDisco können Instanzen des Knowledge Discovery Metamodel (KDM) aus Quelltext extrahiert werden.
- Java Model Parser and Printer (JaMoPP)[23]: JaMoPP definiert ein eigenes Metamodell für die Abbildung von Java-Programmelementen.

Der hier beschriebene Ansatz setzt MoDisco für das Einlesen von Java-Quelltext und die Erstellung einer AST-Darstellung ein. MoDisco bietet einen generischen und erweiterbaren Ansatz für das Reverse Engineering eines abstrakten Modells aus Java-Quelltext (Bruneliere et al. 2010). Sowohl SoMoX als auch MoDisco sind als Eclipse Plugin implementiert und werden in der IDE integriert.

SoMoX zielt auf das Reverse Engineering komplexer Anwendungen ab und versucht dabei Details der Implementierung zu abstrahieren. Beispielsweise werden Klassen ohne Geschäftslogik oder bestimmte Java-Elemente ignoriert. Aufgrund der Anforderung an einen möglichst hohen Detailgrad des Performancemodells wurde die SoMoX-Implementierung erweitert.

Der SoMoX-Ansatz wurde auch um eine Betrachtung des dynamischen Verhaltens von Anwendungen erweitert (Krogmann et al. 2010). Durch den Einsatz mathematischer Modelle wird die Abhängigkeit der Performance von der Komponentenparametrisierung zur Laufzeit untersucht. Diese Aspekte werden durch den Ansatz für Performancebewusstsein nicht adressiert.

SoMoX stellt eine Weiterentwicklung der Ansätze ArchiRec (Chouambe et al. 2008) und Java2PCM dar (Kappler et al. 2008). Im Gegensatz zu den Vorgängern, unterstützt SoMoX das Reverse Engineering verschiedener Programmiersprachen und integriert die Kernfunktionen der zwei Ansätze (Becker et al. 2010).

4.3.1.2 Einlesen des Java-Quelltextes

MoDisco wird zum Einlesen des Quelltextes über ein Hintergrundprozess der IDE gestartet. Als Eingabeparameter erhält das Werkzeug eine Referenz auf das Java-Projekt, zu dem die fokussierte Komponente gehört. Bestimmte Java-Pakete, sowohl aus dem Quelltext als auch aus Bibliotheken, können aus der Analyse ausgeschlossen werden. Als Ergebnis tauchen die betroffenen Java-Klassen in der KDM-Instanz nicht auf. Der Ansatz schließt nur wenige Pakete, die für Unit-Tests eingesetzt werden, aus. Alle übrigen Pakete werden zur Laufzeit der Anwendung benutzt, können somit einen Einfluss auf die Performance haben und sollten nicht ausgeschlossen werden. Als Ergebnis liefert das Werkzeug eine KDM-Instanz in Form mehrerer Dateien zurück. Diese besteht aus den folgenden Untermodellen:

[22] http://sissy.fzi.de
[23] http://www.jamopp.org/index.php/JaMoPP

- Java-Modell: Dieses Modell stellt alle Konstrukte der Java-Spezifikation aus dem eingelesenen Quelltext abstrakt dar.
- KDM-Modell: Dieses Modell stellt die Artefakte der Java-Anwendung, wie z.B. Dateien und Ordner, abstrakt dar.
- Java-KDM-Modell: Dieses Modell vereint das Java- und das KDM-Modell und dient als Input für SoMoX.

MoDisco erstellt die KDM-Instanz mit Hilfe von EMF. Für die Verarbeitung der Inhalte wird das Java-KDM-Modell eingelesen. Die Elemente der KDM-Instanz stehen zueinander in Beziehung. Über die Referenz auf die Abstraktion einer Klasse kann beispielsweise zu allen enthaltenen Methoden und Attribute navigiert werden.

4.3.1.3 Erstellung des Repository-Modells

Zur Erstellung des Repository-Modells liest SoMoX zuerst die KDM-Instanz ein und iteriert über alle enthaltenen Elemente. Für die folgenden Java-Elemente wird jeweils ein Komponentenkandidat angelegt:

- Klassendeklaration: Spezifiziert die Implementierung einer Klasse. Innere Klassendeklarationen werden dabei ignoriert.
- Enum-Deklaration: Der Enum-Datentyp stellt eine vereinfachte Art von Klasse dar und deklariert eine Menge von Werten (Gosling et al. 2013, S. 4).

MoDisco extrahiert Klassen- und Enum-Deklarationen nur aus dem eingelesenen Quelltext, nicht aus den referenzierten Bibliotheken. Somit wird in diesem Schritt sichergestellt, dass Klassen, für die kein Quelltext verfügbar ist, nicht als interne Komponenten klassifiziert werden können (vgl. Abschnitt 4.2.2). Deklarationen von Schnittstellen werden auch aus den Bibliotheken extrahiert, wenn diese im Quelltext referenziert werden.

SoMoX versucht die resultierende Menge der Kandidaten zu reduzieren. Die Standardimplementierung schließt alle Klassen ohne Geschäftslogik, die nur Methoden für das Lesen und Setzen von Attributen anbieten, aus. Für den Kontrollfluss sind jedoch auch diese einfachen Methoden relevant, da auf das zurückgelieferte Attribut auch Methoden ausgeführt werden können. Für die vollständige Abbildung des Kontrollflusses aus dem Quelltext werden diese Kandidaten nicht entfernt. Weiterhin versucht die Standardimplementierung die Kandidaten zusammenzufassen oder zu gruppieren. Damit jede Klasse auf genau eine Komponente abgebildet wird, wurde dieser Schritt entfernt. Die verbleibenden Kandidaten werden als Repository-Komponenten angelegt.

Abschließend werden die Komponenten anhand der Performancedatenbank und der Konfiguration der IDE als interne oder externe Komponenten klassifiziert (vgl. Abschnitt 4.2.2). Die Konfiguration gibt vor, ob Komponenten, für die sowohl Antwortzeitmessungen als auch der Quelltext vorliegen, als interne oder externe Komponenten eingestuft werden sollen. Externe Komponenten werden aus dem Repository entfernt. Das Ausschlussverfahren für

Komponenten, ausgehend vom Java-Projekt bis hin zu der Menge der internen Komponenten, ist in Abbildung 4-15 zusammenfassend dargestellt.

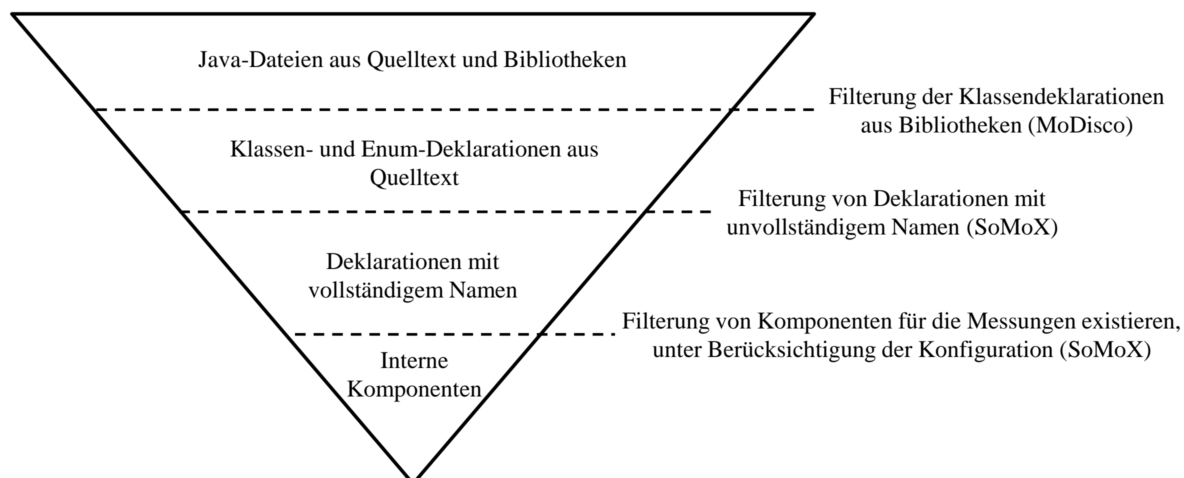


Abbildung 4-15: Ausschlussferfahren für die Identifikation von internen Komponenten
Quelle: Eigene Darstellung

Für jede Repository-Komponente werden benötigte und angebotene Schnittstellen ermittelt. SoMoX iteriert über alle Komponenten und analysiert die entsprechende Java-Klasse. Es werden alle dort referenzierten Klassen und Schnittstellen extrahiert. Für die Menge der extrahierten Klassen werden alle Superklassen und implementierten Schnittstellen ermittelt. Für alle extrahierten Schnittstellen werden wiederum die Superschnittstellen ermittelt. Alle resultierenden Java-Schnittstellen werden als Repository-Schnittstelle angelegt, falls diese noch nicht vorhanden sind. Für jede Repository-Schnittstelle wird eine Beziehung zur Komponente hergestellt. Während der Spezifikation von Repository-Schnittstellen werden auch Signaturen aus den Methoden der Java-Schnittstelle abgeleitet. Die Standardimplementierung von SoMoX entfernt Referenzen von Komponenten auf sich selbst. Damit der Aufruf der eigenen Dienste einer Komponente abgebildet werden kann, wurde dieser Schritt entfernt. Zusätzlich werden auch private oder geschützte Methoden als Signaturen abgebildet.

Als nächstes werden für alle Repository-Komponenten die angebotenen Schnittstellen ermittelt. Falls diese nicht vorhanden sind, werden sie erstellt. Zwischen den Repository-Komponenten und den angebotenen -Schnittstellen wird jeweils eine Beziehung hergestellt. Für jede Repository-Komponente werden leere RDSEFFs für die Implementierung der Signaturen angelegt. Der Kontrollfluss der RDSEFFs wird zu einem späteren Zeitpunkt spezifiziert (siehe Abschnitt 4.3.1.8).

Die SoMoX-Standardimplementierung bildet Java-Schnittstellen, welche nur in Form von Bytecode verfügbar sind, nicht vollständig ab. In diesen Fällen verfügen die erstellten Repository-Schnittstellen nicht über alle notwendigen Signaturen und das resultierende Modell ist inkonsistent. SoMoX wurde dahingehend erweitert, dass die fehlenden Informationen nachträglich extrahiert und im Repository-Modell hinzugefügt werden.

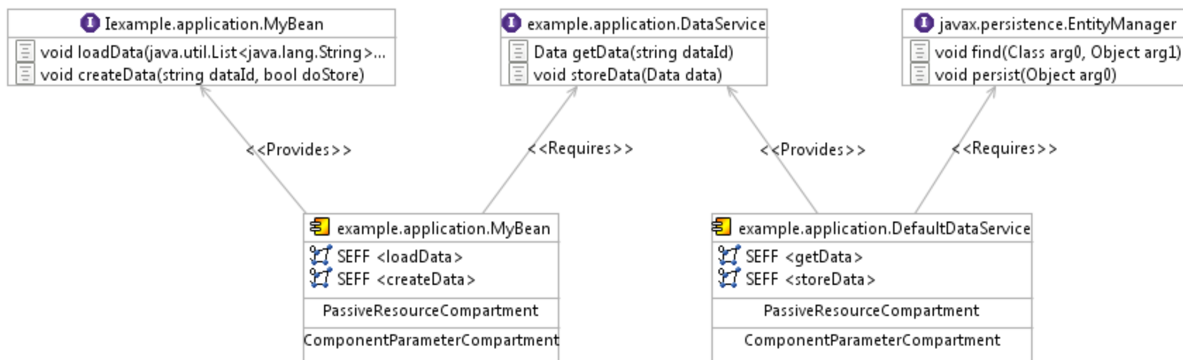


Abbildung 4-16: Vereinfachte Darstellung eines automatisch erstellen Repository-Modells
Quelle: Eigene Darstellung

Ein Repository-Modell der Java-EE-Komponenten aus Listing 4-1 ist in Abbildung 4-16 dargestellt. Die Komponente *MyBean* bietet die Dienste *loadData* und *createData* an. Da *MyBean* keine explizite Schnittstelle implementiert, wird eine implizite Schnittstelle abgeleitet. *MyBean* benutzt die Schnittstelle *DataService*, die von *DefaultDataService* implementiert wird. *Default-DataService* benutzt den *EntityManager* für den Abruf und die Speicherung von Datenbankobjekten. Der *EntityManager* wird aus einer Bibliothek wiederverwendet und stellt eine externe Komponente dar. Aus diesem Grund existiert im Repository-Modell keine Komponentenspezifikation für den *EntityManager*. *MyBean* und *DefaultDataService* referenzieren auch die Klasse *Data*, die implizit auch im Repository-Modell abgebildet wird. Der Einfachheit halber, wird *Data* in Abbildung 4-16 nicht aufgeführt.

4.3.1.4 Erstellung des System-Modells

Während dieser Aktivität werden die Komponenten des Repository-Modells zu einem System verknüpft. SoMoX erstellt für jede Komponente des Repository-Modells einen Assembly-Kontext. Der Kontext beschreibt die Position, die eine Komponente im System einnimmt. Beziehungen zwischen Kontexten werden als Assembly-Konnektoren abgebildet. Für jede benötigte Schnittstelle einer Komponente wird eine passende Implementierung ermittelt und ein entsprechender Konnektor angelegt. Wird eine Schnittstelle von mehreren Komponenten angeboten, resultiert ein inkonsistentes System-Modell. SoMoX wurde dahingehend erweitert, dass die Verknüpfung zu einer beliebigen Implementierung hergestellt wird.

SoMoX sieht vor, dass alle angebotenen Komponentenschnittstellen auch als externe Systemschnittstellen angeboten werden. Abhängig von der Anzahl der Komponenten wird dadurch die Komplexität des System-Modells erhöht. Aufgrund der Zielsetzung, das Antwortzeitverhalten der fokussierten Komponente zu untersuchen, wurde die Implementierung von SoMoX angepasst. Nur Schnittstellen, die von der fokussierten Komponente angeboten werden, werden als Systemschnittstelle abgebildet.

SoMoX sieht auch vor, dass für benötigte Schnittstellen ohne Implementierung eine künstliche Komponentenspezifikation im Repository-Modell angelegt wird. Aufgrund der Zielsetzung, externe Komponenten über ihr Antwortzeitverhalten, anstatt der Implementierung abzubilden, wurde dieses Vorgehen abgeändert. Benötigte Komponentenschnittstellen, für die keine Implementierung vorliegt, werden als externe Systemschnittstelle angefordert. Für benötigte

Systemschnittstellen wird mit Hilfe der Performancedatenbank das Antwortzeitverhalten des entsprechenden Dienstes abgefragt und spezifiziert (siehe Abschnitt 4.3.1.6).

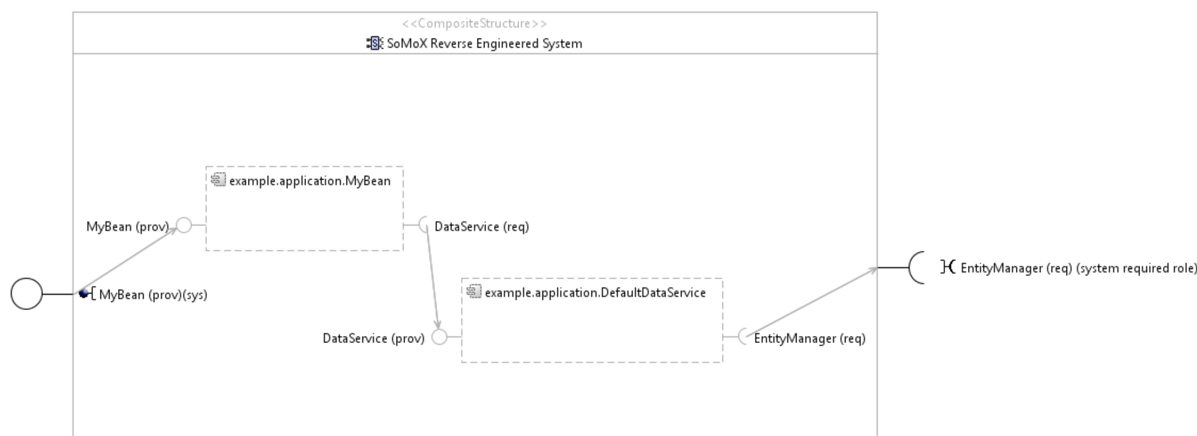


Abbildung 4-17: Vereinfachte Darstellung eines automatisch erstellten System-Modells
Quelle: Eigene Darstellung

Ein System-Modell der Java-EE-Komponenten aus Listing 4-1 ist in Abbildung 4-17 dargestellt. In diesem Beispiel repräsentiert *MyBean* die fokussierte Komponente. Die Schnittstelle dieser Komponente wird als Systemschnittstelle angeboten. Für den *EntityManager* existiert keine Komponentenspezifikation im Repository-Modell. Daher ist die Abhängigkeit zum *EntityManager* als benötigte Systemschnittstelle dargestellt.

4.3.1.5 Erstellung des Resource-Environment- und Allocation-Modells

Das Resource-Environment-Model beschreibt die Umgebung, in der das zuvor spezifizierte System laufen soll. Der hier beschriebene Ansatz für Performancebewusstsein ignoriert jedoch den Einfluss von Ressourcenkonflikten und der Deployment-Plattform auf das Antwortzeitverhalten von Komponenten. Weil die Spezifikation der Komponenten keinen Ressourcenverbrauch aufweist, hat die Ausprägung des Resource-Environment-Modells keinen Einfluss auf das Antwortzeitverhalten der Komponenten. Als Umgebung wird daher eine konstante Minimalkonfiguration angenommen (siehe Abbildung 4-18). Das Resource-Environment-Modell besteht aus einem Server mit einer CPU. Als Verarbeitungsrate der CPU wird der Standardwert *1* konfiguriert.

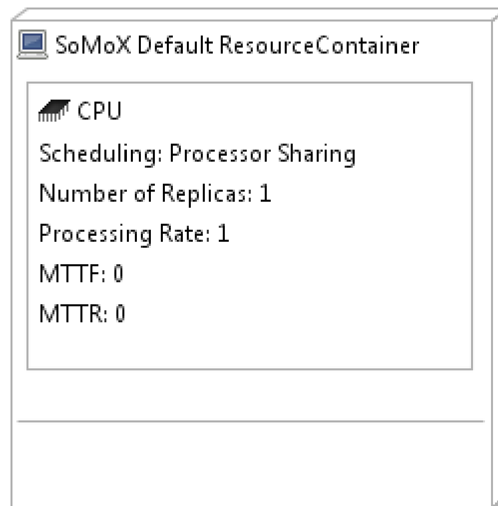


Abbildung 4-18: Vereinfachte Darstellung eines automatisch erstellten Resource-Environment-Modells
Quelle: Eigene Darstellung

Im Allocation-Modell werden die Assembly-Kontexte auf das Ressource-Environment zugeordnet. Aufgrund der vereinfachten Abbildung der Umgebung, werden alle Kontexte ein und derselben Ressource zugeordnet. Ein Allocation-Modell für das Beispiel aus Listing 4-1 ist in Abbildung 4-19 dargestellt.

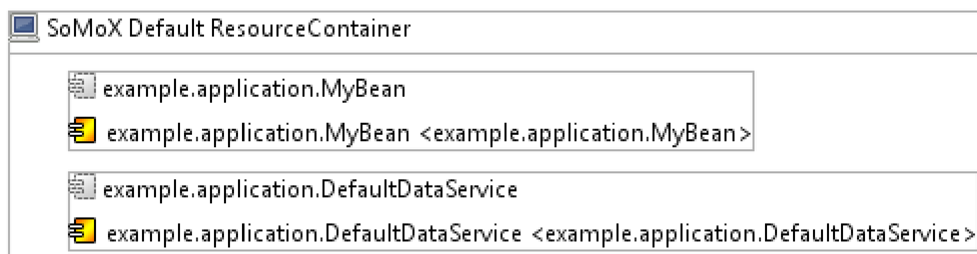


Abbildung 4-19: Vereinfachte Darstellung eines automatisch erstellten Allocation-Modells
Quelle: Eigene Darstellung

4.3.1.6 Spezifizierung des Antwortzeitverhaltens von externen Komponenten

Wiederverwendete Dienste externer Komponenten werden durch benötigte Systemschnittstellen repräsentiert. Das Antwortzeitverhalten von benötigten Systemschnittstellen wird mit Hilfe der Quality of Service Annotation (QoSAnnotation) spezifiziert. QoSAnnotations können die Antwortzeit als s.g. *Stochastic Expression* in Form einer Wahrscheinlichkeitsfunktion oder einer Wahrscheinlichkeitsdichtefunktion darstellen (Koziolek 2008, S. 91f). Sotachastic Expersions können für Werte vom Typ Integer, Double, Boolean sowie für Aufzählungswerte spezifiziert werden. Die Wahrscheinlichkeitsfunktion *IntPMF[(1;0.5), (2;0.5)]* spezifiziert z.B., dass die Werte *1* und *2* mit jeweils einer Wahrscheinlichkeit von 50% auftreten. Stochastic Expersions werden von der Performancedatenbank zur Verfügung gestellt und unverändert in der PCM-Instanz eingetragen.

Der Performance-Curve-Integration-Ansatz (PCI) von Wert et al. (2012) erweitert die QoSAnnotations um die Möglichkeit Performancekurven für Systemschnittstellen zu spezifizieren. Der Ansatz für Performancebewusstsein unterstützt die Spezifikation von Performancekurven als Funktion der parallelen Aufrufe der externen Komponente. Die entsprechende Formel wird in Textform aus der Performancedatenbank abgerufen. Über die interne Variable *Queuelength* verwaltet PCI die Anzahl der parallel bearbeiteten Aufrufe einer externen Komponente und berechnet basierend darauf eine Antwortzeit.

Für die Spezifikation der Antwortzeit als Maß der zentralen Tendenz wird anstatt einer Funktion ein fester Wert als QoSAnnotation übergeben. Die Art wie das Antwortzeitverhalten eines wiederverwendeten Dienstes spezifiziert werden soll, wird von der Performancedatenbank festgelegt. Diese Einstellung kann für jeden Dienst unterschiedlich sein.

4.3.1.7 Erstellung des Usage-Modells

SoMoX wurde um die automatisierte Erstellung eines Usage-Modells erweitert. Zu jeder angebotenen Systemschnittstelle wird ein Aufruf innerhalb einer separaten Branch-Transition erstellt. Die Wahrscheinlichkeit der Transitionen ist gleichverteilt. Damit soll sichergestellt werden, das während einer Simulation das Antwortzeitverhalten aller Dienste der fokussierten Komponente evaluiert wird. Der Ansatz spezifiziert einen geschlossenen Workload für die Abbildung des Verhaltens der simulierten Nutzer. Die Anzahl der Nutzer und ihre Bedenkzeit zwischen zwei Aufrufen der Systemschnittstellen kann durch den Entwickler über die Benutzeroberfläche der IDE konfiguriert werden.

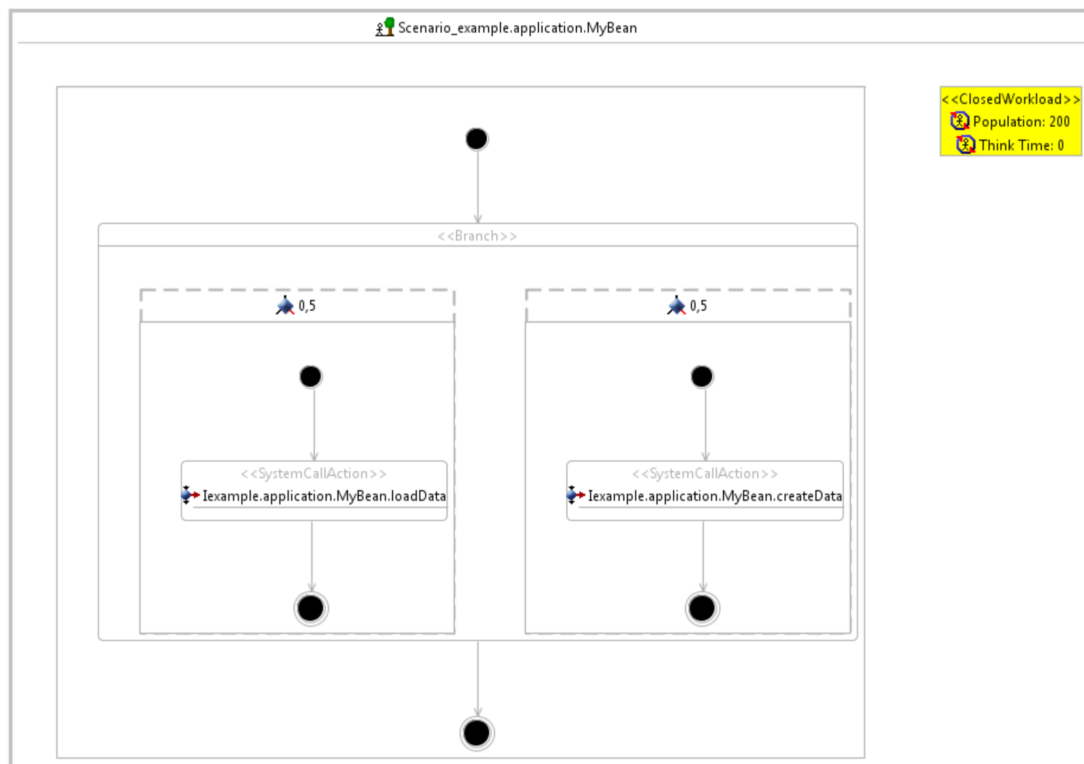


Abbildung 4-20: Vereinfachte Darstellung eines automatisch erstellten Usage-Modells
Quelle: Eigene Darstellung

Ein Usage-Modell für das Beispiel aus Listing 4-1 ist in Abbildung 4-20 dargestellt. Das System bietet die Schnittstellen *laodData* und *createData* der fokussierten Komponenten *MyBean* an. Das Usage-Modell spezifiziert einen geschlossenen Workload mit *200* simulierten Nutzer und einer Bedenkzeit von *0*. Ein Nutzer ruft die Systemschnittstellen mit einer Wahrscheinlichkeit von jeweils 50% auf.

4.3.1.8 Erstellung der RDSEFF

Während der Erstellung des Repository-Modells wurde für jeden Dienst einer Komponente jeweils ein RDSEFF angelegt. Im Rahmen dieser Aktivität werden die RDSEFFs mit Inhalten befüllt. SoMoX analysiert alle Methodendeklarationen der Java-Klassen, die als Repository-Komponente repräsentiert sind. Mit Hilfe des Visitor-Entwurfsmusters (Gamma et al. 1995) werden Anweisungen innerhalb einer Methodendeklaration besucht und ggf. in Unteranweisungen aufgeteilt. Bei der Identifikation bestimmter Anweisungen werden entsprechende Modellelemente im RDSEFF eingefügt. Das Vorgehen zur Erstellung der RDSEFF wurde um folgende Aspekte erweitert:

- Abbildung der Aufrufe von Methoden derselben Klasse,
- Verarbeitung von Java-Annotationen,
- Verarbeitung von verketteten Aufrufen,
- Verarbeitung von Aufrufen innerhalb von Return-Anweisungen,
- präzisere Identifizierung von Schnittstellen, die bestimmte Signaturen anbieten und
- Modellierung von gepufferten Aufrufen (siehe Abschnitt 4.3.1.9)

Aufrufe von Schnittstellen aus dem Repository-Modell werden als *ExternalCallAction* dargestellt. Damit Aufrufe von Methoden derselben Klasse abgebildet werden können, werden diese auch als ExternalCallAction modelliert. Während der Erstellung der RDSEFF werden Annotationen eingelesen und verarbeitet. Entwickler können im Quelltext mit Hilfe von Annotationen die erwartete Ausprägung von Boolean-Variablen und die erwartete Länge von Listenvariablen spezifizieren (siehe auch Abschnitt 4.3.4.1). If-Anweisungen werden als *BranchAction* mit jeweils zwei *Transitionen* und Schleifen als *LoopAction* modelliert. Wenn eine BranchAction von einer annotierten Boolean-Variable abhängt, werden die Wahrscheinlichkeiten der Transitionen entsprechend konfiguriert. Wenn eine LoopAction über die Elemente einer annotierten Listenvariable iteriert, wird die Anzahl der Iterationen entsprechend der Annotation konfiguriert. Enthält eine Anweisung mehrere Methodenaufrufe, wird diese in Unteranweisungen zerlegt und jeder einzelne Aufruf wird abgebildet. Bei der Verarbeitung von Return-Anweisungen werden diese nach Unteranweisungen, die Methodenaufrufe enthalten, durchsucht. Die identifizierten Aufrufe werden innerhalb der RDSEFF vor dem Endknoten eingefügt.

Die Modellierung einer For-Schleife wird in Abbildung 4-21 anhand des Beispiels aus Listing 4-1 dargestellt. Die Methode *loadData* der Klasse *MyBean* besteht aus einer For-Schleife und einem Methodenaufruf. Im Quelltext existieren keine Annotationen und die *LoopAction* spezifiziert eine Iteration je Durchlauf. Der Aufruf der Methode *getData* wird als *ExternalCallAction* modelliert. Die Variablendeklaration innerhalb der For-Schleife wird nicht im RDSEFF modelliert.

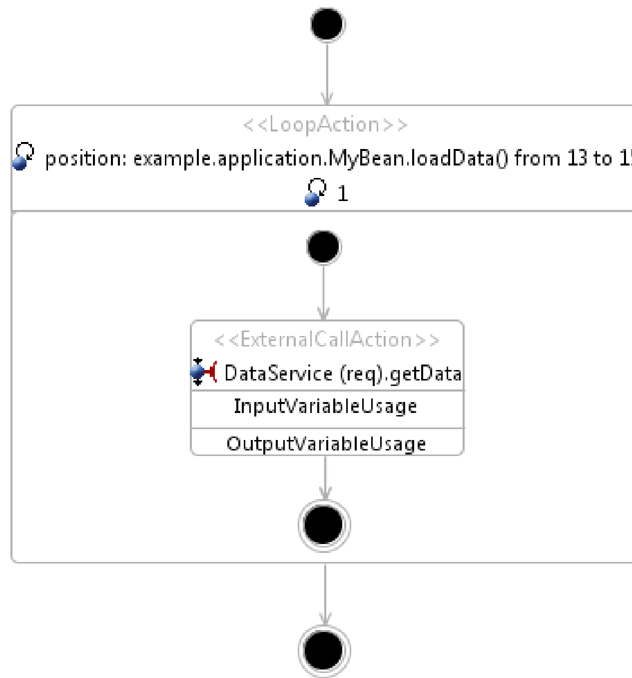


Abbildung 4-21: Vereinfachte Darstellung eines automatisch erstellten RDSEFF für MyBean#loadData
Quelle: Eigene Darstellung

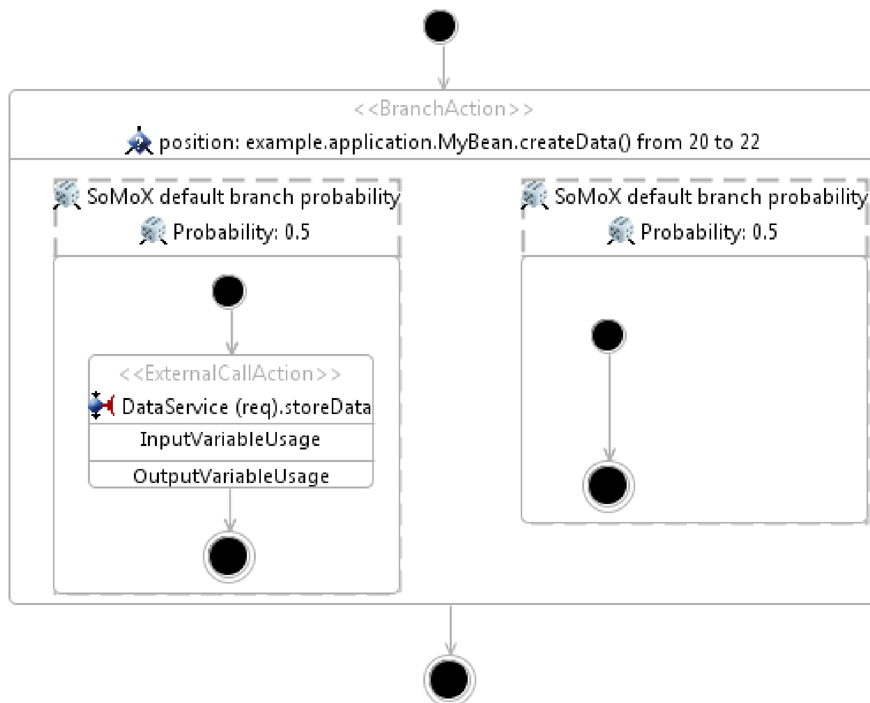


Abbildung 4-22: Vereinfachte Darstellung eines automatisch erstellten RDSEFF für MyBean#createData
Quelle: Eigene Darstellung

Die Modellierung einer If-Anweisung wird in Abbildung 4-22 anhand des Beispiels aus Listing 4-1 dargestellt. Die Methode *createData* der Klasse *MyBean* besteht aus einem Konstruktoraufruf, einer If-Anweisung und einem Methodenaufruf. Konstruktoren werden im RDSEFF nicht abgebildet. Die If-Anweisung wird als BranchAction mit zwei Transitionen modelliert. Da eine Else-Anweisung fehlt, spezifiziert die zweite Transition keine Aktionen. Im Quelltext existieren keine Annotationen und die Transitionen spezifizieren jeweils eine Wahrscheinlichkeit von 50%.

4.3.1.9 Modellierung von gepufferten Aufrufen

Informationen über Methodenaufrufe, deren Rückgabewerte innerhalb einer Nutzersession zwischengespeichert werden, können in der Performancedatenbank verwaltet werden. Während der Erstellung der RDSEFF werden alle gepufferten Methoden ermittelt. Für jede dieser Methoden wird eine Guard-Variable vom Typ Boolean angelegt, die angibt, ob die Methode innerhalb einer Nutzeraktivität schon aufgerufen wurde. Die Variable wird im Usage-Model initialisiert und über alle RDSEFF, die innerhalb der Nutzeraktion aufgerufen werden, weitergegeben. Ungepufferte Aufrufe erhalten die Variable als Eingabe und leiten diese im Kontrollfluss an das darauffolgende RDSEFF weiter. Gepufferte Aufrufe werden als BranchAction mit zwei Transitionen modelliert. Eine Transition enthält den Aufruf und wird dann gewählt, wenn die Variable den Wert *false* hat. Nach dem Aufruf wird dieser Wert auf *true* gesetzt. Die zweite Transition enthält keinen Aufruf und wird dann gewählt, wenn die Variable den Wert true hat.

Ein Beispiel für den Einsatz einer gepufferten Methode ist in Listing 4-2 dargestellt. Die Komponenten *MyBean* und *DefaultDataService* greifen mit Hilfe des *UserService* auf Benutzerdaten zu. Der *UserService* ruft anhand einer ID einen Benutzerdatensatz von der Datenbank ab. Die Methode *readPrivateData* der Komponente *MyBean* ruft den Benutzer ab, um zu prüfen ob dieser existiert. Die Methode *getPrivateData* der Komponente *DefaultDataService* ruft den Benutzer ab, um zu prüfen ob dieser auf die Daten zugreifen darf. Die Implementierung des *UserService* basiert auf JPA und der Datensatz wird automatisch zwischengespeichert. Der zweite Aufruf von getUser durch die Komponente DefaultDataService liest den Datensatz aus dem Puffer.

```
1 //Klasse MyBean.java
2 public class MyBean {
3
4     @Inject
5     private DataService dataService;
6
7     @Inject
8     private UserService userService;
9
10    public void readPrivateData(String userId, String dataId){
11          User user = userService.getUser(userId);
12          if(user == null){
13                return;
14          }
```

```
15          dataService.getPrivateData(userId, dataId);
16    }
17 }
18
19 //Klasse DefaultDataService.java
20 public class DefaultDataService implements DataService {
21
22    @Inject
23    private UserService userService;
24
25    public Data getPrivateData(String userId, String dataId) {
26          User user = userService.getUser(userId);
27          Data data = entityManager.find(Data.class, dataId);
28          if(data.hasAccess(user)){
29                return data;
30          }else{
31                return null;
32          }
33    }
34 }
```

Listing 4-2: ***Erweitertes Beispiel für Komponenten einer Java-EE-Anwendung bei Verwendung einer gepufferten Methode***
Quelle: Eigene Darstellung

Die Modellierung dieses Beispiels wird in Abbildung 4-23 dargestellt. Die Methode *getUser* der Komponente *UserService* ist in der Performancedatenbank als gepuffert gekennzeichnet. Die Guard-Variable *cached_getUser* wird angelegt und mit dem Wert *false* initialisiert. Die Variable wird zuerst über die Systemschnittstelle an die RDSEFF *readPrivateData* der Komponenten *MyBean* übergeben. Hier ist *getUser* zum ersten Mal abgebildet. Der Aufruf ist als BranchAction modelliert. Weil der Wert von *cached_getUser* noch initial ist, wird die Transition mit dem Aufruf gewählt. Nach dem Aufruf wird *cached_getUser* auf *true* gesetzt. Nach der BranchAction wird *getPrivateData* aufgerufen und die Guard-Variable wird übergeben. In der RDSEFF *getPrivateData* ist der Aufruf von *getUser* auch als BranchAction abgebildet. An dieser Stelle wird jedoch die Transition ohne Aufruf gewählt. Der Kontrollfluss wird zurück an *MyBean* übergeben und danach beendet. Bei der darauffolgenden Nutzeraktion wird *cached_getUser* erneut initialisiert.

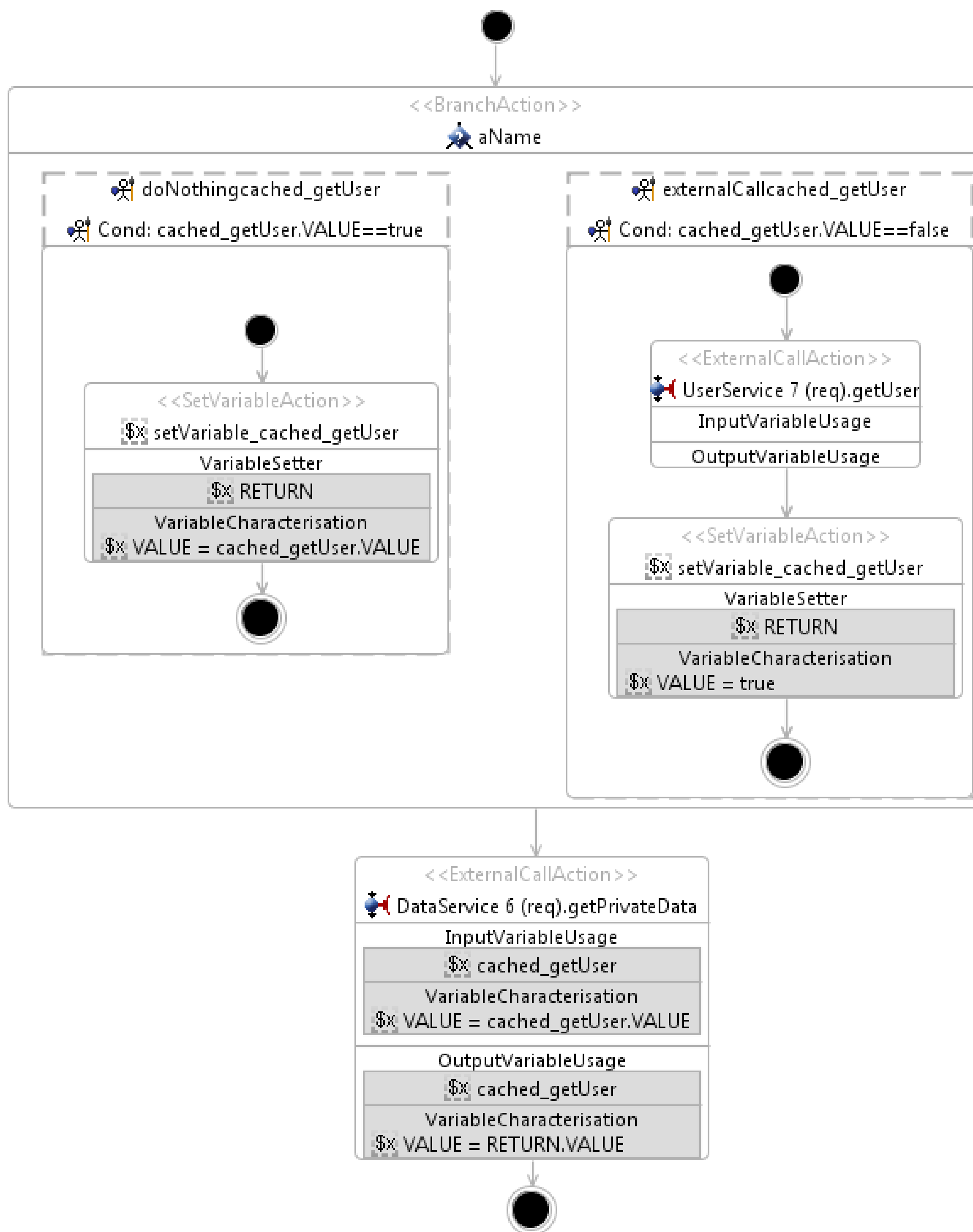


Abbildung 4-23: Beispiel für die Spezifikation von gepufferten Aufrufen
Quelle: Eigene Darstellung

4.3.2 Erhebung des Antwortzeitverhaltens von wiederverwendeten Diensten

Das Antwortzeitverhalten von wiederverwendeten Diensten wird durch die Instrumentierung des Bytecodes von laufenden Unternehmensanwendungen gemessen. Die Umsetzung des Verfahrens wird im Folgenden beschrieben.

4.3.2.1 Monitoring des Antwortzeitverhaltens

Antwortzeitmessungen werden mit Hilfe des Kieker Frameworks[24] durchgeführt. Kieker unterstützt die Überwachung, Analyse und Visualisierung der Performance von Unternehmensanwendungen (van Hoorn et al. 2012). Für das Monitoring von Anwendungen können Messpunkte im Quelltext oder im Bytecode eingefügt werden. Messungen werden lokal gesammelt und können anschließend weiterverarbeitet werden. Rohr et al. (2010) haben beim Einsatz des Frameworks in Produktionsumgebungen einen Overhead im Mikrosekundenbereich je Messpunkt festgestellt. Die wichtigsten Konzepte des Kieker-Monitorings sind (van Hoorn et al. 2009):

- Monitoring Probe: Messungen werden von *Monitoring Probes* durchgeführt. Das Ergebnis der Messung wird als ein *Monitoring Record* übergeben.
- Monitoring Controller: Der *Monitoring Controller* ist für die Initialisierung und Verwaltung der Monitoring-Infrastruktur zuständig. Monitoring Records werden von dem *Controller* entgegengenommen.
- Monitoring Writer: Die Serialisierung bzw. Abspeicherung von Records wird durch den *Monitoring Writer* durchgeführt. Der Writer wird von Controller initialisiert und für jedes einzelne Record aufgerufen. Records können sowohl lokal gespeichert als auch über Netzwerk übergeben werden.

Mit dem Adaptive Monitoring (Kieker Project 2015, S. 22) unterstützt Kieker eine selektive Aktivierung von Messpunkten in einer Anwendung. Durch die Spezifikation von Namensmustern können Probes für eine beliebige Menge an Klassen aktiviert bzw. deaktiviert werden.

Die Umsetzung des Monitorings mit Hilfe von Kieker ist in Abbildung 4-24 dargestellt. Messungen bzw. Records werden auf der lokalen Festplatte gespeichert. Ein Datenbankclient liest diese periodisch ein und sendet sie über eine Representational-State-Transfer-Schnittstelle (REST) an die Performancedatenbank.

---
[24] http://kieker-monitoring.net/

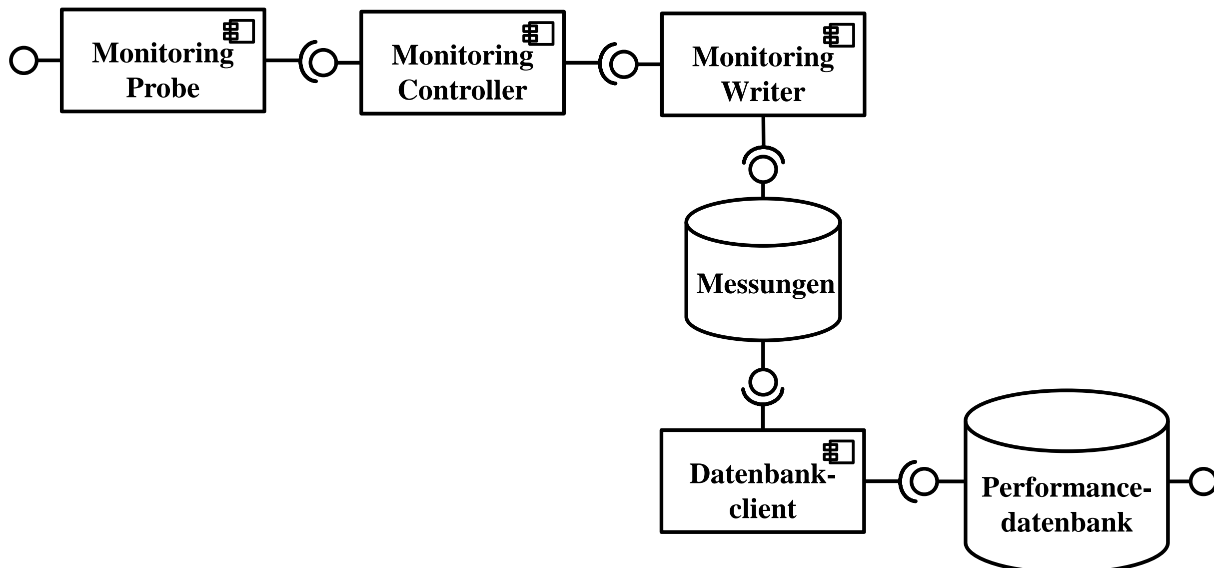


Abbildung 4-24: Erhebung und Speicherung von Antwortzeitmessungen mit Hilfe des Kieker Frameworks
Quelle: In Anlehnung an van Hoorn et al. (2009)

Die Performancedatenbank verfügt über interne Prozeduren für die Berechnung von Durchschnittswerten und Wahrscheinlichkeitsfunktionen für einzelne Dienste. Die Verdichtung der Messungen wird periodisch durchgeführt und zwischengespeichert. Performancekurven spezifizieren das Antwortzeitverhalten abhängig von bestimmten Inputparameter. Über das Monitoring werden nur Antwortzeiten, jedoch keine Parameter erfasst. Die Spezifikation von Performancekurven muss daher manuell erfolgen. Die Performancedatenbank unterstützt die Spezifikation von Formeln für die Berechnung der Antwortzeit in Abhängigkeit der parallel bearbeiteten Aufrufe. Das Antwortzeitverhalten eines Dienstes kann von der IDE über eine REST-Schnittstelle abgefragt werden. Für einen Dienst wird immer nur eine Art der Antwortzeitspezifikation zurückgeliefert. Die Konfiguration der Art der Antwortzeitspezifikation muss für jeden Dienst manuell durchgeführt werden.

4.3.2.2 Instrumentierung von Unternehmensanwedungen

Kieker Monitoring Records können beliebig im Quelltext einer Anwendung eingefügt werden. Dieses Verfahren ist jedoch nur für die temporäre Instrumentierung von einigen wenigen Klassen praktikabel. Eine breite Instrumentierung des Quelltextes kann zur Verschlechterung der Wartbarkeit und Lesbarkeit von Klassen führen. Darüber hinaus wird auch die Verfügbarkeit des Quelltextes vorausgesetzt, was für einige wiederverwendete Dienste nicht immer zutrifft. Durch den Einsatz von AspectJ[25] unterstützt Kieker die Instrumentierung von Bytecode (van Hoorn et al. 2009).

AspectJ stellt eine Erweiterung der Java-Programmiersprache um Aspektorientierung (Laddad 2009, S. 15f) dar. Aspektorientierte Programmierung unterstützt die Trennung der Kernfunktionen einer Software, von Querschnittsaspekten, wie z.B. Sicherheit oder Performance (Kiczales et al. 1997; Binder et al. 2007). Im Rahmen des Weaving werden die Trennung aufgelöst und die Aspekte mit den Kernfunktionen verwoben. Mit diesem Verfahren werden auch Probes als

[25] https://eclipse.org/aspectj/

Aspekte in Klassen eingefügt. Über das Load-Time Weaving ermöglicht AspectJ die Instrumentierung des Bytecode während Klassen geladen werden (Laddad 2009, S. 206). Ein Überblick über das Verfahren ist in Abbildung 4-25 dargestellt.

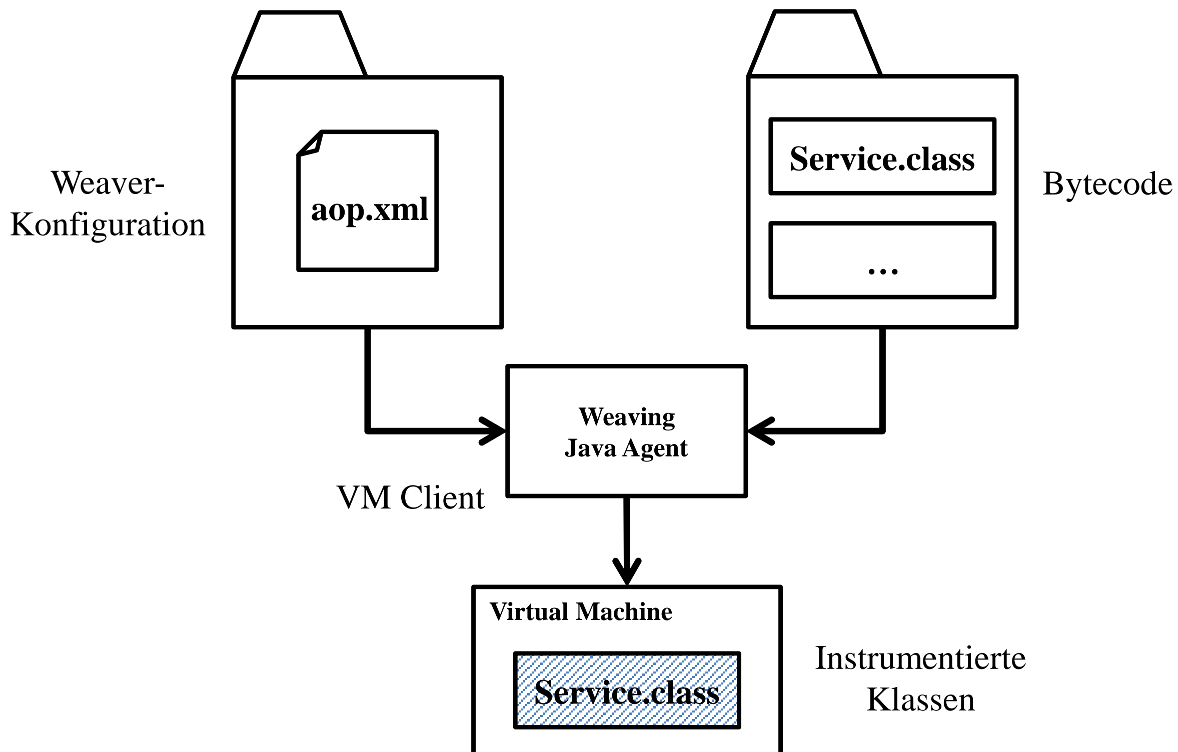


Abbildung 4-25: Instrumentierung des Bytecodes mit Hilfe von AspectJ
Quelle: In Anlehnung an Laddad (2009, S. 207)

Als Input erhält der Weaving Java Agent den Bytecode einer Anwendung und eine Weaver-Konfiguration. Konfigurationen werden in Dateien mit dem Namen aop.xml abgelegt und beschreiben welche Aspekte mit welchen Klassen und Methoden verwoben werden sollen. Der Agent beobachtet die Aktivität der virtuellen Maschine und wartet das Laden von Klassen ab. Während dem Laden einer Klasse weist der Agent die virtuelle Maschine an, die konfigurierten Aspekte in die Klasse einzubauen.

4.3.2.3 Zuordnung von Komponenten zu Schnittstellen

Für die Instrumentierung von Anwendungen mit AspectJ bietet Kieker verschiedene vordefinierte Aspekte an. Zustandsbasierte Aspekte erfassen die Ausführung einer Methode und berechnen automatisch die Zeitdauer. Ereignisbasierte Aspekte erfassen nur den Zeitpunkt des Aufrufs oder der Ausführung einer Methode. Bei der Erfassung der Aufrufe von Methoden wird sowohl der Name der aufgerufenen als auch der ausgeführten Klasse/Methode protokolliert. Hiermit können, wie im Abschnitt 4.2.7 beschrieben, Komponenten zu den implementierten Schnittstellen zugeordnet werden. Neben Methoden können auch Konstruktoren und Objektinstanzen beobachtet werden. Der Ansatz sieht vor, dass zustandsbasierte Aspekte für die Messung der Antwortzeiten und ereignisbasierte Aspekte für die Zuordnung von Implementierungen zu Schnittstellen miteinander kombiniert werden.

Während der Modellerstellung wird zwischen expliziten und impliziten Schnittstellen unterschieden (siehe auch Abschnitt 4.2.1). Der SoMoX-Ansatz verfolgt hierbei eine bestimmte Namenskonvention. Namen von impliziten Schnittstellen werden mit dem Präfix *I* versehen. Namen von expliziten Schnittstellen entsprechen dem Namen der Komponente, von der sie abgeleitet wurden. Bei der Abfrage des Antwortzeitverhaltens erkennt die Performancedatenbank anhand dieser Namenskonvention ob Messungen für eine Komponente oder eine Schnittstelle benötigt werden.

4.3.2.4 Architektur zur Erhebung und Abfrage von Antwortzeitmessungen

Die Architektur zur Erhebung, Speicherung und Abfrage von Antwortzeitmessungen ist in Abbildung 4-26 dargestellt. Unternehmensanwendungen werden in der Produktions- oder Testumgebung auf Applikationsserver deployed und instrumentiert. Im Java EE Container werden Komponenten zur Bearbeitung der Geschäftslogik ausgeführt. Über einen Zeitraum werden Antwortzeitmessungen durchgeführt und periodisch von Datenbankclients an die zentrale Performancedatenbank übermittelt. Mittelwerte und Wahrscheinlichkeitsfunktionen werden periodisch neu berechnet und zwischengespeichert. Während der Erstellung von Performancemodellen in der IDE des Entwicklers wird das Antwortzeitverhalten bestimmter Dienste bei der Performancedatenbank abgefragt.

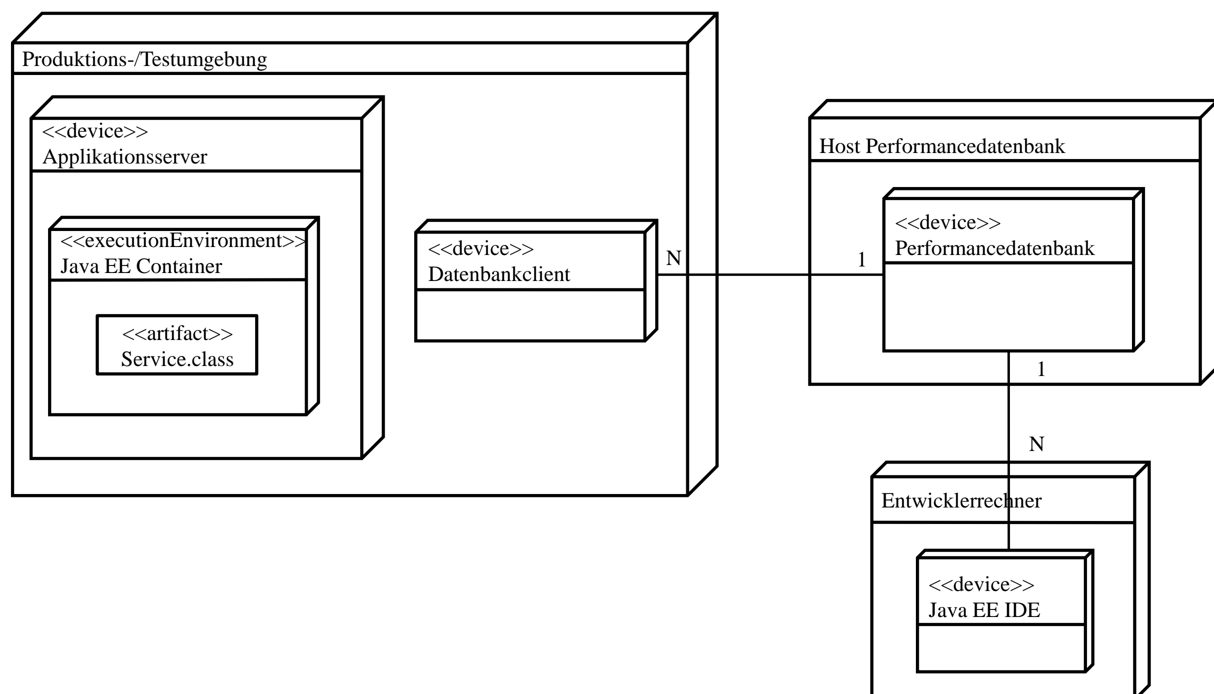


Abbildung 4-26: Architektur zur Erhebung und Speicherung von Antwortzeitmessungen
Quelle: Eigene Darstellung

4.3.3 Simulation des Antwortzeitverhaltens

Das erstellte Performancemodell wird mit Hilfe des SimuCom Frameworks simuliert (Becker et al. 2009). Als Ausgangsbasis dienen die PCM-Instanz und eine Simulationskonfiguration.

SimuCom transformiert die PCM-Instanz zu Java-Quelltext, der die Struktur und das Verhalten des Modells abbildet. PCM-Komponenten werden als Java-Klassen dargestellt. Externe Aufrufe von RDSEFFs werden als Aufrufe von Java-Methoden abgebildet. Während der Ausführung des Quelltextes werden Performancemessungen erhoben.

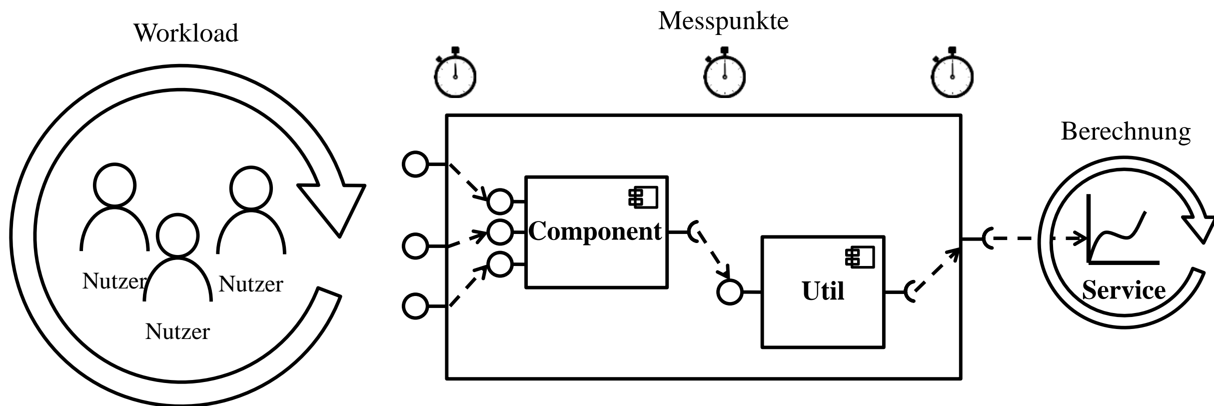


Abbildung 4-27: Simulation des Performancemodells mit Hilfe von SimuCom
Quelle: Eigene Darstellung

Ein Überblick über die wichtigsten Aspekte der Simulation ist in Abbildung 4-27 dargestellt. Während der Simulation wird der im Usage-Modell spezifizierte Workload auf die in RDSEFFs abgebildeten Operationen ausgeführt. Jeder Dienst der fokussierten Komponente wird als Schnittstelle des Systems nach außen angeboten. Die restlichen Komponenten können durch Nutzer nicht direkt aufgerufen werden. Nutzer, die im Usage-Modell spezifiziert sind, werden mit Hilfe von Java-Threads abgebildet. Diese nutzen die externen Systemschnittstellen mit einer gleichverteilten Wahrscheinlichkeit. Bei einem offenen Workload werden Nutzer mit einer konfigurierten Frequenz erzeugt. Beim geschlossenen Workload existiert eine feste Anzahl Nutzer, die mit dem System interagieren. Für Dienste von Komponenten werden Messpunkte in Form von s.g. Sensoren angelegt. Abhängig von der verwendeten Systemschnittstelle kann eine Nutzerinteraktion zur Ausführung verschiedener Komponenten und Dienste führen. Eine Simulation wird beendet nachdem entweder die konfigurierte Zeitdauer abgelaufen ist, oder genug Messungen erhoben wurden. Je intensiver der Workload, desto schneller kann eine Simulation beendet werden.

Die Antwortzeit eines wiederverwendeten externen Dienstes wird für jede einzelne Ausführung im Kontrollfluss erneut bestimmt. Bei der Spezifikation eines Maßes der zentralen Tendenz, wird jeweils ein konstanter Wert zurückgeliefert. Für Wahrscheinlichkeitsfunktionen wird jeweils eine Zufallszahl mit den konfigurierten Wahrscheinlichkeiten generiert. Bei Performancekurven wird anhand der Inputparameter ein Wert aus der Tabelle abgerufen oder anhand der Formel ein Wert berechnet. Inputparameter, wie z.B. die aktuelle Anzahl paralleler Aufrufe eines Dienstes, werden von SimuCom verwaltet und für die Berechnung von Performancekurven bereitgestellt. Beim Einsatz von konstanten Werten ist auch das beobachtete Performanceverhalten relativ konstant und Simulationen können frühzeitig beendet werden. Wahrscheinlichkeitsfunktionen und Performancekurven bedürfen einer längeren Simulationszeit für eine möglichst lückenlose Darstellung des spezifizierten Antwortzeitverhaltens. Oszillationen in der

Gesamtantwortzeit werden auch durch die Existenz von bedingten Aufrufen und Schleifen herbeigeführt.

Nach der Simulation werden alle Messungen der Sensoren abgerufen und je Dienst zusammengefasst. Die Verdichtung von Messungen aus der Simulation unterliegt denselben Fragestellungen wie für die Messungen aus der Produktions-/Testumgebung (siehe auch Abschnitt 4.2.6). Die Verteilung der Beobachtungen während der Simulation wird von der Abbildung des Verhaltens wiederverwendeter Dienste und der Konfiguration von Kontrollstrukturen maßgeblich beeinflusst. Die aktuelle Implementierung des Ansatzes sieht die Berechnung des arithmetischen Mittels über die beobachtete Antwortzeit je Sensor vor.

4.3.4 Interaktion mit dem Softwareentwickler

Der Ansatz für Performancebewusstsein unterstützt die Interaktion mit dem Softwareentwickler in beide Richtungen. Der Softwareentwickler kann eine Antwortzeitvorhersage anfordern und konfigurieren. Der Ansatz meldet dem Entwickler die Ergebnisse der Vorhersage auf mehreren Ebenen. Die Konfiguration der Vorhersage und die Rückmeldung der Ergebnisse werden im Folgenden beschrieben.

4.3.4.1 Konfiguration durch den Softwareentwickler

Die Konfiguration der Vorhersage durch den Softwareentwickler kann auf zwei Ebenen stattfinden. Über die Eclipse-Einstellungen werden für alle Antwortzeitvorhersagen die folgenden Parameter eingestellt:

- Klassifizierung von externen Komponenten: Über diesen booleschen Parameter wird die Klassifizierung von Komponenten, für die sowohl der Quelltext als auch Antwortzeitmessungen existieren, beeinflusst.
- Schwellenwert für Warnhinweise: Dieser numerische Parameter spezifiziert ab welcher vorhergesagten Antwortzeit eine Warnmeldung angezeigt werden soll.
- Schwellenwert für Fehlerhinweise: Dieser numerische Parameter spezifiziert ab welcher vorhergesagten Antwortzeit eine Fehlermeldung angezeigt werden soll.
- Anzahl der simulierten Nutzer: Über diesen Wert wird die Anzahl der virtuellen Nutzer während eines Simulationslaufs spezifiziert.
- Bedenkzeit für simulierte Nutzer: Über diesen Wert wird die Zeitverzögerung zwischen den Systemaufrufen durch die simulierten Nutzer während der Simulation spezifiziert.

Der Ansatz unterstützt die Verfeinerung der Modellerstellung für bestimmte Komponenten durch den Entwickler mit Hilfe von Annotationen (siehe auch Abschnitt 4.3.1.8). Mit Hilfe der Annotation *BooleanVariableUsage* kann der Entwickler für einen booleschen Parameter oder eine lokale boolesche Variable die Wahrscheinlichkeit, dass zur Laufzeit der Komponente der Wert true angenommen wird, spezifizieren. Alle If-Anweisungen, welche von diesem Parameter abhängen, werden während der Erstellung der RDSEFF als BranchAction modelliert und entsprechend konfiguriert. Mit Hilfe der Annotation *ListVariableUsage* kann der Entwickler für ein beliebiges Listen-Objekt die Anzahl Elemente, die zur Laufzeit der Komponente

erwartet werden, spezifizieren. Als Listen-Objekt gelten in diesem Kontext Parameter und lokale Variablen, über deren Elemente iteriert wird. Alle Schleife, die über Elemente dieses Objekts iterieren, werden während der Erstellung der RDSEFF als LoopAction modelliert und entsprechend konfiguriert. Ein Beispiel für den Einsatz dieser Annotationen ist in Listing 4-3 dargestellt. Ohne diese Annotationen wird für BranchAction-Transitionen eine Wahrscheinlichkeit von jeweils 50% und für LoopAction eine Wiederholung angenommen.

```
1 //Klasse MyBean.java
2 public class MyBean {
3
4     @Inject
5     private DataService dataService;
6
7     public void loadData(
8     @ListVariableUsage(expectedSize = 2) List<String> dataIdList){
9
10          for(String id : dataIdList){
11                dataService.getData(id);
12          }
13    }
14
15    public void createData(String dataId,
16          @BooleanVariableUsage(probabilityTrue = 0.9) boolean doStore){
17
18          Data data = new Data(dataId);
19          if(doStore){
20                dataService.storeData(data);
21          }
22    }
23 }
```

Listing 4-3: ***Spezifikation von Annotationen für die Verfeinerung der Modellerstellung***
Quelle: Eigene Darstellung

Innerhalb der Deklaration der Methode *loadData* wird der Parameter *dataIdList* mit der Annotation *ListVariableUsage* versehen. Über das Annotationsattribut *expectedSize* gibt der Entwickler an, dass die Liste zur Laufzeit im Durchschnitt zwei Elemente enthalten wird. Eine Schleife innerhalb der Methodendeklaration iteriert über die Elemente von *dataIdList*. Aufgrund der Annotation wird die vorhergesagte Antwortzeit doppelt so hoch sein wie davor. Innerhalb der Deklaration der Methode *createData* wird die boolesche Variable *doStore* mit der Annotation *BooleanVariableUsage* versehen. Über das Annotationsattribut *probabilityTrue* gibt der Entwickler an, dass der Wert zur Laufzeit mit einer Wahrscheinlichkeit von 90% *true* sein wird. Aufgrund der Annotation erhält die Transition mit dem Aufruf von *storeData* eine höhere Wahrscheinlichkeit. Bei einer hinreichend großen Anzahl an Wiederholungen würde die vorhergesagte Antwortzeit um den Faktor 1,8 höher sein als davor.

4.3.4.2 Rückmeldung zum Softwareentwickler

Nach der Durchführung einer Simulation wird die Antwortzeitvorhersage für jeden einzelnen Dienst der fokussierten Komponente geprüft. Übersteigt diese den ersten Schwellenwert wird eine Warnung angezeigt. Bei der Überschreitung des zweiten Schwellenwertes wird ein Fehler gemeldet. Hinweise werden mit Hilfe von Eclipse Marker (Gallardo et al. 2003, S. 335) angezeigt. Marker referenzieren eine Zeichenfolge innerhalb eines Editors und weisen folgende Elemente auf:

- Symbol: Auf der linken Seite des Editors wird gegenüber der referenzierten Zeichenfolge ein grafisches Symbol angezeigt.
- Textumrandung: Optional kann die referenzierte Zeichenfolge auch im Editor über eine Textumrandung hervorgehoben werden.
- Link: Auf der linken Seite des Editors wird ein Link, der auf die referenzierte Stelle im Text verweist, eingeblendet.
- Tooltip: Zusätzliche Texthinweise oder Aktionen zu einem Marker können mit Hilfe von Tooltips angeboten werden.

Marker können einzelne Wörter oder ganze Textabschnitte referenzieren. Die Eclipse-Plattform kann beliebig um neue Marker erweitert werden. Die Rückmeldung der Antwortzeitvorhersagen auf unterschiedliche Ebenen wird im Folgenden beschrieben.

Hinweise für Methodendeklarationen

Die Überschreitung des Schwellenwerts bei den Diensten der fokussierten Methode wird auf Ebene der entsprechenden Methodendeklaration angezeigt. Als Symbol wird ein gelber bzw. roter Tacho gegenüber der Deklaration verwendet (siehe Abbildung 4-28). Der erzeugte Marker umrandet den Methodennamen mit einem roten Rechteck.

Hinweise zu Methodenaufrufen

Die Überschreitung des Schwellenwerts einzelner Aufrufe innerhalb der Methodendeklaration wird mit einem separaten Marker angezeigt. Als Symbol wird ein gelbes oder ein rotes Quadrat gegenüber dem Methodenaufruf verwendet (siehe Abbildung 4-28). Der erzeugte Marker umrandet den ganzen Methodenaufruf, inkl. des Objektnamens. Verkettete Methodenaufrufe werden dabei voneinander abgegrenzt. Es werden sowohl Aufrufe von internen als auch externen Komponenten berücksichtigt.

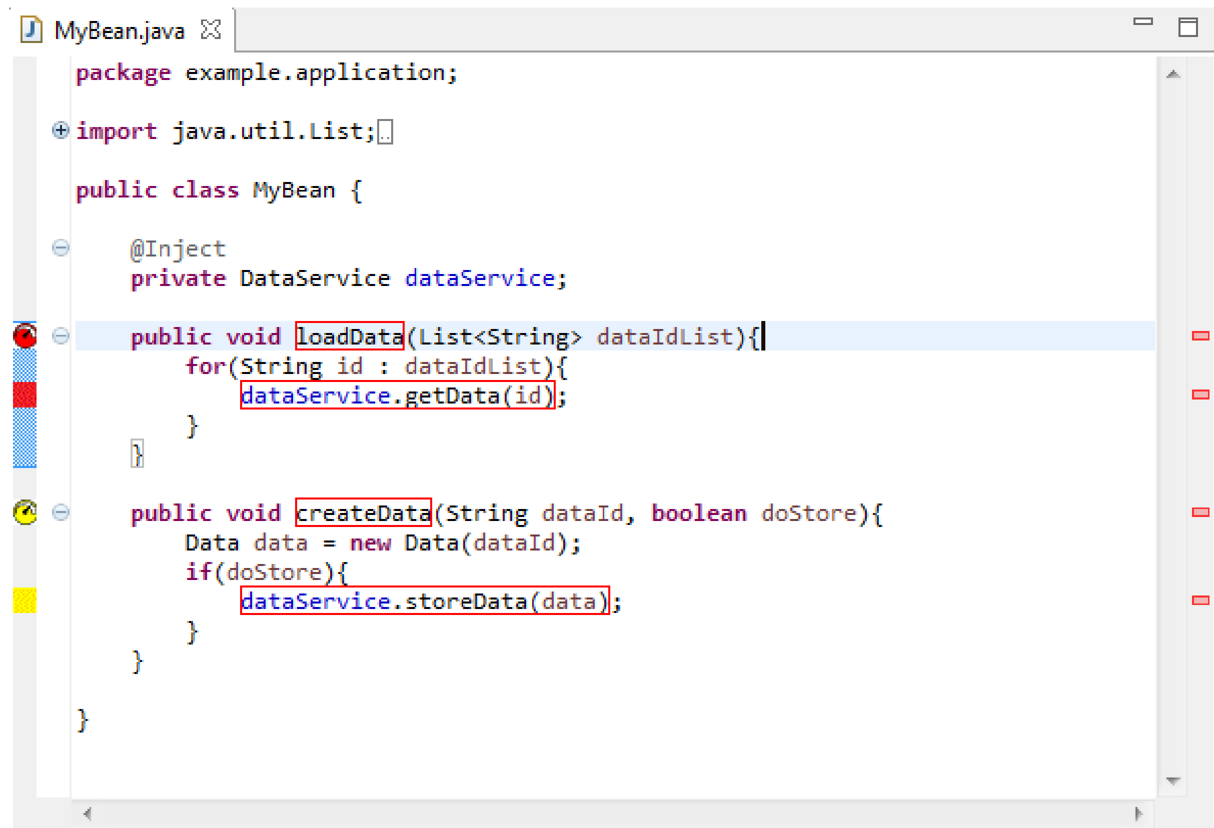


Abbildung 4-28: Rückmeldung der Antwortzeitvorhersage zum Entwickler im Quelltexteditor
Quelle: Danciu/Krcmar (2018)

Im Beispiel aus Abbildung 4-28 überschreitet die Antwortzeitvorhersage für *loadData* den Schwellenwert für Fehlermeldungen. Ausschlaggebend hierfür ist der Aufruf von *getData*, der den Schwellenwert auch überschreitet. Die Vorhersage für createData ist geringer als der Schwellenwert für Fehlermeldungen, aber höher als der für Warnmeldungen. Ausschlaggebend für die Überschreitung ist der Aufruf von *storeData*.

Anzeige der vorhergesagten Antwortzeit als Tooltip

Die eigentliche Antwortzeitvorhersage wird für Methodendeklarationen und Aufrufen mit Hilfe von Tooltips am Symbol und am Link angezeigt (siehe Abbildung 4-29). Im Tooltip werden der Methodenname und die Antwortzeit in Millisekunden angezeigt. Existieren mehrere Marker innerhalb einer Zeile, überlagern sich die Symbole. Im Tooltip werden die Hinweise aller Marker angezeigt.

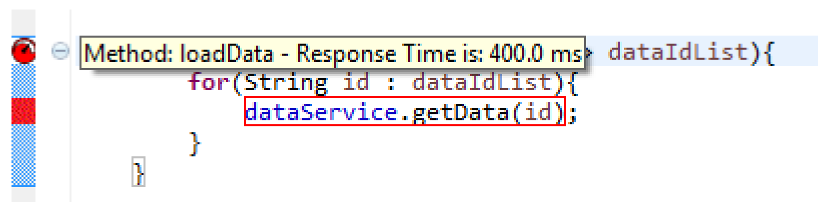


Abbildung 4-29: Rückmeldung der Antwortzeitvorhersage als Tooltip
Quelle: Danciu/Krcmar (2018)

Im Beispiel aus Abbildung 4-29 beträgt die vorhergesagte Antwortzeit für *loadData* 400 Millisekunden.

Hinweise in der Aufrufhierarchie

Die Ursache einer hohen Antwortzeit kann innerhalb der fokussierten Komponenten oder auch tiefer in der Aufrufhierarchie anzutreffen sein. Eine große Anzahl an aufgerufene Methoden und eine tiefe Aufrufhierarchie erschweren die Suche des Entwicklers nach der Ursache. Der Ansatz für Performancebewusstsein fügt daher Hinweismeldungen in der ganzen Aufrufhierarchie einer Komponente ein (siehe Abbildung 4-30). Damit wird sowohl auf horizontaler als auch auf vertikaler Ebene der Suchraum eingegrenzt.

Ausgehend von der fokussierten Komponente werden alle Aufrufe, welche den Schwellenwert überschreiten, identifiziert. Je Aufruf wird die Java-Klasse, welche die Methode deklariert, ermittelt. Innerhalb dieser Deklaration wird der Vorgang wiederholt.

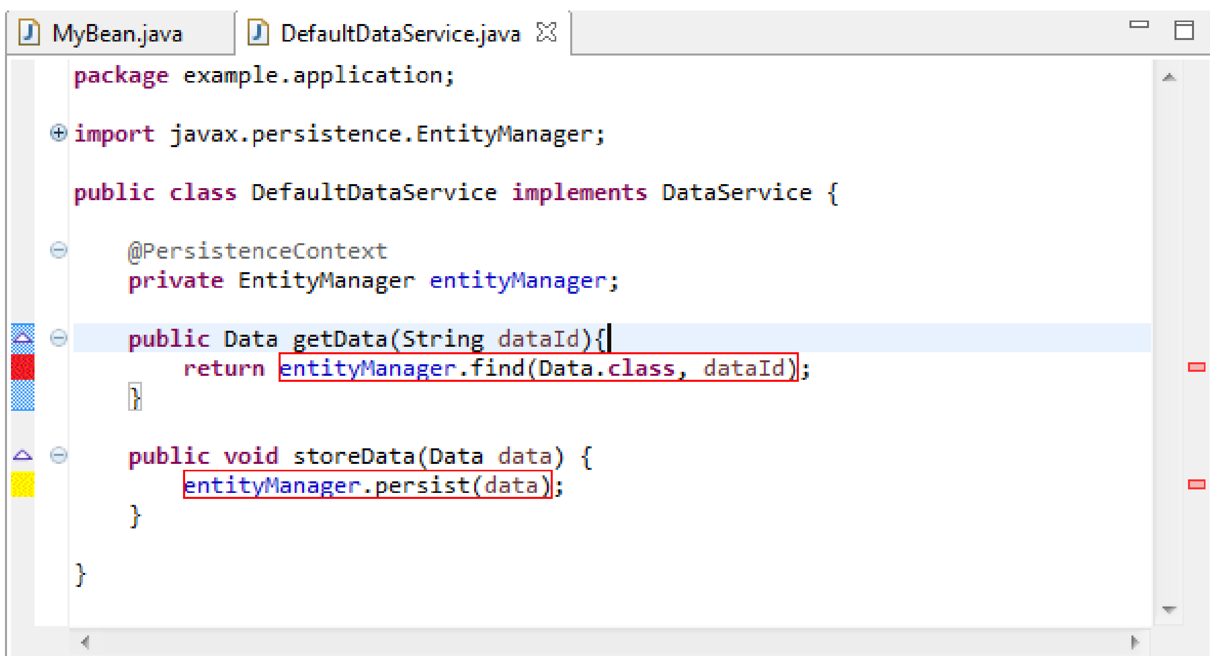


Abbildung 4-30: Rückmeldung der Antwortzeitvorhersage in der Aufrufhierarchie
Quelle: Danciu/Krcmar (2018)

Im Beispiel aus Abbildung 4-30 wurden zuerst auf der Ebene der fokussierten Komponente *MyBean* Hinweise eingefügt. Anschließend wurden die Aufrufe von getData und storeData verfolgt. In der Klasse *DefaultDataService* werden diese Methoden deklariert. Die Aufrufe des *EntityManager* überschreiten die konfigurierten Schwellenwerte und werden entsprechend markiert.

4.4 Evaluation der Vorhersagegenauigkeit

Das Ziel des Ansatzes für Performancebewusstsein ist es Softwareentwickler über die erwartete Antwortzeit von Komponenten zu informieren und bei deren Optimierung zu unterstützen. Die

Antwortzeit der Komponenten soll dadurch verbessert werden. Eine hohe Präzision der Antwortzeitvorhersagen bildet dabei eine wichtige Voraussetzung zur Erreichung dieses Ziels.

Den Schwerpunkt der Antwortzeitvorhersage bilden die wiederverwendeten Dienste. Die Berücksichtigung des Kontrollflusses und der Parametrisierung von Aufrufen dient dazu den Einfluss von Wiederverwendung auf die Antwortzeit noch genauer abzubilden. Im Rahmen dieser Evaluation wird die Durchführbarkeit des Ansatzes untersucht. Folgende Aspekte werden dabei im Detail adressiert:

- Wie präzise sind Antwortzeitvorhersagen für Komponenten unter Berücksichtigung der wiederverwendeten Dienste?
- Welchen Einfluss hat die Wahl eines Maßes der zentralen Tendenz für die Abbildung des Antwortzeitverhaltens von wiederverwendeten Diensten auf die Präzision der Vorhersagen?
- Welchen Einfluss hat die Spezifikation der Ausprägung von Boolean-Parametern und der Anzahl der Elemente von Listenparametern auf die Präzision der Vorhersagen?

Zur Untersuchung der Präzision der Vorhersagen wird ein technologieorientiertes Experiment (Wohlin et al. 2012, S. 76) durchgeführt. Antwortzeiten werden für Dienste von Komponenten zuerst vorhergesagt. Die Komponenten werden anschließend ausgeführt und die tatsächlichen Antwortzeiten gemessen. Die Vorhersagen werden mit den Messungen verglichen und die durchschnittliche Abweichung wird berechnet. Das Vorgehen setzt die Verfügbarkeit des Quelltexts einer ausführbaren Komponente und die Wiederverwendung von Komponenten, für die Antwortzeitmessungen vorliegen, voraus. Diese Voraussetzungen werden beim Einsatz des SPECjEnterprise2010-Benchmarks[26] erfüllt. Darüber hinaus bietet der Benchmark eine standardisierte Technik für die Erzeugung von Last für Java-EE-Komponenten. Damit werden verschiedene Messläufe miteinander vergleichbar. Der Einsatz des Benchmarks und die Ergebnisse der Evaluation werden im Folgenden beschrieben.

4.4.1 Aufbau des Experiments

Für einzelne Dienste von Komponenten wird die vorhergesagte mit der tatsächlich gemessenen Antwortzeit verglichen. Das Antwortzeitverhalten von Komponenten wird in zwei unterschiedlichen Wiederholungen des Experiments als Median und als 95-Perzentil abgebildet. Mit Hilfe des Medians wird der Einfluss von Ausreißern reduziert. Die Verwendung des 95-Perzentils bildet ein pessimistisches Szenario ab, in dem von höheren Antwortzeiten ausgegangen wird, jedoch nicht vom maximalen Wert. In einem Szenario wird dasselbe Maß der zentralen Tendenz sowohl für die Verdichtung der Antwortzeitmessungen von wiederverwendeten Komponenten als auch für die Berechnung der Antwortzeit der fokussierten Komponente verwendet.

[26] SPECjEnterprise2010 ist ein Warenzeichen der Standard Performance Evaluation Corporation (SPEC). Die Ergebnisse des Benchmarks in dieser Publikation wurden durch von der SPEC nicht begutachtet oder akzeptiert. Daher können weder Vergleiche, noch Schlußfolgerungen im Zusammenhang mit anderen SPEC-Ergebnissen druchgeführt werden.Die offizielle Webseite des SPECjEnterprise2010 kann unter http://www.spec.org/jEnterprise2010 abgerufen werden.

Simulationsergebnisse werden aufgrund der geringen Oszillation als arithmetisches Mittel verdichtet.

Der Einfluss der Verfügbarkeit von Informationen über die Parametrisierung auf die Präzision der Vorhersagen wird durch die Spezifikation von Annotationen untersucht. In zwei unterschiedlichen Wiederholungen wird die Präzision der Vorhersagen beim Vorhandensein und beim Fehlen von Annotationen bestimmt.

4.4.2 Experimentumgebung

SPECjEnterprise2010 ist ein Benchmark zur Untersuchung und zum Vergleich der Performance von Java-EE-Servern (SPEC 2014). Eine Java-EE-Anwendung implementiert die Geschäftslogik eines Automobilherstellers und wird über eine Benutzeroberfläche oder über Web-Service-Schnittstellen vom Lasttreiber aufgerufen. Wie in Abbildung 4-31 dargestellt, ist die Anwendung in Domains, die Geschäftsbereiche darstellen, aufgeteilt (SPEC 2014):

- Die Orders Domain implementiert die Verwaltung der Bestände von Autohändler. Nutzer dieser Domain können Autos abrufen, bestellen und verkaufen.
- Die Manufacturing Domain implementiert die Planung und Steuerung der Produktion von Autos über unterschiedliche Werke.
- Die Supplier Domain implementiert die Materialplanung und den Einkauf bei Zulieferer.

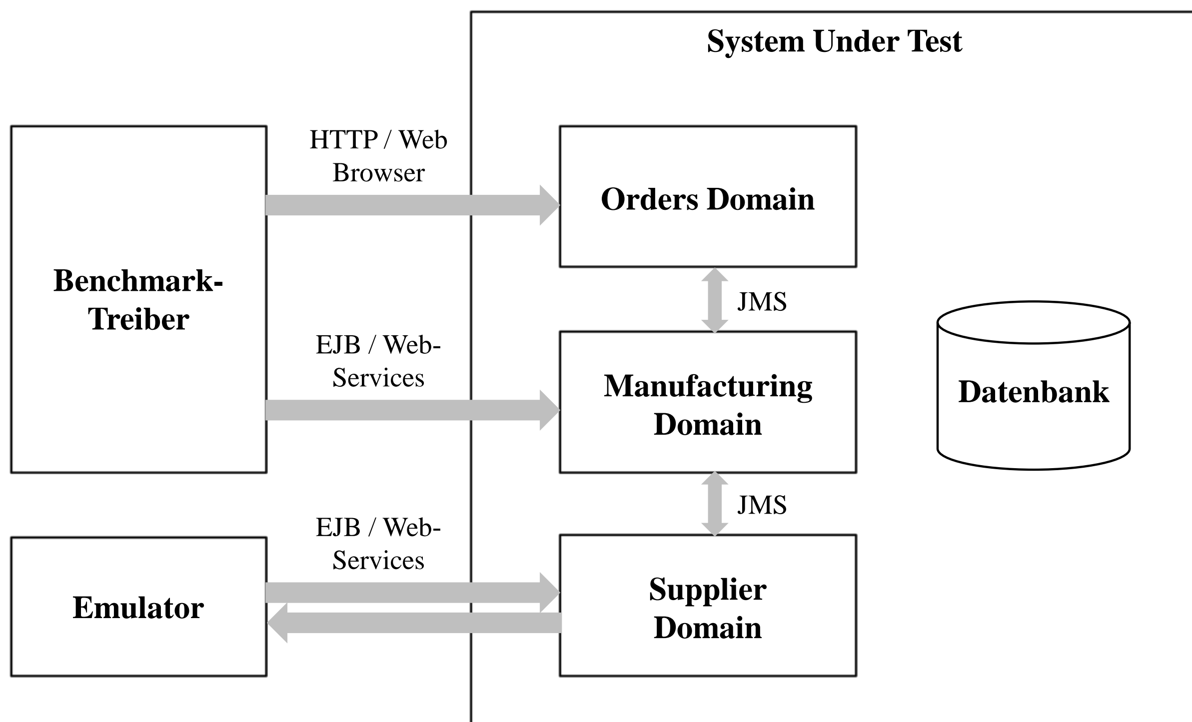


Abbildung 4-31: Treiber und Domains des SPECjEnterprise2010-Benchmark
Quelle: In Anlehnung an SPEC (2014)

Die Domains kommunizieren untereinander über Java Message Service (JMS) und greifen auf eine Datenbank zu. Der Treiber simuliert Nutzer, welche mit der Anwendung interagieren, und

führt Antwortzeitmessungen durch. Der Emulator implementiert Web Services für den Empfang von Bestellungen und die Rückmeldung von Lieferungen. Die Geschäftslogik der Domains wird von Java-EE-Komponenten, die einander wiederverwenden, umgesetzt. Mit Hilfe des Treibers können Antwortzeitmessungen für die Komponenten erhoben werden. Aufgrund der Verfügbarkeit des Quelltextes kann die Antwortzeit einzelner Komponenten vorhergesagt werden. Die Grundlage dieses Experiments bilden die Komponenten der Orders Domain. Auch andere Referenzimplementierungen für Java-EE-Komponenten können für die Untersuchung der Vorhersagegenauigkeit eingesetzt werden. Diese unterstützen jedoch keine standardisierte Erzeugung von Last.

Die Orders Domain besteht aus drei Schichten (siehe Abbildung 4-32), die jeweils die darunterliegende Schicht wiederverwenden. Ein zentrales Servlet nimmt Nutzeranfragen entgegen und leitet diese weiter. Stateless EJB sind für die Bearbeitung der Nutzeranfragen zuständig. JPA Entities bilden Geschäftsobjekte, wie z.B. Kunden und Bestellungen, ab und werden mit Hilfe der JPA ausgelesen bzw. gespeichert.

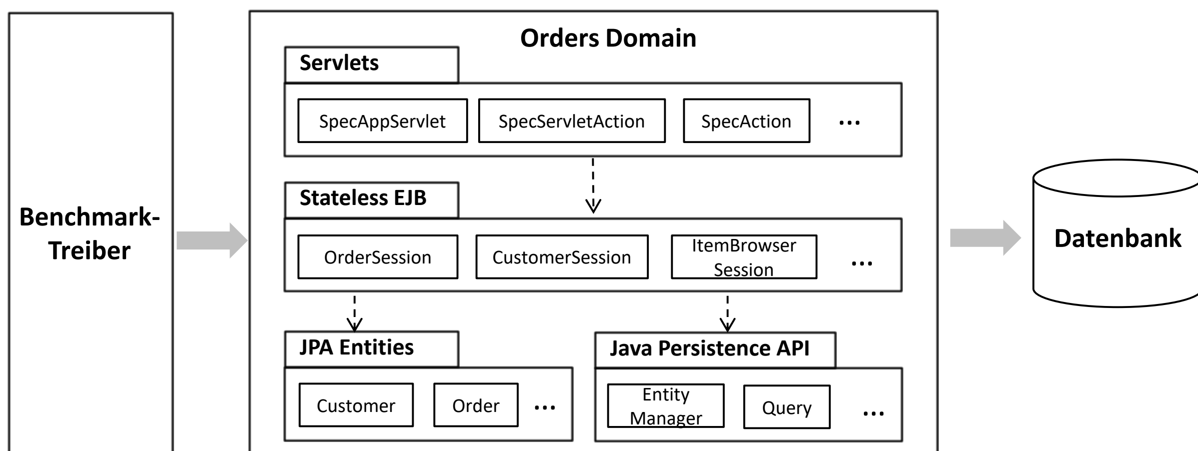


Abbildung 4-32: Überblick über die Schichten der SPECjEnterprise2010 Orders Domain
Quelle: In Anlehnung an Danciu et al. (2015b)

Im Rahmen des Experiments werden Antwortzeitvorhersagen für einzelne EJB-Methoden durchgeführt. Jeweils eine EJB stellt die fokussierte Komponente dar. Elemente der JPA-Schicht stellen wiederverwendete Dienste dar, für die Antwortzeitmessungen vorliegen. Von der fokussierten Komponente aufgerufene EJB werden als interne Komponenten abgebildet. Für einzelne Methoden der fokussierten Komponente werden Antwortzeitvorhersagen durchgeführt. Antwortzeitmessungen werden während dem Ausführen des Benchmark-Treibers durchgeführt. Zur Erhebung von Antwortzeiten der JPA-Schicht werden alle Aufrufe der entsprechenden Komponenten instrumentiert. Zur Erhebung von Vergleichswerten für Vorhersagen wird die Ausführung der EJB-Methoden instrumentiert. Die Instrumentierung einzelner Methoden kann einen Overhead bei der Messung einer übergeordneten Schicht bewirken. Um dies auszuschließen, werden Komponenten einzeln in separaten Durchläufen instrumentiert.

Für die Einbeziehung als Dienst einer fokussierten Komponente sind jedoch einige EJB-Methoden ungeeignet und werden aus dem Experiment ausgeschlossen. Die zugrundliegenden Kriterien sind im Folgenden beschrieben.

Vorangegangene Pufferung von Datenbankobjekten

Einige EJB greifen beim Aufruf des *EntityManager* auf denselben Puffer zu. Nach dem Einlesen eines Entity durch die *OrderSession* könnte die *CustomerSession* dasselbe Objekt aus dem Puffer abrufen, anstatt von der Datenbank. Die Abbildung dieses Mechanismus, von der ersten Datenbankabfrage bis hin zum Pufferzugriff, wird vom Ansatz unterstützt. Bei der Modellierung der nachgelagerten EJB als fokussierte Komponente wird ein vorangegangener Aufruf des *EntityManager* nicht abgebildet. Bei der Ausführung des Benchmarks wird jedoch die Antwortzeit des Pufferaufrufs gemessen. Die Gegenüberstellung der Vorhersage mit der Messung für diese Methode weist große Abweichungen auf. Antwortzeitvorhersagen für Methoden, die von einer vorangegangenen Pufferung betroffen sind, werden nicht betrachtet. Aufgrund dieses Kriteriums wird z.B. die Methode *addInventory* des *CustomerSession* ausgeschlossen. Die übrigen Methoden der *CustomerSession* werden jedoch weiterhin betrachtet.

Ausführung von Datenbankabfragen

Einige EJB-Methoden führen Datenbankabfragen aus. Die Antwortzeit der Abfragen hängt stark von der Parametrisierung hinsichtlich der referenzierten Tabellen, Spalten und Zeilen ab. Diese Parameter werden weder bei der der Messung des Antwortzeitverhaltens von wiederverwendeten Diensten, noch bei der Erstellung von Performancemodellen betrachtet. Im Rahmen dieses Experiments werden EJB-Methoden als Datenzugriffsdienste betrachtet, wenn sie direkte Datenbankabfragen ausführen. Datenzugriffsdienste implementieren grundlegende Funktionen, anstatt Komponenten wiederzuverwenden und werden vom Experiment ausgeschlossen.

Die übrigen EJB-Methoden, für die Antwortzeitvorhersagen durchgeführt werden, sind in Tabelle 4-4 aufgelistet. Zu jeder Methode werden einige der für Vorhersagen relevanten Eigenschaften aufgeführt. Die Source Lines of Code (SLOC) beschreiben die Anzahl der Anweisungen, Kommentare werden hierbei nicht berücksichtigt. Die Anzahl der externen Aufrufe gibt an wie viele Dienste der JPA-Schicht aufgerufen werden. Aufrufe zwischen EJB werden hierbei nicht mitgezählt. Die SLOC, externe Aufrufe und If-/For-Anweisungen aus den aufgerufenen EJB-Methoden werden mitgezählt.

Komponente	**Operation**	**Eingabeparameter**	**SLOC**	**Externe Operations-aufrufe**	**If-Anweisungen**	**For-Schleifen**
CustomerSession	getCustomer	Integer	1	1	0	0
	getInventories	Integer	6	2	1	0
	sellInventory	Integer, long, boolean	17	5	3	0
	validateCustomer	Integer	2	1	0	0
LargeOrderSender-Session	sendOrdersToManufac-turing	List<OrderLine>, int	26	5	0	1
OrderSession	cancelOrder	int	8	2	1	0
	getItem	String	1	1	0	0
	newOrder	int, ShoppingCart, int, boolean, boolean	96	22	8	3

Tabelle 4-4: Für die Evaluierung der Vorhersagegenauigkeit ausgewählte EJB-Methoden und deren Eigenschaften
Quelle: In Anlehnung an Danciu et al. (2015b)

Der Benchmark-Treiber und die Java-EE-Applikation werden für die Durchführung von Antwortzeitmessungen auf unterschiedliche virtuelle Maschinen (VM) verteilt. Die VMs werden auf einem IBM X3755 M3 Server mit 4 AMD Opteron 6172 Prozessoren mit jeweils 12 2.1 GHz Kernen und 256 GB RAM ausgeführt. Als Ablage für die Dateisysteme der VMs dient ein IBM DS3512 Speichernetzwerk. Der physische Server ist über VMWare ESXi 5.1.0 virtualisiert. Als Betriebssystem für die VMs wird openSUSE 12.3 eingesetzt. Jede VM verfügt über 8 virtuelle Prozessoren und 12 GB RAM Hauptspeicher. Die Java-EE-Applikation wird auf einem JBoss 7.1.1 deployed. Antwortzeitvorhersagen werden auf einem Clientrechner durchgeführt.

4.4.3 Durchführung des Experiments

Zur Erfassung des Laufzeitverhaltens der Java-EE-Komponenten wird der Quelltext manuell mit Hilfe des Kieker Frameworks instrumentiert. Während eines Benchmark-Laufs wird eine vordefinierte Last erzeugt und führt dazu, dass EJB-Methoden aufgerufen werden. Die Ausführung des Benchmark-Treibers beginnt mit einer Einschwingphase von vier Minuten in der die simulierten Nutzer schrittweise instanziiert werden. Der Workload bleibt dann für zehn Minuten konstant, bevor die simulierten Nutzer in einem Zeitraum von vier Minuten abgebaut werden. Während der Einschwingphase und dem Abbau der Nutzer durchgeführte Messungen werden nicht berücksichtigt. Während eines Benchmark-Laufs können EJB-Methoden abhängig von der Implementierung mehrere tausend Mal aufgerufen werden.

Für die Ableitung von Informationen über die Parametrisierung werden die EJB-Methoden in einer separaten Iteration instrumentiert. Die Werte der Parameter werden zunächst im Rahmen eines Benchmark-Laufs bei jeder Ausführung der Methode protokolliert. Anschließend werden Durchschnittswerte für die Ausprägung von Boolean- und die Anzahl der Elemente von Listenparameter berechnet.

Mehrere Benchmark-Läufe werden für die Erfassung der Antwortzeit und der Parametrisierung einzelner Komponenten durchgeführt. Aufgrund des standardisierten Workloads sind diese Messläufe miteinander vergleichbar. Vor jedem Lauf werden der Applikationsserver und die Datenbank neu gestartet und initialisiert.

4.4.4 Ergebnisse

Die Ergebnisse des Experiments bei Verwendung des Medians zur Abbildung des Antwortzeitverhaltens von Komponenten sind in Tabelle 4-5 aufgelistet. Die Werte der Tabelle wurden auf zwei Nachkommastellen aufgerundet. Die gemessene Antwortzeit wird als Median der zugrundeliegenden Messwerte berechnet. Die Abweichung berechnet sich als die prozentuale Differenz zwischen der gemessenen und dem simulierten Wert. Die durchschnittliche Abweichung ohne Annotationen beträgt 68% und mit Annotationen 42%. Ohne die Spezifikation von Informationen über die Parametrisierung reicht die Abweichung zwischen 9,84% und 227,01%.

Komponente	Operation	Gemessene Antwortzeit	Ohne Annotationen		Mit Annotationen	
			Simulierte Antwortzeit	Abweichung	Simulierte Antwortzeit	Abweichung
CustomerSession	getCustomer	0,96 ms	0,71 ms	25,60 %	0,71 ms	25,60 %
	getInventories	0,80 ms	0,71 ms	10,63 %	0,71 ms	10,63 %
	sellInventory	0,91 ms	0,73 ms	19,87 %	0,73 ms	19,87 %
	validateCustomer	1,24 ms	0,71 ms	42,41 %	0,71 ms	42,41 %
LargeOrderSenderSession	sendOrdersToManufacturing	12,35 ms	11,13 ms	9,84 %	11,42 ms	7,52 %
OrderSession	cancelOrder	1,88 ms	0,72 ms	61,57 %	0,73 ms	61,13 %
	getItem	0,28 ms	0,71 ms	152,09 %	0,71 ms	152,09 %
	newOrder	1,83 ms	6 ms	227,01 %	2,30 ms	22,57 %

Tabelle 4-5: ***Ergebnisse der Evaluierung bei Verwendung des Medians für die Abbildung des Antwortzeitverhaltens***
Quelle: In Anlehnung an Danciu et al. (2015b)

Die einfachsten Methoden, *getCustomer* und *getItem*, bestehen jeweils nur aus einem Aufruf der Methode *find* des *EntityManager*. Der vorhergesagte Wert entspricht daher dem Median der Antwortzeitmessungen für *find*. Hierbei werden Messungen über unterschiedliche Parametrisierungen verdichtet. Abhängig von der referenzierten Tabelle weist der Aufruf der Methode *find* stark abweichende Antwortzeiten auf. Die Aufrufe innerhalb von *getCustomer* und *getItem* greifen auf bestimmte Tabellen zu. Die Suche eines Datenobjekts vom Typ *Item* weist beispielsweise eine viel geringere Antwortzeit auf, als für andere Typen. Aufgrund der Einfachheit dieser Methoden können keine sinnvollen Annotationen spezifiziert werden und die simulierte Antwortzeit ist für beide Szenarien gleich. Die höchste Präzision, unabhängig von der Verfügbarkeit von Annotationen, wird für die komplexere Methode *sendOrdersToManufacturing* erzielt. Dadurch, dass die Methode nur eine For-Schleife spezifiziert, hat die Verfügbarkeit von Annotationen einen geringen Einfluss. Die niedrigste Präzision bei fehlenden Annotationen wird für die Methode *newOrder* erzielt. In diesem Fall werden 22 JPA-Dienste aufgerufen und es existieren acht If-Anweisungen. Durch die Annahme einer Gleichverteilung der Wahrscheinlichkeit für das Ergebnis von If-Anweisungen kommt es zu einer Abweichung von 227,01%. Bei Hinzunahme von Annotationen wird die Abweichung um ein Zehnfaches verbessert. Bei Methoden die keine If-/For-Anweisungen aufweisen, oder JPA-Komponenten außerhalb dieser Anweisungen aufrufen, wird durch die Hinzunahme von Annotationen keine Verbesserung erzielt.

Die Ergebnisse des Experiments bei Verwendung des 95-Perzentils für die Abbildung des Antwortzeitverhaltens von Komponenten sind in Tabelle 4-6 aufgelistet. Die Werte der Tabelle wurden auch hier auf zwei Nachkommastellen aufgerundet. Die gemessene Antwortzeit wird als 95-Perzentil der zugrundeliegenden Messwerte berechnet. Die durchschnittliche Abweichung ohne Annotationen beträgt 38% und mit Annotationen 46%. Die Vorhersage bei Verwendung des 95-Perzentils weist für die meisten Methoden eine höhere Präzision auf, als bei Verwendung des Median, unabhängig von der Verfügbarkeit von Annotationen. Für die Methode *getItem* wird nun eine noch höhere Antwortzeit des *EntityManagers* angenommen und die Abweichung steigt damit auf 196,69%. Die Methode *newOrder* weist bei der Hinzunahme der Annotationen eine niedrigere Präzision auf als davor. Die Annotationen spezifizieren in diesem Fall eine viel geringere Wahrscheinlichkeit für den Aufruf von JPA-Komponenten und somit wird auch eine geringere Antwortzeit als davor vorhergesagt. Durch die Darstellung der gemessenen Antwortzeit als 95-Perzentil haben Aufrufe, die vom Performancemodell nicht abgebildet werden, eine größere Auswirkung als bei der Verwendung des Median. Für die Methode *sendOrdersToManufacturing* sieht die Annotation eines Listenparameters zwei zusätzliche Iterationen einer Schleife vor und dadurch wird eine höhere Antwortzeit vorhergesagt.

Komponente	Operation	Gemessene Antwortzeit	Ohne Annotationen		Mit Annotationen	
			Simulierte Antwortzeit	Abweichung	Simulierte Antwortzeit	Abweichung
CustomerSession	getCustomer	2,43 ms	2,22 ms	8,63 %	2,22 ms	8,63 %
	getInventories	2,14 ms	2,22 ms	3,92 %	2,22 ms	3,92 %
	sellInventory	2,35 ms	2,28 ms	2,84 %	2,28 ms	2,84 %
	validateCustomer	2,87 ms	2,22 ms	22,71 %	2,22 ms	22,71 %
LargeOrderSenderSession	sendOrdersToManufacturing	19,81 ms	21,92 ms	10,64 %	23,06 ms	16,39 %
OrderSession	cancelOrder	4,83 ms	2,25 ms	53,41 %	2,28 ms	52,83 %
	getItem	0,75 ms	2,22 ms	196,69 %	2,22 ms	196,69 %
	newOrder	14,77 ms	13,34 ms	9,71 %	5,30 ms	64,13 %

Tabelle 4-6: ***Ergebnisse der Evaluierung bei Verwendung des 95-Perzentils für die Abbildung des Antwortzeitverhaltens***
Quelle: In Anlehnung an Danciu et al. (2015b)

Unabhängig vom gewählten Maß der zentralen Tendenz weisen bis auf wenige Fälle die meisten Methoden höhere Antwortzeiten auf als vorhergesagt. Dies liegt vor allem daran, dass im Performancemodell nur eine Untermenge aller Anweisungen und Aufrufe abgebildet und mit Antwortzeiten hinterlegt werden. Anweisungen wie beispielsweise der Aufruf von Konstruktoren werden nicht berücksichtigt, tragen jedoch zur gemessenen Antwortzeit bei.

Die Verfügbarkeit von Annotationen trägt allgemein zu einer Verbesserung der Präzision der Vorhersagen bei. Einzig für die Methoden *newOrder* und *sendOrdersToManufacturing* im Szenario mit 95-Perzentilen ist eine Verschlechterung festzustellen. Für *newOrder* wird durch die Hinzunahme der Annotationen eine noch geringere Antwortzeit vorhergesagt. Bei *sendOrdersToManufacturing* umgekehrt.

Die Abweichungen zwischen den simulierten und den gemessenen Antwortzeiten für die Komponente OrderSession können durch die Verteilung der einzelnen Messwerte erklärt werden. Eine Zusammenfassung der Lage- und Streuungsmaße über die gemessenen Antwortzeiten ist in Abbildung 4-33 mit Hilfe einer logarithmischen Skala dargestellt. Wie aus dieser Darstellung hervorgeht, kann für alle Methoden eine signifikante Streuung der gemessenen Antwortzeiten festgestellt werden. Demgegenüber stehen simulierte Antwortzeiten (bei Verfügbarkeit von Annotationen), die eine sehr geringe Varianz aufweisen und als aggregierter Wert zurückgemeldet werden. Eine Varianz ergibt sich bei simulierten Antwortzeiten nur aufgrund von If-Anweisungen. Trotz der teilweise großen Abweichungen in der Prozentdarstellung kann festgestellt werden, dass sich die simulierten Werte innerhalb der Streuung der Messwerte befinden.

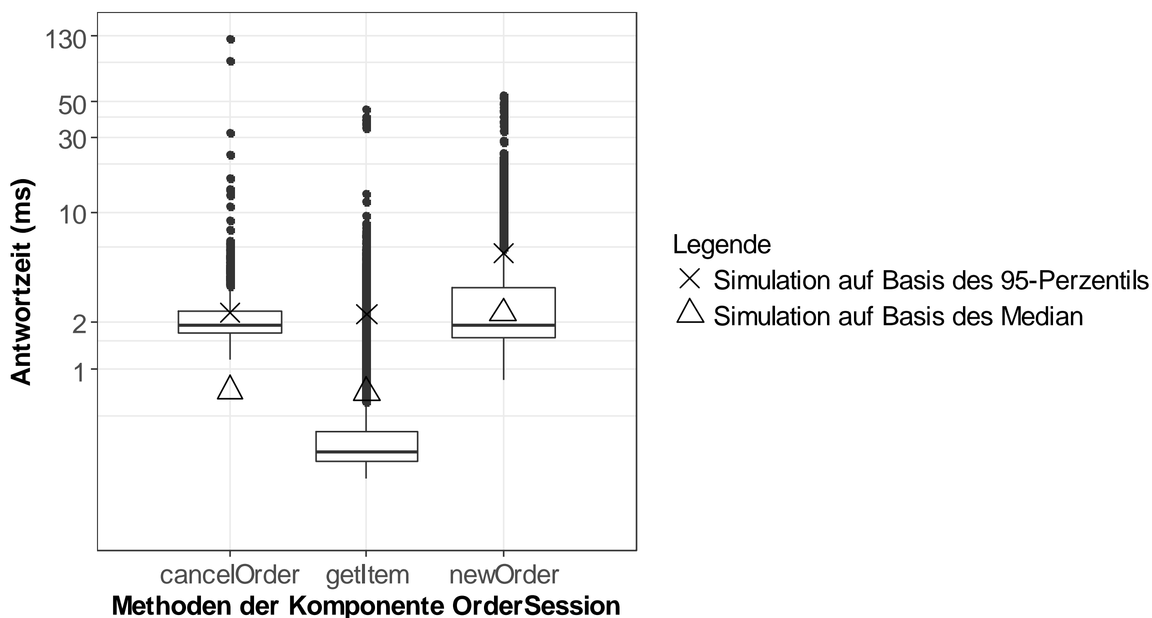


Abbildung 4-33: Verteilung der gemessenen Antwortzeiten am Beispiel der Komponente OrderSession
Quelle: Eigene Darstellung

Aus der Evaluation lassen sich folgende zwei relevante Einflussfaktoren auf die Präzision der Antwortzeitvorhersagen ableiten:

- Anteil der nicht modellierten Anweisungen einer Methode: Eine größere Menge an nicht modellierten Anweisungen kann unter Umständen dazu führen, dass eine viel zu geringe Antwortzeit vorhergesagt wird.
- Schiefe der Verteilungsfunktion von Antwortzeitmessungen eines wiederverwendeten Dienstes: Abhängig von der Schiefe einer Verteilungsfunktion kann die Eignung der verschiedenen Maße der zentralen Tendenz für die Verdichtung von Antwortzeitmessungen variieren.

4.4.5 Validität der Ergebnisse

Verschiedene Aspekte des hier beschriebenen Experiments können die Aussagekraft der Ergebnisse beeinflussen. Die Validität der Ergebnisse nach Wohlin et al. (2012, S. 102) werden im Folgenden beschrieben.

Statistische Validität

Die statistische Validität der Ergebnisse hängt stark von dem Verfahren und den Instrumenten zur Messung und der Simulation der Antwortzeiten ab. Der in der Industrie anerkannte SPECjEnterprise2010-Benchmark bietet einen standardisierten Workload und macht unterschiedliche Messläufe miteinander vergleichbar. Messungen werden mit Hilfe des Kieker-Frameworks durchgeführt. Kieker weist einen relativ kleinen Overhead auf und wurde in unterschiedlichen wissenschaftlichen Arbeiten eingesetzt (Rohr et al. 2010; Heger et al. 2013; Waller/Hasselbring 2013). Während einem Benchmark-Lauf über zehn Minuten werden abhängig von der Methode Zehntausende Messungen durchgeführt. Nur Programme, die als notwendig für die Laufzeit des Benchmarks erachtet wurden, liefen während den Messläufen. Damit der Overhead der Instrumentierung sich nicht auf Messungen auf einer höheren Ebene auswirkt, wurden die unterschiedlichen Schichten einzeln instrumentiert. Messungen aus der Einschwingphase wurden nicht berücksichtigt. Die Simulation von Antwortzeiten wird mit Hilfe von SimuCom durchgeführt. Das Framework wird von zahlreichen wissenschaftlichen Arbeiten für die Simulation von PCM-Instanzen eingesetzt (Brunnert et al. 2013; Brunnert/Krcmar 2015) und weist eine ähnliche Genauigkeit auf wie vergleichbare Werkzeuge (Koziolek et al. 2013). Während eines Simulationslaufs wird auch sichergestellt, dass mehrere tausend Messungen durchgeführt werden.

Interne Validität

Die interne Validität der Ergebnisse wird durch den hohen Automatisierungsgrad der Verfahren zur Generierung von Workload, Messungen von Antwortzeiten, Berechnung von Maßen der zentralen Tendenz und Simulation unterstützt.

Konstruktvalidität

Die Konstruktvalidität der Ergebnisse kann durch Aspekte des Entwurfs und durch soziale Aspekte beeinflusst werden (Wohlin et al. 2012, S. 108f). Die Adressierung der Aspekte des Entwurfs wird im Folgenden beschrieben:

- Beschreibung der Konstrukte: Die grundlegenden Konstrukte dieses Experiments sind die Beeinflussung, Messung, Evaluation und Vorhersage des Antwortzeitverhaltens von Komponenten. Eine detaillierte Beschreibung dieser Konzepte existiert in der Literatur und wurde als Grundlage für dieses Experiment verwendet.
- Betrachtung eines einziges Objekts: Im Rahmen des Experiments werden Vorhersagen für mehr als eine Komponente durchgeführt. Die betracheteten Komponenten verwenden eine vielzahl von Diensten wieder.
- Betrachtung einer einzigen Methode: Zur Darstellung des Antwortzeitverhaltens von Diensten werden der Median und das 95-Perzentil verwendet. Das Experiement verwendet unterschiedliche Darstellungen für das Antwortzeitverhalten und berücksichtigt die Verfügbarkeit von Informationen über die Parametrisierung von Methoden.
- Präsenz und Intensität von Konstrukten: Im Rahmen des Experiments wird neben der Exsitenz von Faktoren, wie Informationenen über die Parametrisierung oder If-/For-Anweisungen, auch deren Ausprägung adressiert.
- Interaktion zwischen Behandlungen bzw. zwischen Messung und Behandlung: Als Subjekte des Experiment dienen Technologien, keine Versuchspersonen. Der Zustand der Komponenten und der Experimentumgebung wird vor jeder Iteration neu initialisiert.
- Eingeschränkte Verallgemeinbarkeit auf andere Konstrukte: Im Rahmen dieses Experiments führt die Behandlung zu keiner Veränderung eines Zustands. Andere Konstrukte, wie z.B. die Ressourcenauslastung von Komponenten, werden im Rahmen dieser Evaluation nicht betrachtet.

Soziale Aspekte haben keinen Einfluss auf die Validität der Ergebnisse, da keine Versuchspersonen, sondern Technologien eingesetzt werden.

Externe Validität

Die Generalisierbarkeit der Ergebnisse auf die Praxis kann durch folgende Aspekte beeinflusst werden (Wohlin et al. 2012, S. 110):

- Interaktion zwischen Selektion und Treatment: Das Ziel des SPECjEnterprise2010-Benchmarks ist die Bereitstellung eines realistischen Workloads für Java-Applikationsserver bei einer großen Abdeckung der Java-EE-Spezifikation und unter Verwendung der wichtigsten Programmierpraktiken aus der Industrie (SPEC 2014). Es wird daher angenommen, dass die untersuchten Komponenten große Ähnlichkeit mit Implementierungen aus der Industrie aufweisen.
- Interaktion zwischen Umgebung und Treatment: Die Verfügbarkeit einer ähnlichen Umgebung, im Sinne der Java-EE-Anwendung, des Applilationsservers, der

Virtualisierungstechnologie, der Hardwareressourcen und der IDE kann auch in der Praxis angenommen werden.

- Interaktion zwischen Vergangenheit und Treatment: Das hier beschriebene technologiebasierte Experiment wird von Ereignissen aus der Vergangenheit nicht beeinflusst.

4.5 Zusammenfassung und Ausblick

In diesem Kapitel wurde das Konzept und die Implementierung eines Ansatzes für die Unterstützung für Performancebewusstsein in Java-EE-Entwicklungsumgebungen beschrieben. Die durchschnittliche Abweichung der Antwortzeitvorhersagen wurde im Rahmen eines Experiments untersucht und lag bei dem Einsatz von 95-Perzentilen und Annotationen bei 38%. Die Limitationen des Ansatzes für Performancebewusstsein werden im Abschnitt 4.5.1 und zukünftige Forschungsrichtungen im Abschnitt 4.5.2 beschrieben.

4.5.1 Limitationen

Die Limitationen des Ansatzes hinsichtlich der Erstellung von Performancemodellen, deren Simulation und der Evaluation werden im Folgenden beschrieben.

Modellerstellung

Die Erstellung von Performancemodellen berücksichtigt vor allem die Wiederverwendung von Diensten und Kontrollstrukturen. Diese werden in Form von Anweisungen aus Klassen und Methodendeklarationen extrahiert. Die Java-EE-Spezifikation unterstützt jedoch auch die Steuerung des Kontrollflusses über Annotationen. Über die Methodenannotation @PostConstruct kann beispielsweise der Aufruf einer Methode nach der Initialisierung einer Komponente spezifiziert werden. Die entsprechende Methode muss dafür nicht explizit aufgerufen werden. Dieser Kontrollfluss wird durch den Ansatz nicht erkannt und auch nicht abgebildet. Neben Annotationen können auch Konfigurationsdateien für die Einbindung von Komponenten und Operationen eingesetzt werden. Diese Arten von Abhängigkeiten werden durch den Ansatz auch nicht berücksichtigt.

Java-Anweisungen können ineinander verschachtelt werden. Beispiele hierfür sind der Aufruf einer Methode innerhalb einer If-Abfrage oder die Übergabe eines Parameters in Form eines Methodenaufrufs. Die aktuelle Implementierung würde diese verschachtelten Methodenaufrufe nicht abbilden. Der Aufruf von Konstruktoren kann auch einen relevanten Einfluss auf die Antwortzeit einer Komponente haben. In der aktuellen Implementierung des Ansatzes werden weder die Deklaration von Konstruktoren als RDSEFF noch der Aufruf von Konstruktoren als ExternalCallAction abgebildet.

Die Wiederverwendung von Diensten wird aktuell nur auf der Ebene von Methodenaufrufen identifiziert. Komponenten können ihre Dienste jedoch auch als Web Service oder REST-Schnittstelle anbieten. Andere Komponenten rufen diese anhand eines Uniform Resource Locator (URL) auf. Die Verarbeitung entsprechender Aufrufe und die Zuordnung auf entsprechende Komponenten und Dienste wird aktuell nicht unterstützt.

Namen von Repository-Komponenten bestehen aus dem Paket und dem Klassennamen. Von einem vollqualifizierten Paketnamen übernimmt SoMoX jedoch nur die letztem zwei Ebenen. Zwei Komponenten mit gleichem Namen, aus unterschiedlichen Paketen, die jedoch gleiche Endungen haben, wären dadurch nicht mehr eindeutig identifizierbar.

Implementiert eine Klasse mehrere Schnittstellen, welche dieselbe Methode anbieten, so werden auch mehrere RDSEFFs mit demselben Namen angelegt. Da die RDSEFFs dieselbe Implementierung aufweisen, führt dies jedoch zu keinen Fehlern.

Über einen Konfigurationsparameter beeinflusst der Entwickler die Klassifizierung von Komponenten, für die sowohl der Quelltext als auch Antwortzeitmessungen existieren. Im Beispiel aus Listing 4-1 könnte die Komponente DataService bei Verfügbarkeit von Antwortzeitmessungen sowohl als intern als auch extern klassifiziert werden. Die Eignung der zwei Klassifizierungen kann von der jeweiligen Komponente und zusätzlichen Faktoren abhängen. Die Entscheidung muss jedoch manuell durch den Entwickler getroffen werden und gilt für alle Komponenten.

Simulation

Bei der Simulation des Antwortzeitverhaltens von Komponenten mit Hilfe von SimuCom wird für jede Spezifikation eines RDSEFF-Aufrufs ein Sensor angelegt. Der Sensor ist für die Messung und Speicherung der Antwortzeit der Aufrufe während der Simulation zuständig. Das SensorFramework von SimuCom verwendet für die Benennung der Sensoren die Namen der entsprechenden RDSEFF und fügt als Suffix noch Zahlen an. Die Namen der RDSEFF wiederum entsprechen den Namen der abgebildeten Java-Methoden. Nach der Durchführung der Simulation werden anhand der Namen der Sensoren die Werte ausgelesen und Java-Methoden zugeordnet. Aufgrund der künstlich eingefügten Ziffern könnten Inkonsistenzen entstehen. Beispielsweise können Methodennamen auch Ziffern enthalten und falsch zugeordnet werden.

Evaluation

Informationen über die Parametrisierung der Methodenaufrufe wurden im Rahmen des Experiments mit Hilfe von Annotationen spezifiziert. Die Parametrisierung der relevanten Methoden wurde in einem separaten Benchmark-Lauf für jeden Aufruf erhoben und anschließend als Durchschnittswert zusammengefasst. Damit repräsentieren die spezifizierten Werte die Realität relativ genau ab. In der Praxis würden diese Werte evtl. nicht zur Verfügung stehen und sie müssten geschätzt werden.

Bei der Selektion von Komponenten und Diensten, deren Antwortzeit im Rahmen des Experiments vorhergesagt werden soll, wurden Datenzugriffsdienste ausgeschlossen. Diese Dienste führen direkte Datenbankabfragen aus und deren Antwortzeit hängt stark von den adressieren Tabellen, Spalten und Zeilen ab (siehe Abbildung 4-34). Die Parametrisierung der Abfragen wird durch den Ansatz nicht abgebildet und Antwortzeitvorhersagen für diese Dienste können daher ungenau sein. Auch der Aufruf der Methode find des EntityManager hängt von der adressierten Tabelle ab. Um die Selektion nicht zu stark einzugrenzen, wurden aufrufende EJBs jedoch nicht ausgeschlossen.

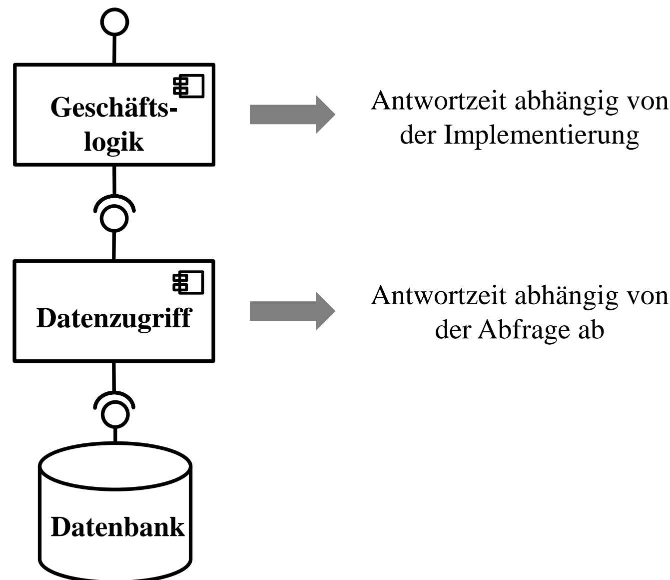


Abbildung 4-34: Unterscheidung zwischen der Implementierung der Geschäftslogik und des Datenzugriffs
Quelle: Eigene Darstellung

Das Antwortzeitverhalten von Diensten, welche ein spezifisches Zugriffsmuster kapseln, weist geringere Schwankungen auf und kann besser durch Maße der zentralen Tendenz abgebildet werden. Das Verhalten der aufrufenden Komponenten hängt wiederum stärker von deren Implementierung ab. Die Implementierung der Komponenten wird durch den Ansatz detaillierter abgebildet und Antwortzeiten können präziser vorhergesagt werden.

4.5.2 Zukünftige Forschung

Zukünftige Forschung sollte Aspekte der Modellierung, der Automatisierung und der Interaktion adressieren. Die Steuerung des Kontrollflusses über Java-EE-Annotationen könnte explizit identifiziert und im Performancemodell abgebildet werden. Hierfür könnten Erweiterungen des PCM-Metamodells notwendig sein. Die Wiederverwendung von Web Services und REST-Schnittstellen könnte durch die Verarbeitung von URL-Aufrufe und deren Zuordnung zu Diensten auch abgebildet werden. Puffermechanismen könnten detaillierter abgebildet werden, indem auch der Typ des Datenobjekts berücksichtigt wird. Durch die Simulation der PCM-Instanzen mit Hilfe von EventSim (Merkle/Henss 2011) könnten Inkonsistenzen bei der Benennung von Sensoren vermieden werden. EventSim könnte auch zu einer Beschleunigung der Simulation beitragen. Die Möglichkeit, Informationen über die Parametrisierung zu spezifizieren, könnte zukünftig detaillierter untersucht werden. Aktuell kann der Entwickler nur Boolean- und Listenparameter über jeweils eine Eigenschaft beschreiben. Werden die Parameter nicht deklariert, sondern über einen Aufruf ermittelt, können keine Annotationen spezifiziert werden. Einerseits könnten mehrere Datentypen und Eigenschaften abgedeckt werden. Andererseits müsste die Notation so erweitert werden, dass auch kompliziertere Konstrukte annotiert werden können.

Puffermechanismen, die manuell in der Performancedatenbank konfiguriert werden müssen, könnten während dem Monitoring automatisch identifiziert werden. Die Evaluation des Ansatzes hat gezeigt, dass die Auswahl einer Abbildung des Antwortzeitverhaltens die Vorhersagegenauigkeit stark beeinflussen kann. Abhängig von der Verteilungsfunktion der Messungen

könnte die Performancedatenbank automatisch die passendste Art der Verdichtung auswählen. Manuell wird aktuell auch noch die Einstellung für die Klassifizierung von Komponenten, für die sowohl der Quelltext als auch Antwortzeitmessungen existieren, vorgenommen. Zukünftige Forschung könnte untersuchen, wie diese Entscheidung für jede einzelne Komponente automatisiert getroffen werden kann. Eine andere wichtige Erkenntnis aus der Evaluation ist, dass die Vorhersagegenauigkeit von der Anzahl der nicht abgebildeten Anweisungen und der Schiefe der Verteilungsfunktionen von Messungen abhängt. Eine zukünftige Entwicklung könnte basierend auf diesen Informationen dem Entwickler eine Einschätzung über die erwartete Genauigkeit mitteilen.

5 Einfluss von Performancebewusstsein auf die Antwortzeit von Softwarekomponenten

In diesem Kapitel wird die Evaluation des Ansatzes für Performancebewusstsein beschrieben. Im Rahmen eines Experiments wird der Einfluss des Ansatzes auf das Antwortzeiterhalten von Java-EE-Komponenten untersucht. Die Struktur dieses Kapitels richtet sich nach Jedlitschka/Pfahl (2005). Im Abschnitt 5.1 wird die Problemstellung und die Zielsetzung dieser Untersuchung dargestellt. Die Gestaltung eines Experiments zur Untersuchung des Einflusses wird im Abschnitt 5.2 vorgestellt. Die Durchführung des Experiments und die Auswertung der Ergebnisse werden in den Abschnitten 5.3 bis 5.8 beschrieben. Abschließend werden die Ergebnisse interpretiert und das Experiment zusammengefasst (siehe Abschnitt 5.9 und 5.10).

Die Gestaltung, Durchführung und die Ergebnisse des in diesem Kapitel beschriebenen Experiments wurden in (Danciu/Krcmar 2018) veröffentlicht.

5.1 Motivation

5.1.1 Problemstellung

Eine kontinuierliche Evaluierung der Performance von Anwendungen kann die Produktivität von Entwicklern beeinträchtigen (Brunnert et al. 2015b, S. 16). Ansätze für Performancebewusstsein zielen auf die Bereitstellung von Einsichten über die Performance ab. Damit sollen Entwickler dabei unterstützt werden, die Performance von Anwendungen ggf. zu verbessern. Existierende Ansätze für Performancebewusstsein werden jedoch vor allem hinsichtlich ihrer funktionalen Korrektheit evaluiert (Heger et al. 2013; Bulej et al. 2012b). Der Einfluss dieser Ansätze auf Entwickler und die Qualität ihrer Implementierungen wird dabei ignoriert. Aus den Evaluationen der Ansätze geht somit nicht hervor, ob diese einen positiven Einfluss auf die Performance haben und wie hoch dieser ist.

Horký et al. (2015) berichten über den misslungenen Versuch, den Einfluss ihres Ansatzes auf die Antwortzeit von Implementierungen mit Hilfe von Studenten zu untersuchen. Aufgrund von wenigen und teilweise stark abweichende Rückmeldungen konnte keine signifikante Verbesserung durch die Verfügbarkeit des Ansatzes festgestellt werden. Ersatzweise wurde der Nutzen des Ansatzes anhand von Szenarien erklärt.

Der Ansatz für Performancebewusstsein wurde in Kapitel 4 nur hinsichtlich der Vorhersagegenauigkeit evaluiert (Danciu et al. 2015b). In diesem Kapitel wird die Auswirkung der Verfügbarkeit dieses Ansatzes auf die Entwickler und die dadurch resultierende Antwortzeit von Java-EE-Komponenten untersucht.

5.1.2 Forschungsziele

Das Ziel dieser Untersuchung ist die Quantifizierung des Einflusses von Performancebewusstsein von Entwicklern auf die Antwortzeit der implementierten Software. Genauer soll der Einfluss des Ansatzes aus Kapitel 4 auf das Antwortzeitverhalten der implementierten Java-EE-Komponenten untersucht werden.

Die Forschungsziele werden mit Hilfe des Goal-Question-Metric-Ansatzes (GQM) von Basili et al. (1994) unter Anwendung der Vorlage aus (Wohlin et al. 2012, S. 88; Ciolkowski et al. 1997) strukturiert:

- Analyse des Ansatzes für Performancebewusstsein
- zum Zwecke der Evaluierung
- in Bezug auf die Effektivität
- aus der Sicht des Forschers
- im Kontext der Bearbeitung einer Aufgabe in einer Laborumgebung.

5.1.3 Kontext

Die Untersuchung wird mit Hilfe von Versuchspersonen durchgeführt. Hierfür werden:

- Studenten der Fakultät für Informatik der Technischen Universität München, unabhängig von ihrem Studiengang,
- wissenschaftliche Mitarbeiter der Technischen Universität München und der fortiss GmbH aus dem Bereich Informatik und
- Softwareentwickler, unabhängig von ihrer Fimenzugehörigkeit

eingesetzt. Teilnehmer müssen mindestens grundlegende Programmierkenntnisse besitzen, die sie beispielsweise aus Kursen erworben haben. Für die Teilnahme existieren keine Anreizmaßnahmen. Versuchspersonen werden eingeladen und nehmen freiwillig teil.

Der Zeitrahmen und die Umgebung zur Bearbeitung der Aufgabe stellen wichtige Einflussfaktoren auf die erzielten Ergebnisse dar. Je mehr Zeit zur Verfügung steht, desto bessere Ergebnisse können erzielt werden. Andererseits könnte der spezifizierte Zeitrahmen nicht effektiv genutzt werden. Zur Kontrolle dieses Faktors sollen Versuchspersonen die Aufgabe unter Beaufsichtigung innerhalb eines kurzen Zeitraums bearbeiten. Aufgrund der freiwilligen Teilnahme und zur Vermeidung einer abschreckenden Wirkung darf die Untersuchung nicht länger als eine Stunde dauern. Die Produktivität einer Versuchsperson wird unter anderem auch durch die verfügbare Rechenleistung beeinflusst. Daher sollen die Versuchspersonen ähnliche Ressourcen nutzen. Die Untersuchung erfolgt daher in dem Computerlabor der fortiss GmbH.

Das Experiment soll untersuchen, ob Versuchspersonen, die den Ansatz für Performancebewusstsein nutzen, Komponenten mit einem besseren Antwortzeitverhalten implementieren, als Versuchspersonen, die den Ansatz nicht nutzen. Als Implementierungsaktivität kommt sowohl die Neuentwicklung von Komponenten als auch die Weiterentwicklung bzw. Wartung von existierenden Komponenten in Betracht. Die Entwicklung neuer Komponenten im Rahmen eines Experiments impliziert aufgrund funktionaler Anforderungen einige Risiken. Die Implementierung einer Versuchsperson kann nur dann bewertet werden, wenn diese die funktionalen Anforderungen erfüllt. Im Kontext komplexer Unternehmensanwendungen und der Betrachtung von Wiederverwendung zwischen Komponenten können innerhalb eines kurzen Zeitraums nur einfache Funktionen implementiert werden. Dabei besteht das Risiko, dass keine signifikanten Unterschiede zwischen den Implementierungen festzustellen sind. Als Aufgabenstellung fordern Horký et al. (2015) die Implementierung eines Algorithmus zum Einlesen von

XML-Dokumenten und berichten von einem geringen Anteil funktionierender Ergebnisse. Die Implementierung von Komponenten unter Einsatz komplexer Frameworks und bei Wiederverwendung von Komponenten stellt dabei eine noch größere Hürde dar. Inhalt der Aufgabe soll daher die Wartung von Komponenten einer beispielhaften Java-EE-Anwendung sein. Versuchspersonen erhalten eine existierende Implementierung und müssen das Antwortzeitverhalten optimieren (vgl. Rothlisberger et al. 2012).

Ausgehend von dem hier beschriebenen Kontext stellt die Untersuchung ein kontrolliertes Experiment dar.

5.2 Gestaltung des Experiments

Die Untersuchung wird in Form eines Experiments nach Wohlin et al. (2012) durchgeführt. Die Gestaltung des Experiments ist an Menschen orientiert (Wohlin et al. 2012, S. 76) und wird im Folgenden beschrieben.

5.2.1 Ziele, Hypothesen und Variablen des Experiments

Die umgesetzten Optimierungsmaßnahmen der Versuchspersonen können auf unterschiedliche Arten quantifiziert werden. Wie bei Horký et al. (2015) könnten Implementierungen ausgeführt und die erzielten Antwortzeiten gemessen werden. Im Kontext komplexer Unternehmensanwendungen impliziert dieses Vorgehen einige Nachteile. Um Fehler und Interferenzen auszuschließen müssten je Implementierung mehrere Messläufe durchgeführt werden. Einzelne Optimierungsmaßnahmen können abhängig vom Inhalt der Datenbank und vom Workload eine unterschiedliche Auswirkung aufweisen. Aus diesem Grund werden die Ergebnisse der Versuchspersonen anhand der Anzahl der behobenen Performancebugs quantifiziert.

Untergeordnete Forschungsfrage 1: Erzielen Teilnehmer durch die Nutzung des Plugins bessere Ergebnisse?

Hauptziel des Experiments ist die Untersuchung, ob Versuchspersonen durch die Nutzung des Ansatzes für Performancebewusstsein mehr Performancebugs beheben, als Versuchspersonen, die den Ansatz nicht nutzen (siehe Abbildung 5-1). Performancebugs sind ineffiziente Codesequenzen, die eine signifikante Verschlechterung der Performance herbeiführen (Jin et al. 2012; Nistor et al. 2013a). Nach Jin et al. (2012) können diese durch relativ einfache Quelltextänderungen beseitigt werden.

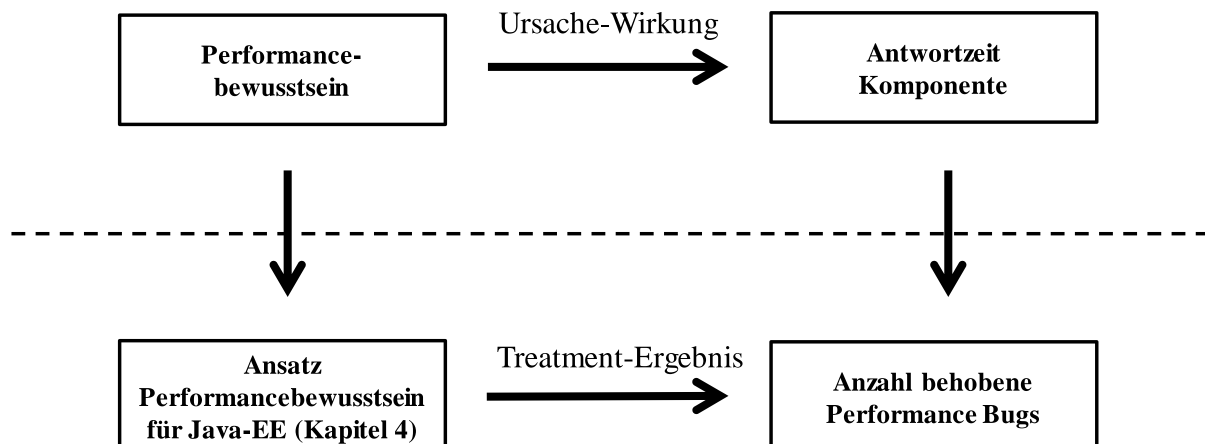


Abbildung 5-1: ***Überblick über die Gestaltung des Experiments***
Quelle: In Anlehnung an Wohlin et al. (2012, S. 74)

Ausgehend von dem Ziel des Experiments werden folgende Hypothesen formuliert:

- H_{1-0}: Die Nutzung des Ansatzes für Performancebewusstsein hat keinen Einfluss auf die Anzahl der Performancebugs, die während der Optimierung von Komponenten identifiziert werden.
- H_{1-1}: Die Nutzung des Ansatzes für Performancebewusstsein hat einen positiven Einfluss auf die Anzahl der Performancebugs, die während der Optimierung von Komponenten identifiziert werden.

Die unabhängige Variable des Experiments ist die Verfügbarkeit des Ansatzes in der Entwicklungsumgebung der Versuchsperson. Die Abhängige Variable ist die Anzahl der behobenen Performancebugs.

Untergeordnete Forschungsfrage 2: Wie verbringen Teilnehmer die Zeit während der Bearbeitung der Optimierungsaufgabe?

Nach Sanchez et al. (2015) können häufige Kontextwechsel und die Fragmentierung der Arbeitsabläufe einen negativen Einfluss auf die Produktivität von Entwicklern während der Bearbeitung von Wartungsaufgaben haben. Nachdem Entwickler die IDE verlassen und dann wieder zurückkehren, benötigen diese Zeit, um ihre Aktivitäten wieder aufzunehmen (Minelli et al. 2015). Zur Quantifizierung der Anzahl der Unterbrechungen eines Probanden wird erfasst, wie oft die IDE verlassen wird, wie viel Zeit außerhalb der IDE verbracht wird, wie viel Zeit für die Navigation innerhalb der IDE aufgewendet wird und wie oft Java-Klassen besucht werden. Die für Navigation aufgewendete Zeit wird als die Dauer der Sequenzen von Workbench-Interaktionen, die in einem Zeitintervall kleiner als die Reaktionszeit von einer Sekunde (Minelli et al. 2015) aufeinanderfolgen, quantifiziert. Intervalle zwischen Ereignissen, die mehr als eine

Sekunde auseinander liegen, werden dem Prozess des Verstehens oder Editierens zugeschrieben. Für die Untersuchung der zweiten Forschungsfrage wird folgende Hypothese aufgestellt:

- $H_{2\text{-}0}$: Die Nutzung des Ansatzes für Performancebewusstsein hat keinen Einfluss auf den Anteil der Zeit, die Versuchspersonen während der Optimierung von Komponenten innerhalb ihrer IDE verbringen.
- $H_{2\text{-}1}$: Die Nutzung des Ansatzes für Performancebewusstsein hat einen positiven Einfluss auf den Anteil der Zeit, die Versuchspersonen während der Optimierung von Komponenten innerhalb ihrer IDE verbringen.

Untergeordnete Forschungsfrage 3: Wie beeinflusst ein methodisches Vorgehen die Ergebnisse der Teilnehmer?

Robillard et al. (2004) analysieren die Auswirkungen eines methodischen Vorgehens bei der Untersuchung von Quelltext auf die Effektivität von Entwicklern während der Bearbeitung einer Wartungsaufgabe. Die Autoren definieren dabei ein methodisches Vorgehen als die Anwendung von Suchen auf Basis von Querverweisen und Stickwörtern. Im Gegensatz dazu beruht ein opportunistischer Ansatz auf Scrollen und Browsen. Basierend auf den Definitionen von Robillard et al. (2004) wird die methodische Untersuchung des Quelltext als die Anzahl der untersuchten Klassen, die für die Behebung von Performancebugs relevant sind, quantifiziert. Für die Untersuchung der dritten Forschungsfrage wird folgende Hypothese aufgestellt:

- $H_{3\text{-}0}$: Die Nutzung des Ansatzes für Performancebewusstsein hat keinen Einfluss auf die Anzahl der untersuchten Klassen, die für die Behebung von Performancebugs relevant sind.
- $H_{3\text{-}1}$: Die Nutzung des Ansatzes für Performancebewusstsein hat einen positiven Einfluss auf die Anzahl der untersuchten Klassen, die für die Behebung von Performancebugs relevant sind.

5.2.2 Aufbau des Experiments

Durch die Spezifikation der Verfügbarkeit des Ansatzes als unabhängige Variable ergeben sich zwei Treatments. Die Rechner des Computerlabors werden mit Java EE IDEs ausgestattet. Für eine Teilmenge der IDEs ist der Ansatz für Performancebewusstsein verfügbar. Bei den restlichen IDEs ist das Plugin deaktiviert. Jeder der Arbeitsplätze verfügt über eine individuelle Nummer. Durch die Zuweisung zu einem Arbeitsplatz erfolgt die Zuordnung zur Test- bzw. Kontrollgruppe.

Aufgrund der Anzahl der verfügbaren Rechner und der Verfügbarkeit der freiwillig gemeldeten Versuchspersonen sind zur Durchführung des Experiments mehrere Iterationen notwendig. Die Anzahl der Teilnehmer kann dabei zwischen den Iterationen schwanken. Dieselbe Versuchsperson kann nur einmalig am Experiment teilnehmen. Die der Zuordnung von Versuchspersonen zu Gruppen zugrundeliegenden Prinzipien werden im Folgenden beschrieben.

Randomisierung

Die Zuordnung von Versuchspersonen zu Arbeitsplätzen erfolgt über ein Losverfahren. Unmittelbar vor Beginn des Experiments wählen Versuchspersonen nacheinander jeweils eine

Nummer bzw. einen Arbeitsplatz. Bei der Wahl sind sich die Versuchspersonen der Zugehörigkeit einer Gruppe nicht bewusst.

Blockbildung

Es wird davon ausgegangen, dass die Leistung einer Versuchsperson von ihrer Erfahrung in der Softwareentwicklung beeinflusst wird. Zur Kontrolle dieses Faktors wird Blockbildung angewendet. Vor der Durchführung des Experiments ist die Ausprägung dieses Faktors bei den Versuchspersonen nicht bekannt. Eine detaillierte Erhebung wird erst am Ende des Experiments durchgeführt. Als Indikator für die Erfahrung wird vor dem Experiment das Arbeitsverhältnis der Versuchsperson angenommen. Hierbei wird grob zwischen Studenten (Bachelor und Master) und Berufserfahrenen (wissenschaftliche Mitarbeiter und Softwareentwickler) unterschieden. Eine detailliertere Unterscheidung, z.B. in Bachelor- und Masterstudenten wäre nicht unbedingt besser geeignet. Wie von Wohlin et al. (2012, S. 114) vorgeschlagen, werden die Teilnehmer aus den zwei Kategorien gleichmäßig auf die Test- und Kontrollgruppe aufgeteilt.

Durch die Durchführung mehrerer Wiederholungen mit unterschiedlicher Teilnehmerzahl muss die Möglichkeit zur Blockbildung nicht unbedingt für jede Iteration gegeben sein. Bei einer Anzahl von zwei Versuchspersonen mit unterschiedlicher Erfahrung kann beispielsweise keine Blockbildung durchgeführt werden. Für die betroffenen Iterationen wird eine reine Randomisierung durchgeführt. Bei einer darauffolgenden Iteration wird eine Wiederherstellung des Gleichgewichts angestrebt. Beispielsweise dadurch, dass erneut zwei Versuchspersonen mit unterschiedlichen Erfahrungen ohne Losverfahren entgegengesetzt zu der vorherigen Iteration zugeordnet werden.

Ausbalancierung

Insgesamt sollen die Test- und Kontrollgruppe dieselbe Anzahl an Versuchspersonen aufweisen. Innerhalb einer Iteration muss diese Möglichkeit zur Ausbalancierung nicht unbedingt gegeben sein. Bei einer darauffolgenden Iteration wird die Wiederherstellung des Gleichgewichts angestrebt.

Das Experiment entspricht dem Standardtyp mit einem Faktor und zwei Treatments (Wohlin et al. 2012, S. 95). Die Art der Unterstützung von Performancebewusstsein repräsentiert den Faktor und die zwei Entwicklungsumgebungen, mit bzw. ohne aktiviertes Plugin, repräsentieren die Treatments. Nach Wohlin et al. (2012) existieren für diese Kategorie von Experiment folgende Gestaltungsarten:

- Vollständige Randomisierung (engl. completely randomized design): Bei der auch *Inter-Subject* genannten Gestaltung wird jede Versuchsperson genau einem Treatment zugeordnet.
- Paarweiser Vergleich (engl. paired comparison design): Bei der auch *Within-Subject* genannten Gestaltung wird jede Versuchsperson beiden Treatments zugeordnet.

Der Vorteil des paarweisen Vergleichs besteht in der Erhebung der doppelten Menge an Stichproben. Nach Kitchenham et al. (2003) können bei dieser Gestaltung jedoch Lerneffekte auftreten und daher müssen unterschiedliche Versuchsobjekte für die zwei Treatments vorausgesetzt werden. Aufgrund der Komplexität der Aufgabenstellung, der kurzen Dauer des

Experiments und der verfügbaren Beispielanwendungen wird die Alternative des paarweisen Vergleichs verworfen und als Gestaltung die vollständige Randomisierung ausgewählt. Zum Testen der Hypothesen dieser Gestaltung eignen sich der t-Test und der Mann-Whitney-Test (Wohlin et al. 2012, S. 96).

5.2.3 Auswahl von Versuchspersonen

Versuchspersonen werden im Experiment aufgefordert Änderungen an Java-EE-Komponenten vorzunehmen. Diese müssen daher mindestens grundlegende Programmierkenntnisse besitzen. Dadurch, dass eine existierende Implementierung von Komponenten zur Verfügung gestellt wird und diese lediglich optimiert werden muss, werden spezielle Kenntnisse über Java, Java EE oder JPA nicht vorausgesetzt. Die Erfüllung der Mindestanforderung wird erstens durch die Adressierung von Entwicklern und von Studenten oder wissenschaftlichen Mitarbeitern aus dem Bereich Informatik sichergestellt. Zusätzlich wird vor der Zulassung zur Teilnahme über ein informelles Gespräch geprüft, ob eine Person mindestens eine Vorlesung im Bereich der Softwareentwicklung absolviert und schon erste praktische Erfahrung gesammelt hat.

Die adressierten Studenten belegen eines der Studiengänge Informatik, Wirtschaftsinformatik oder Games Engineering im Bachelor oder Master an der Technischen Universität München. Aus Vorlesungen, Programmierpraktika, Forschungsprojekten und Abschlussarbeiten können Studenten Erfahrungen in der Softwareentwicklung sammeln. Zusätzlich werden wissenschaftliche Mitarbeiter der Fakultät für Informatik der Technischen Universität München und der fortiss GmbH zur Teilnahme am Experiment eingeladen. Diese betreiben Forschung und Lehre im Bereich der Informatik. Im Rahmen von Forschungsprojekten wenden wissenschaftliche Mitarbeiter ihre Kenntnisse der Softwareentwicklung praktisch an. Die adressierten Softwareentwickler haben ein Studium an der Fakultät für Informatik der Technischen Universität München absolviert und arbeiten in der Industrie.

Als Stichprobenverfahren für die Auswahl von Personen, die zur Teilnahme eingeladen werden, wird Convenience Sampling (Wohlin et al. 2012, S. 93) angewendet. Für die Teilnahme werden keine Anreize angeboten. Die angesprochenen Personen können freiwillig über die Teilnahme entscheiden.

5.2.4 Objekte des Experiments

Die Aufgabe der Versuchspersonen besteht darin die Antwortzeit existierender Java-EE-Komponenten zu verbessern. Komponenten werden in Form einer beispielhaften Java-EE-Anwendung zur Verfügung gestellt. Für die Bearbeitung des Quelltextes wird eine Java EE IDE bereitgestellt. Diese Objekte werden im Folgenden beschrieben.

5.2.4.1 Beispielhafte Unternehmensanwendung

Als beispielhafte Anwendung wird Cargo Tracker[27] verwendet. Cargo Tracker ist ein Open-Source-Projekt und demonstriert die Implementierung einer Anwendung auf Basis von Java EE. Die Anwendung implementiert die Geschäftslogik eines Schiffsfrachtlogistik-

[27] https://cargotracker.java.net/

unternehmens. Eine administrative Oberfläche erlaubt die Planung von Transporten unter Angabe des Start- und Zielhafens sowie einer Transportroute. Je Auftrag können bestimmte Ereignisse, wie die Be- und Entladung spezifiziert werden. Kunden haben die Möglichkeit den aktuellen Status einer Fracht nachzuverfolgen.

Der Hauptquelltext besteht aus 97 Java-Dateien, davon 83 Klassen, 11 Schnittstellen und 3 Enums, die auf 27 Pakete verteilt sind. Die Benutzerschnittstelle ist auf Basis von JSF implementiert. Andere Funktionen werden als REST-Schnittstelle angeboten. Die Geschäftslogik wird mit Hilfe von EJBs umgesetzt. Geschäftsobjekte werden als JPA Entities abgebildet und mit Hilfe von JPA auf eine Datenbank persistiert. Während dem Deployment wird die Datenbank mit Beispieldaten befüllt. Komponenten kommunizieren untereinander entweder über direkte Aufrufe oder JMS.

Im Rahmen des Experiments werden alle Java-Klassen der Anwendung als interne Komponenten klassifiziert (vgl. Abschnitt 4.2.2). Antwortzeitmessungen werden für die Methoden der *EntityManager*-Komponente zur Verfügung gestellt. Die Komponenten *JpaVoyageRepository*, *JpaLocationRepository*, *JpaHandlingEventRepository* und *JpaCargoRepository* verwenden Dienste des *EntityManager* und bilden somit die Haupttreiber für die Antwortzeit der restlichen Komponenten.

Im Vergleich zur SPECjEnterprise2010-Anwendung verfügt Cargo Tracker nicht über einen Lasttreiber mit standardisiertem Workload. Zur Durchführung von Messungen wird ein einfacher Workload, der die sequenzielle Nutzung aller Oberflächenfunktionen vorsieht, spezifiziert. Die Vorteile beim Einsatz von Cargo Tracker gegenüber von SPECjEnterprise2010 für dieses Experiment sind:

- Die intuitive Benutzeroberfläche erleichtert es den Versuchspersonen die Funktionen der Anwendung zu verstehen (vgl. Anhang A).
- Die Anwendung unterstützt ein einfaches und vollautomatisiertes Deployment auf der lokalen Maschine. Die Auswirkung von Änderungen am Quelltext kann somit direkt am laufenden System nachverfolgt werden.
- Cargo Tracker basiert auf der neueren Version Java EE 7.
- Die Paketstruktur des Quelltextes ist im Verglich zu SPECjEnterprise2010, wo eine Unterteilung in Domains stattfindet, intuitiver.

Der Ausgangszustand der Anwendung weist relativ wenig Optimierungspotenziale auf. Leichtgewichtige und spezialisierte Komponenten setzten einzelne Funktionen der Benutzeroberfläche um. Dies erschwert einerseits die Spezifikation einer konkreten Aufgabenstellung und kann andererseits dazu führen, dass Versuchspersonen sehr ähnliche Ergebnisse erzielen. Aufgrund der Einfachheit der existierenden Komponenten wird die Anwendung um die neue Komponente *BatchProcessingBean* erweitert. Diese verwendet einen großen Anteil der Komponenten direkt oder indirekt wieder. Die Funktion der BatchProcessingBean ist es alle verspäteten Frachttransporte zu identifizieren und ggf. erneut einzuplanen. Eine Fracht gilt dann als verspätet, wenn der geplante Zeitpunkt für die Entladung am Zielhafen schon in der Vergangenheit liegt. Die betroffene Fracht wird während dem nächsten Zwischenstopp auf der geplanten Route entladen und im Gegenzug wird ein neuer Transport eingeplant. Wenn ein Transport in Auftrag gegeben

wurde, jedoch keiner Route zugeordnet ist, wird eine passende Auswahl getroffen. Inhalt der Aufgabe der Versuchspersonen ist es die Antwortzeit der BatchProcessingBean und der wiederverwendeten Komponenten zu optimieren. Die Erweiterung der Anwendung um Performancebugs wird in den folgenden Abschnitten beschrieben.

5.2.4.2 Antipatterns als Grundlage von Performancebugs

Die Implementierung von Cargo Tracker besteht aus simplen Komponenten und weist eine relativ hohe Qualität auf. Bei einer Analyse des Quelltextes konnten keine Performancebugs identifiziert werden. Um eine Optimierung der Implementierung zu ermöglichen werden künstliche Performancebugs im Quelltext eingebaut. Die Grundlage für die Konstruktion von Bugs bilden dabei Performance Antipattern, welche Lösungsmuster für bekannte Performanceprobleme beschreiben (Smith/Williams 2000, 2002, 2003). Technologieunabhängige Performance Antipattern sind in Tabelle 5-1 aufgelistet.

Antipattern	**Beschreibung**	**Lösung**
Falling Dominoes	Der Ausfall einer Komponente führt zu Performanceproblemen in anderen Komponenten.	Sicherstellung, dass fehlerhafte Komponenten bis zu ihrer Behebung isoliert werden.
Empty Semi Trucks	Für die Bearbeitung einer Aufgabe ist eine exzessive Anzahl an Anfragen notwendig.	Bündelung mehrerer Einheiten in einer Nachricht, Anwendung von Entwurfsmuster für die Neugestaltung der Schnittstellen.
Roundtripping	Spezialfall von Empti Semi Trucks – Mehrere Felder einer Benutzeroberfläche werden von einem entfernten System einzeln abgefragt.	Erfassung aller Felder in einer Abfrage.
Tower of Babel	Interne Datenstrukturen werden exzessiv zu abstrakten Austauschformaten konvertiert.	Minimierung der Konvertierungsaktivität auf kritischen Prozesspfaden.
Unbalanced Processing	Verfügbare Prozessoren werden nicht ausgenutzt.	Restrukturierung der Software, Aufteilung großer Verarbeitungsschritte, Verlagerung von Verarbeitungsaktivitäten.
Unnecessary Processing	Verarbeitung ist nicht notwendig.	Löschung oder Verlagerung von Verarbeitungsaktivitäten.
The Ramp	Eine bei längerer Nutzung des Systems erhöhte Verarbeitungszeit.	Auswahl passender Algorithmen.

Antipattern	**Beschreibung**	**Lösung**
Sisyphus Database Retrieval	Spezialfall von The Ramp – Durchführung umfangreichger Abfragen, bei welchen nur eine Teilmenge der Ergebnisse benötigt werden.	Lediglich benötigte Daten abfragen.
More is Less	Aufgrund von Thrashing relevante Verarbeitungsschritte vernachlässigt.	Identifizierung von Schwellenwerten, bei denen Thrashing auftritt.
„God“ Class	Eine einzelne Klasse führt alle Verarbeitungsschritte aus oder verwaltet alle Daten.	Neustrukturiertung und Verteilung der Aufgaben.
Excessive Dynamic Allocation	Unnötige Erstellung und Entsorgung großer Mengen von Datenobjekten.	Wiederverwendung existierender Objekte.
Circuitous Treasure Hunt	Benötigte Information muss an unterschiedlichen Stellen nachgefragt werden.	Umstrukturierung des Datenzugriffs oder Optimierung der Zugriffszeit.
One-Lane Bridge	Parallele Verarbeitung ist nicht oder nur eingeschänkt möglich.	Gemeinsame Nutzung von Ressourcen.
Traffic Jam	Ein Problem in der Verarbeitung führt zur Bildung einer Warteschlange, welche auch nach Lösung des Problems bestehen bleibt.	Behebung der Fehlerursache oder Bereitstellung von Ressourcen für größte anzunehmende Last.

Tabelle 5-1: ***Performance Antipatterns***
Quelle: In Anlehnung an Smith/Williams (2003)

Tate (2002) beschreiben Antipattern in der Implementierung von Java-Anwendungen und adressieren neben Aspekte wie Wartbarkeit und Zuverlässigkeit auch die Performance. Unter anderem werden hier auch Antipattern wie Roundtripping abgedeckt (Tate 2002, S. 225). In (Tate et al. 2003) werden Antipatterns vorgestellt, die speziell bei der Implementierung von EJB auftreten können. Diese reichen von der Gestaltung von Schnittstellen bis zu organisatorischen Aspekten, wie z.B. die Integration des Quelltextes mehrerer Entwickler. Chen et al. (2014) beschreiben die Erscheinungsform der Antipattern Empty Semi Trucks und Sisyphus Database Retrieval bei EJB sowie deren Identifikation auf Basis der Annotationen.

Wert (2015, S. 54) ordnet Performance Antipattern hinsichtlich ihrer Abstraktionsebene in die Kategorien Architektur, Entwurf, Implementierung und Deployment ein. Auf der Ebene des Entwurfs sind beispielsweise die Antipattern Circuitous Treasure Hunt und „God“ Class anzutreffen. Im Rahmen des Experiments steht den Versuchspersonen der Quelltext der Anwendung zur Verfügung. Diese verfügen weder über Architekturdokumente, noch über Werkzeugen für die Analyse der Kommunikation zwischen Komponenten. Aufgrund dieser Rahmenbedingungen kommen für die Konstruktion von Performancebugs nur Antipattern auf Implementierungsebene in Betracht. Einige Antipatterns dieser Ebene treffen jedoch nicht auf Cargo Tracker zu.

Beispielsweise deckt die Implementierung der Anwendung kein Einlesen oder Schreiben externer Daten und das Antipattern Tower of Babel kann somit nicht eingebaut werden.

5.2.4.3 Konstruktion von Performancebugs

Aufgrund der Fokussierung der Anwendung auf das Lesen und Schreiben von Entities auf der Datenbank werden folgende Antipatterns systematisch im Quelltext eingebaut:

- Exzessive Datenabfrage (engl. sisyphus database retrieval, Dugan et al. 2002): Abfragen eines spezifischen Objekts werden mit der Abfrage einer Sammlung von Entities ersetzt. Das benötigte Objekt wird durch die Iteration über die abgefragte Sammlung gesucht.
- Einzelabfragen (engl. empty semi trucks, Smith/Williams 2003): Existierende Abfragen einer Sammlung von Entities werden mit Einzelabfragen von individuellen Objekten innerhalb einer Schleife ersetzt.

Die Konstruktion eines Performancebugs als exzessive Datenabfrage ist in Listing 5-1 beispielhaft dargestellt. Der vorhandene Aufruf sucht ein Objekt vom Typ *Cargo* anhand einer bestimmten *TrackingId*. Im angepassten Aufruf werden alle *Cargos* aus der Datenbank ausgelesen. Die Methode *getCargo* iteriert dann über alle Objekte und vergleicht die jeweilige *TrackingId*. Die Methode *findAll* des CargoRepository weist eine viel höhere Antwortzeit auf, als die Methode *find*.

```
1 //Vorhandener Aufruf
2 private Cargo getCargo(TrackingId trackingId){
3     return cargoRepository.find(trackingId);
4 }
5
6 //Konstruierter Performancebug
7 private Cargo getCargo(TrackingId trackingId){
8     List<Cargo> cargos = cargoRepository.findAll();
9     for(Cargo cargo : cargos){
10          if(cargo.getTrackingId().equals(trackingId)){
11                return cargo;
12          }
13    }
14    return null;
15 }
```

Listing 5-1: **_Beispiel für die Konstruktion eines Performancebugs als exzessive Datenabfrage_**
Quelle: Eigene Darstellung

Bei der Konstruktion eines Performancebug als Einzelabfrage würde ein vorhandener Aufruf der Methode *findAll* des *CargoRepository* mit dem wiederholten Aufruf der Methode *find* ausgetauscht werden. Zur Laufzeit der Anwendung würde der Performancebug eine viel höhere Antwortzeit verursachen. Der Ansatz für Performancebewusstsein geht jedoch von der

Standardeinstellung mit einer Wiederholung für Schleifen aus. Nichtsdestotrotz wird auch eine einzelne Iteration über die Methode *find* des *CargoRepository* aufgrund der Überschreitung des ersten Schwellenwerts zu einer Warnmeldung führen und die Versuchsperson damit auf diese Stelle im Quelltext aufmerksam machen.

Neben der systematischen Instanziierung von Antipattern ergeben sich im Quelltext der Anwendung zusätzlich folgende Opportunitäten für Performancebugs:

- Redundante Datenabfrage: Eine vorliegende Sammlung von Entities wird innerhalb einer Methode für einen anderen Zweck erneut abgerufen anstatt wiederverwendet zu werden.
- Bedingter Aufruf: Ein Dienst wird nur unter einer bestimmten Bedingung benötigt, wird jedoch vor der Prüfung der Bedingung aufgerufen.
- Auflösung von Caching: Das Zwischenspeichern einer Sammulung von Entities wird aufgelöst und mit der wiederholten Abfrage von der Datenbank ersetzt.

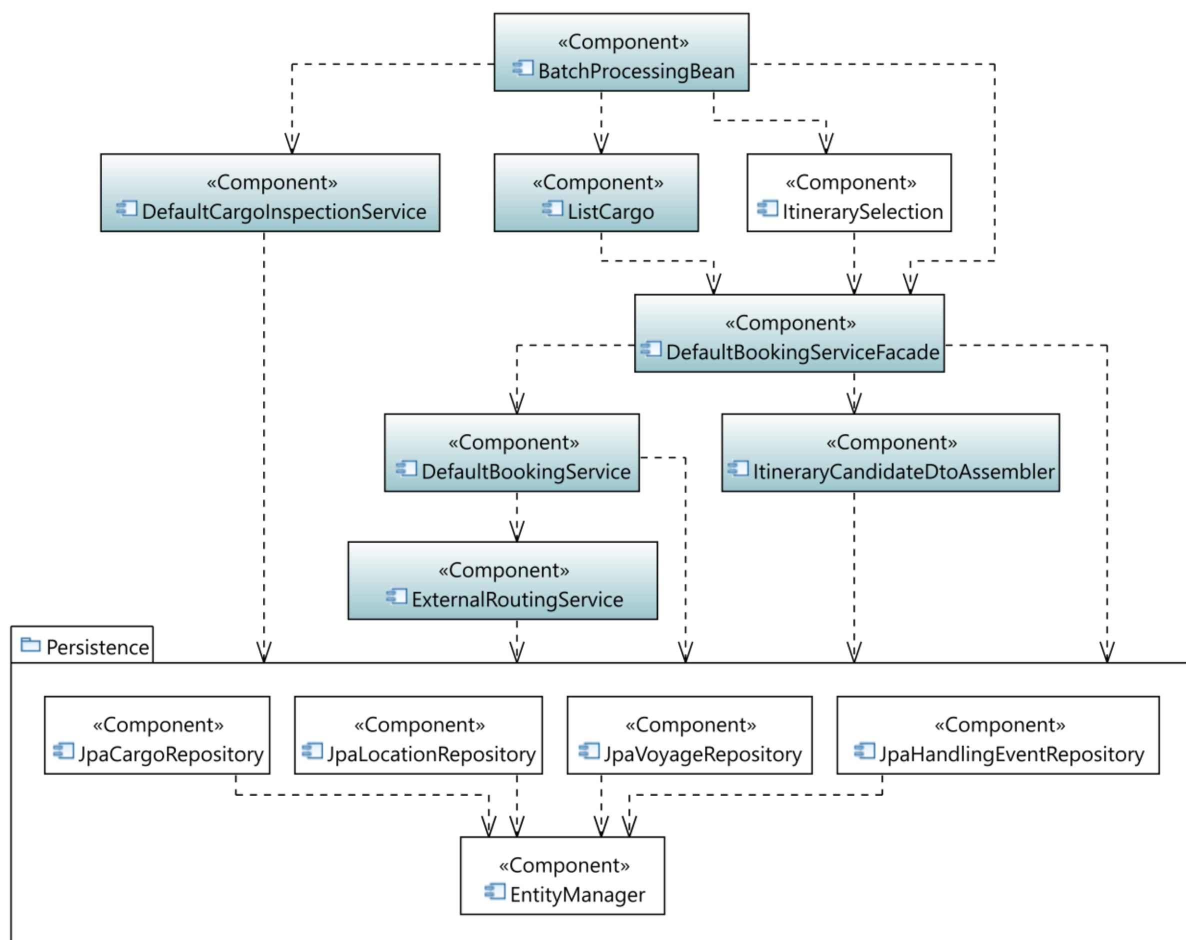


Abbildung 5-2: Überblick über angepasste Komponenten der Anwendung Cargo Tracker
Quelle: In Anlehnung an Danciu/Krcmar (2018)

Komponenten, die um Performancebugs erweitert wurden, sind in Abbildung 5-2 farbig dargestellt. Auf der obersten Ebene befindet sich die BatchProcessingBean, welche für die Versuchspersonen den Einstiegspunkt in die Anwendungslogik darstellt. Diese verwendet

verschiedene andere Komponenten direkt oder indirekt wieder. Auf der untersten Ebene befinden sich die Persistenzschicht, die für das Lesen und Schreiben auf der Datenbank zuständig ist.

Die individuellen Performancebugs und die davon betroffenen Klassen und Methoden sind in Tabelle 5-2 aufgelistet. Die häufigsten Performancebugs führen exzessive Datenabfragen durch, die restlichen Kategorien kommen jeweils einmalig vor.

Bug	**Klasse**	**Methode**	**Einzel-abfrage**	**Exzessive Datenabfrage**	**Redundante Datenabfrage**	**Bedingter Aufruf**	**Auflösung Caching**
PB1	BatchProcessingBean	getBatchProcessingSatus getVoyageByNumber*		X			
PB2	BatchProcessingBean	getBatchProcessingSatus			X		
PB3	BatchProcessingBean	getBatchProcessingSatus				X	
PB4	DefaultCargoInspectionService	inspectCargo isUnloadedAtDestination** getCargo*		X			
PB5	ListCargo	getCargos getRoutedCargos getRoutedUnclaimedCargos getClaimedCargos getNotRoutedCargos					X
PB6	DefaultBookingServiceFacade	loadCargoForRouting		X			
PB7	DefaultBookingServiceFacade	listAllCargos	X				
PB8	DefaultBookingService	bookNewCargo changeDestination getLocation*		X			
PB9	DefaultBookingService	requestPossibleRoutesForCargo assignCargoToRoute changeDestination getCargo*		X			
PB10	ItineraryCandidateDtoAssembler	fromDTO getVoyage*		X			
PB11	ItineraryCandidateDtoAssembler	fromDTO getLocation*		X			

Bug	Klasse	Methode	Einzel-abfrage	Exzessive Datenabfrage	Redundante Datenabfrage	Bedingter Aufruf	Auflösung Caching
PB12	ExternalRoutingService	toLeg getVoyage*		X			
PB13	ExternalRoutingService	toLeg getLocation*		X			

Tabelle 5-2: Eingeführte Performancebugs und die davon betroffenen Klassen und Methoden
Quelle: Eigene Tabelle

Mit dem Symbol * gekennzeichnete Methoden wurden zur Auslagerung eines Performancebugs, der innerhalb einer Klasse mehrfach aufgerufen wird, neu hinzugefügt. Die mit dem Symbol ** gekennzeichnete Methode wurde für die Unterstützung einer neuen Funktion, die von der *BatchProcessingBean* benötigt wird, hinzugefügt.

Den Versuchspersonen wird im Experiment der um Performancebugs erweiterte Quelltext der Anwendung zur Verfügung gestellt. Diese können beliebige Änderungen durchführen. Bei der Bewertung der eingeführten Optimierungen werden folgende Fälle unterschieden:

- **Behebung eines Performancebug (1 Punkt):** Die Versuchsperson hat einen Performancebug identifiziert und korrekt behoben. Die eingefügten Anweisungen weisen keine Syntaxfehler auf und liefern funktional richtige Ergebnisse. Die Behebung eines Performancebug wird mit einem Punkt bewertet.
- **Teilweise Behebung eines Performancebug (0,5 Punkte):** Die Versuchsperson hat einen Performancebug identifiziert und teilweise korrekte Anpassungen durchgeführt. Diese weisen jedoch Syntaxfehler auf oder liefern keine funktional richtigen Ergebnisse. Die Identifikation eines Performancebug wird mit einem halben Punkt bewertet.
- **Alternative Lösung (1 Punkt):** Die Versuchsperson hat ein Performancebug identifiziert und eine alternative Lösung eingeführt. Die eingefügten Anweisungen weisen keine Syntaxfehler auf und liefern funktional richtige Ergebnisse. Alternative Lösungen werden mit einem Punkt bewertet.
- **Geringfügige Verbesserungen (0 Punkte):** Die Versuchsperson ändert Anweisungen, die nicht unbedingt direkt von einem Performancebug betroffen sein müssen und führt geringfügige Optimierungen ein, wie z.B. die Vermeidung unnötiger Variablendeklarationen. Diese Anpassungen werden nicht als gleichwertig mit der Behebung eines Performancebug erachtet und damit nicht bewertet.
- **Unbrauchbare Lösung (0 Punkte):** Die Versuchsperson ändert Anweisungen, die nicht unbedingt direkt von einem Performancebug betroffen sein müssen und führt Optimierungen ein. Isoliert betrachtet könnten diese Änderungen zur einer Reduzierung der Antwortzeit führen. Im Kontext der umgebenden Implementierung tirfft dies jedoch nicht zu. Ein Beispiel hierfür ist die Zwischenspeicherung eines Rückgabewerts, der einmalig benötigt wird. Die durchgeführten Optimierungen können jedoch auch falsch sein. Entsprechende Änderungen werden nicht bewertet.

Performancebugs können durch folgende Änderungen behoben werden:

- Einzelabfrage / exzessive Datenabfragen: Austausch mit dem tatsächlich benötigten Aufruf.
- Redundante Abfrage: der wiederholte Aufruf wird entfernt.
- Bedingter Aufruf: der Aufruf wird erst nach der Prüfung der Bedingung durchgeführt.
- Auflösung von Caching: die Referenz auf dem betroffenen Rückgabewert wird als Klassenattribut zwischengespeichert.

Nach Ablauf der vorgegebenen Zeit muss die Bearbeitung der Aufgabe unterbrochen werden und nur der letzte gespeicherte Stand des Quelltextes wird bewertet. Es ist zu erwarten, dass dabei bereits angefangene Optimierungsmaßnahmen zu einem inkonsistenten Zwischenstand abgebrochen werden. Unvollständige oder fehlerhafte Änderungen führen daher nicht zu einem Punkteabzug.

5.2.4.4 Entwicklungsumgebung

Die Entwicklungsumgebung der Versuchspersonen besteht aus einer IDE, einem lokalen Anwendungsserver, einem Deployment-Skript und einem Browser. Bei der IDE handelt es sich um eine Eclipse IDE for Java EE Developers basierend auf der Plattformversion 4.5.2. Das Plugin für Performancebewusstsein ist nur für Versuchspersonen der Testgruppe aktiviert. Die Funktionen des Plugins sind innerhalb der IDEs der Kontrollgruppe nicht sichtbar. Im Workspace der IDE befindet sich die Anwendung als Java-EE-Projekt. Beim Beginn der Bearbeitung der Aufgabe ist die IDE schon gestartet.

Mit Hilfe eines Automatisierungsskripts können Versuchspersonen in einem Schritt den Quelltext der Anwendung kompilieren, bauen und das resultierende Web Archive auf einem lokalen Anwendungsserver deployen. Dazu wird im Hintergrund ein lokaler Glassfish-Server[28] installiert und hochgefahren. Für diese Aktivität benötigen die Versuchspersonen keinerlei Spezialkenntnisse aus dem Bereich Java EE. Als Ergebnis können die Versuchspersonen mit Hilfe des Browsers auf die laufende Anwendung zugreifen. Nach der Durchführung von Anpassungen muss das Skript erneut ausgeführt werden, damit die Änderungen sichtbar werden.

5.2.5 Instrumentierung des Experiments

Der Ablauf des Experiments besteht aus der Zuordnung der Arbeitsplätze, dem selbständigen Durchlesen einer Anleitung, der Bearbeitung der Aufgabe und das abschließende Ausfüllen eines Fragebogens. Die einzelnen Dokumente und Messverfahren werden im Folgenden beschrieben.

5.2.5.1 Anleitung für Versuchspersonen

Die Beschreibung des Szenarios, der Aufgabe und der Umgebung erfolgt über eine schriftliche Anleitung. Die Versuchspersonen der zwei Gruppen erhalten unterschiedliche Versionen (siehe Anhang A). Die Anleitung enthält die folgenden Abschnitte:

- Ablauf des Experiments: Beschreibt die Abfolge und die Dauer einzelner Aktivitäten während des Experiments.
- Szenario: Beschreibt einen imaginären Kontext in dem Versuchspersonen die Rolle eines Entwicklers der Anwendung Cargo Tracker annehmen. In diesem Szenario hat der Entwickler die erste Version der BatchProcessingBean implementiert. Zur Erleichterung des Einstiegs in die Projektstruktur wird auch der Pfad bzw. das Paket der Klasse genannt. Versuchspersonen erfahren von der Anforderung, dass die Antwortzeit von Funktionen auf der Benutzeroberfläche so niedrig wie möglich sein sollen, idealerweise unter einer Sekunde.
- Szenario (Testgruppe): Versuchspersonen der Testgruppe erfahren zusätzlich von der Existenz eines Plugins für die Vorhersage der Antwortzeit von Methoden.

[28] https://glassfish.java.net/

- Anwendung: Beschreibt die Oberfläche und Funktionen der Anwendung. Mit Hilfe von Screenshots erfährt die Versuchsperson, wie die BatchProcessingBean aufgerufen werden kann.
- Entwicklungsumgebung: Beschreibt die Ordnerstruktur der Entwicklungsumgebung und wie das Automatisierungsskript bedient werden kann.
- Antwortzeitvorhersage (Testgruppe): Die Anleitung für Versuchspersonen aus der Testgruppe beschreibt die Bedienung des Plugins für Performancebewusstsein im Detail mit Hilfe von Screenshots.
- Aufgabe: Beschreibt als Aufgabenstellung die Überarbeitung der BatchProcessingBean und der aufgerufenen Komponenten. Versuchspersonen müssen Optimierungspotenziale identifizieren und umsetzen. Diese werden auch darauf hingewiesen, dass die funktionale Korrektheit durch Optimierungen nicht beeinträchitigt werden darf und diese getestet werden soll.
- Aufgabe (Testgruppe): Versuchspersonen aus der Testgruppe werden zusätzlich aufgefordert, das Plugin für Performancebewusstsein für die Bearbeitung der Aufgabe einzusetzten.
- Kommentare: Versuchspersonen erhalten die zusätzlichen Hinweise, dass nur Java-Dateien geändert werden sollen, dass bestimmte Dateien und Ordner durch Hintergrundprozesse bearbeitet werden und diese nicht gelöscht werden dürfen und dass im Internet nicht nach Optimierungsmaßnahmen gesucht werden soll.

Die Anleitung wird in englischer Sprache ausgedruckt zur Verfügung gestellt. Während der Bearbeitung der Aufgabe darf die Anleitung weiterhin benutzt werden. Eine sonstige Art der Einarbeitung von Versuchspersonen wird nicht durchgeführt.

5.2.5.2 Instrumentierung der Entwicklungsumgebung

Die Aktivitäten der Versuchspersonen werden mit Hilfe der IDE automatisch erfasst. Ein eigens dafür entwickeltes Eclipse Plugin, welches auf dem Eclipse Usage Data Collector (Snipes et al. 2015, S. 91f) basiert, erlaubt es bestimmte Arten von Ereignisse zu identifizieren und zu protokollieren. Zu den erfassten Ereignissen gehören die Änderung einer Klasse, das Ausführen von Kommandos, die Durchführung einer Antwortzeitvorhersage oder die Interaktion mit den Fensterbereichen. Die Informationen, welche im Zusammenhang mit einem Ereignis protokolliert werden, sind in Tabelle 5-3 aufgelistet.

Abhängig von der Ereignisquelle können unterschiedliche Aktionen mit einer spezifischen Beschreibung protokolliert werden. Die unterschiedlichen Ereignisquellen werden im Folgenden beschrieben.

Datei

Bei der Speicherung von Änderungen an einer Java-Datei werden der Dateiname und der neue Inhalt protokolliert. Mit Hilfe des Zeitstempels können unzulässige Änderungen nach dem Ablaufen der verfügbaren Zeit identifiziert und ausgeschlossen werden.

Eigenschaften des Ereignisses	Beschreibung
Zeitstempel	Eine bis auf Millisekunden genaue Angabe über den Zeitpunkt eines Ereignisses.
Ereignisempfänger	Ereignisse werden abhängig von ihrem Ursprung von unterschiedlichen Empfängern identifiziert und protokolliert.
Ereignisquelle	Beschreibt eine Komponente, von der ein Ereignis verursacht wurde.
Aktion	Beschreibt welche Aktion zur Erzeugung eines Ereignisses geführt hat.
Beschreibung	Zusätzliche Informationen in Abhängigkeit von der Ereignisquelle.

Tabelle 5-3: Protokollierung von Ereignissen innerhalb der IDE
Quelle: Eigene Tabelle

Command

Eclipse Commands implementieren verschiedene Oberflächenfunktionen und Tastenkombinationen, wie z.B. der Aufruf der Textsuche oder das Auskommentieren von Quelltextzeilen. Jede Ausführung eines Command wird unter Angabe der implementierten Funktion protokolliert. Die hierdurch erfassten Informationen unterstützen eine detaillierte Untersuchung der Ergebnisse einzelner Versuchspersonen. Die Nutzung von Oberflächenfunktionen, die als Eclipse Action (Gallardo et al. 2003, S. 237) implementiert sind, kann nicht automatisch protokolliert werden.

Workbench

Innerhalb von Eclipse werden Inhalte in unterschiedlichen Fensterbereichen dargestellt. Die Workbench stellt das Hauptfenster der IDE dar und ist in Views und Editoren aufgeteilt (Gallardo et al. 2003, S. 16). Der Project Explorer ist die Implementierung einer View und unterstützt die Navigation durch die Paket- und Ordnerstruktur eines Projektes. Perspektiven strukturieren Funktionen nach Technologiebereichen. Inhalte von Dateien können mit Editoren dargestellt und bearbeitet werden. Aktionen auf Fensterbereiche, wie z.B. das Öffnen, Schließen, Aktivieren und Deaktivieren werden von der IDE erfasst und als Ereignis protokolliert. Für Editoren wird neben der konkreten Implementierung auch die angezeigte Datei erfasst. Diese Ereignisse erlauben die Untersuchung verschiedener Aspekte, wie z.B.:

- Nachverfolgung des Start- und Entzeitpunkts der Bearbeitung von Aufgaben.
- Nachverfolgung des Navigationsverhaltens.
- Berechnung der Zeit, die eine Versuchsperson innerhalb bzw. außerhalb der IDE oder eines bestimmten Editors verbringt.

Plugin für Performancebewusstsein

Interaktionen der Versuchspersonen mit dem Plugin für Performancebewusstsein werden als spezielle Ereignisse erfasst. Bei der Anforderung einer Antwortzeitvorhersage wird der Name

der fokussierten Komponente spezifiziert. Die Rückmeldung von einzelnen Hinweisen in Form von Marker wird unter Berücksichtigung des Typs, des Methodennamens und der vorhergesagten Antwortzeit protokolliert.

Logeinträge

Alle von der IDE während der Bearbeitung der Aufgabe generierten Logeinträge werden erfasst und als Ereignis protokolliert. Somit können unvorhergesehene Ereignisse, wie z.B. das Auftreten von Laufzeitfehlern, identifiziert werden.

Das Protokoll der Ereignisse wird in einem standardisierten Format auf einer lokalen Datei gespeichert.

5.2.5.3 Nachgelagerte Befragung zur Erhebung der Kenntnisse

Informationen über die Ausbildung und die Kenntnisse der Versuchspersonen werden nach der Durchführung des Experiments mit Hilfe eines Fragebogens erhoben (siehe Anhang B).

Im ersten Teil des Fragebogens wird der höchste Abschluss und das Arbeitsverhältnis der Versuchsperson abgefragt. Beim höchsten Abschluss wird zwischen Schulabschluss, Bachelor, Master und Doktorgrad unterschieden. Aufgrund des Kontexts des Experiments werden bei der Erhebung des Arbeitsverhältnisses Arbeitnehmer in Wissenschaftler und Praktiker unterteilt. Insgesamt wird zwischen Studenten, angestellte Wissenschaftler, angestellte Praktiker und Selbständige unterschieden.

Im Rahmen des Experiments setzen Versuchspersonen unterschiedliche Technologien und Werkzeuge zur Bearbeitung der Aufgabe ein. Die jeweiligen Kenntnisse können dabei einen Einfluss auf die erzielten Ergebnisse haben und werden daher im Fragebogen abgefragt. Versuchspersonen geben eine Einschätzung über ihre Fähigkeiten anhand von jeweils vier Stufen nach dem Beispiel von Wohlin et al. (2012, S. 205) ab. Die abgefragten Bereiche werden im Folgenden beschrieben.

Grundlegende Kenntisse

Allgemeine Programmierkenntnisse werden für die Teilnahme am Experiment vorausgesetzt. Zu Auswahl stehen die Möglichkeiten:

- eins bis zwei Kurse,
- drei oder mehrere Kurse, jedoch keine Industrieerfahrung,
- einige Kurse und etwas Industrieerfahrung und
- mehr als drei Kurse und mehr als ein Jahr Industrieerfahrung.

Spezifische Sprachen und Technologien

Die Java-Programmiersprache, die Java-EE-Plattform, Problemstellungen der Softwareperformance, die Eclipse IDE und JPA werden im Fragebogen individuell adressiert. Durch die Bereitstellung einer existierenden Implementierung und die Automatisierung des Buildprozesses

werden für die Teilnahme am Experiment keine spezifischen Kenntnisse vorausgesetzt. Zu Auswahl stehen die Möglichkeiten:

- keine Erfahrung,
- ein Buch gelesen oder einen Kurs belegt,
- Industrieerfahrung, jedoch weniger als sechs Monate und
- Industrieerfahrung.

Kenntisse über die Anwendung

Kenntnisse über Cargo Tracker könnten das Ergebnis einer Versuchsperson stark beeinflussen, denn aufgrund einer Vertrautheit mit dem Quelltext könnten die Performancebugs offensichtlich erscheinen. Zu Auswahl stehen die Möglichkeiten:

- keine Erfahrung,
- informiert über die Anwendung,
- Anwendung wurde genutzt und
- Anwendung wurde entwickelt.

Kenntnisse über Java-EE-Anwendungsserver werden nicht abgefragt, da der Glassfish-Server im Hintergrund gestartet wird und Versuchspersonen diesen nicht direkt nutzen müssen.

5.2.5.4 Nachgelagerte Befragung zur Erhebung der Benutzerfreundlichkeit

Neben den Programmierkenntnissen der Versuchspersonen wird nach der Bearbeitung der Aufgabe auch die von ihnen bei der Nutzung des Plugins bzw. der IDE wahrgenommene Benutzerfreundlichkeit erfasst.

Die Benutzerfreundlichkeit (engl. usability) beschreibt den Grad, in dem ein Produkt durch ein Nutzer für die effiziente, effektive und zufriedene Lösung einer Aufgabe eingesetzt werden kann (ISO 1998). Für eine Evaluation des Ansatzes für Performancebewusstsein soll neben der Anzahl der behobenen Performancebugs auch die von den Versuchspersonen empfundene Benutzerfreundlichkeit mit Hilfe eines Fragebogens erhoben werden. Nach Sauro/Lewis (2012, S. 186ff) existieren zwei Arten von Standardfragebögen für die Messung der Benutzerfreundlichkeit:

- Post-Study-Fragebögen unterstützen die Messung der Benutzerfreundlichkeit eines ganzen Systems nach der Durchführung einer Studie, bestehend aus mehreren Aktivitäten. Hiermit können unterschiedliche Systeme oder Versionen eines Systems miteinander verglichen werden.
- Post-Task-Fragebögen unterstützen die Messung der Benutzerfreundlichkeit einzelner Funktionen nachdem diese für die Bearbeitung einer Aktivität im Rahmen einer Studie genutzt wurden. Hiermit können Probleme in der Nutzung einzelner Bereiche der Benutzeroberfläche untersucht werden.

Das vorliegende Experiment besteht aus einer Aufgabe und Versuchspersonen sollen eine einzige Funktion des Plugins für Performancebewusstsein verwenden. Andererseits nimmt die Bearbeitung der Aufgabe relativ viel Zeit in Anspruch und die Versuchspersonen müssen parallel mit unterschiedlichen Funktionen der IDE interagieren. Das Plugin für Performancebewusstsein ist in vielen Hinsichten eng mit diesen Funktionen verknüpft:

- Antwortzeitvorhersagen werden über ein Menüpunkt des Project Explorer angefordert.
- Der Fortischritt der Vorhersage wird in der Progress Bar angezeigt.
- Hinweise werden in Form von Marker und Tooltips im Editor angezeigt.
- Die Navigation zwischen aufgerufene und aufrufende Klassen erfolgt über Eclipse-Funktionen.

Einzelne Schritte der Interaktion können nicht durch die Unterbrechung des Arbeitsflusses der Versuchspersonen mit Hilfe eines Post-Task-Fragebogens untersucht werden. Daher wird für die Messung der Benutzerfreundlichkeit ein Post-Study-Fragebogen ausgewählt. Sauro/Lewis (2012, S. 186) empfehlen hierfür die *System Usability Scale* (SUS). SUS besteht aus zehn Items unter Anwendung einer Likert-Skala und erfasst eine allgemeine und subjektive Einschätzung der Benutzerfreundlichkeit (Brooke 1996). Alle Fragen sind durchlaufend nummeriert. Fragen mit einer ungeraden Nummer adressieren positive Eigenschaften, wie z.B. die Einfachheit des Systems. Negative Eigenschaften, wie z.B. die Komplexität des Systems, werden durch Fragen mit geraden Nummern adressiert. Brooke (1996) schlagen die Verwendung einer Likert-Skala mit fünf Schritten vor. Laut Sauro/Lewis (2012, S. 187) liefern Skalen mit mehr Schritten generell bessere Ergebnisse. Fragen werden daher über eine Likert-Skala mit sieben Schritten, von einer vollen Zustimmung, bis zur vollen Ablehnung, beantwortet. Das Zwischenergebnis einer Frage mit ungerader Nummer berechnet sich aus der Subtraktion von eins aus der Punktezahl. Bei geraden Nummern wird die Punktezahl von dem Wert sieben subtrahiert. Das Gesamtergebnis berechnet sich aus der Summe der Zwischenergebnisse aller Fragen.

Anhand der Stichproben über das SUS-Ergebnis zweier Produkte kann deren Benutzerfreundlichkeit verglichen werden (Sauro/Lewis 2012, S. 68). Im Rahmen des Experiments werden zur Erhebung der Benutzerfreundlichkeit zwei unterschiedliche Versionen des Fragebogens eingesetzt. Versuchspersonen der Testgruppe werden aufgefordert das Plugin für Performancebewusstsein zu bewerten. Weil die Versuchspersonen der Kontrollgruppe das Plugin nicht einsetzen, werden sie aufgefordert allgemein die IDE zu bewerten. Für den Vergleich der zwei Stichproben kommt der t-Test in Frage (Sauro/Lewis 2012, S. 68).

Am Ende des Fragebogens können Versuchspersonen in einem Freitextfeld zusätzliche Kommentare zur Benutzerfreundlichkeit und den Funktionen schreiben.

5.2.6 Verfahren zum Zusammentragen der Daten

Der Aufbau des Experiments sieht die Durchführung mehrerer Iterationen an unterschiedlichen Tagen und zu unterschiedlichen Uhrzeiten vor (siehe auch Abschnitt 5.2.2). Jede Iteration verfolg einen festen Ablauf:

- Einlass und Zuordnung zu Arbeitsplätzen: Versuchspersonen werden zuerst über ein Losverfahren zu Arbeitsplätzen zugeordnet (siehe auch Abschnitt 5.2.2).
- Durchlesen der Anleitung (10 Minuten): Abhängig von der Zuordnung zu einer Experimentgruppe erhalten die Teilnehmer eine unterschiedliche Anleitung für das Experiment (siehe auch Abschnitt 5.2.5.1).
- Bearbeitung der Aufgabe (40 Minuten): Versuchspersonen dürfen gleichzeiting mit der Bearbeitung der Aufgabe beginnen. Nach 40 Minuten muss die Bearbeitung abgerbrochen werden.
- Fragebogen: Direkt nach der Bearbeitung der Aufgabe werden die Fragebögen ausgeteilt. Die Zeit zum Ausfüllen des Fragebogens wird nicht begrenzt.
- Abschluss: Nach dem Ausfüllen der Fragebögen verlassen die Versuchspersonen das Computerlabor und die Ergebnisse werden anschließend eingesammelt.

Ergebnisse der Versuchspersonen werden durch die IDE automatisch protokolliert. Nachdem alle Versuchspersonen das Computerlabor verlassen wird jede einzelne Protokolldatei auf einer zentralen Netzwerkablage kopiert. Die ausgefüllten Fragebögen werden von dem Experimentator eingesammelt und manuell in eine Tabelle überführt. Alle Arbeitsplätze werden vor einer neuen Iteration initialisiert. Die zuvor beschriebenen Aktivitäten werden einzig durch den Autor dieser Arbeit durchgeführt.

Die im Experiment erhobenen Daten sind anonym. Versuchspersonen erhalten über die zufällige Zuordnung zu einem Arbeitsplatz eine Nummer. Einzelne Datensätze werden bei der Auswertung der Ergebnisse über diese Nummer identifiziert.

5.2.7 Verfahren zur Analyse der Ergebnisse

Der Fokus des Experiments liegt auf der Untersuchung der Hypothese, dass die Nutzung des Ansatzes für Performancebewusstsein die Anzahl der Performancebugs, die während der Optimierung von Komponenten identifiziert werden, nicht beeinflusst. Hierbei handelt es sich um eine Unterschiedshypothese (Bortz/Döring 2007, S. 492). Aufgrund der Alternativhypothese, die eine positive Beeinflussung impliziert, ist die Hypothese gerichtet. Aus dem Aufbau des Experiments ergeben sich zwei Stichproben über die Anzahl der behobenen Performancebugs. Bei dieser Anzahl handelt es sich um einen Wert der Verhältnisskala (Stevens 1946). Nach Wohlin et al. (2012, S. 137) kommen für den gewählten Aufbau des Experiments folgende Signifikanztests in Frage:

- Parametrische Signifikanztests, wie der t- und der F-Test, die eine Normalverteilung der Stichproben voraussetzen.
- Nonparametrische Signifikanztests, wie der Mann-Whitney- und der Chi-2-Test, die für nichtnormalverteilte Stichproben geeignet sind.

Mit Hilfe des Kolmogorov-Smirnov- und Shapiro-Wilk-Tests kann die Eigenschaft der Normalverteilung von Stichproben getestet werden (Elliott/Woodward 2007, S. 25). Dabei wird die Hypothese, dass die untersuchte Stichprobe nicht von einer normalverteilten Stichprobe

unterscheidet, getestet. Abhängig von der Verteilung der Stichproben wird für die Auswertung der Ergebnisse entweder der t-Test oder der Mann-Whitney-Test ausgewählt.

Es existieren unterschiedliche Arten des t-Test, für die Untersuchung einer Stichprobe (engl. one-sample t-test), zweier Stichproben (engl. two-sample t-test) und für die paarweise Untersuchung zweier Stichproben (engl. paired t-test) (Elliott/Woodward 2007, S. 47). Für den gewählten Aufbau eignet sich der t-Test für die Untersuchung zweier Stichproben.

Neben der Anzahl der behobenen Performancebugs werden im Experiment noch zahlreiche andere Merkmale gemessen, wie z.B. die Benutzerzufriedenheit, das Qualifikationsniveau, die in der IDE verbrachte Zeit, Anzahl der navigierten und der geänderten Java-Dateien und die Anzahl der ausgeführten Commands. Bis auf die Merkmale der Benutzerfreundlichkeit und des Qualifikationsniveaus gehören alle Messwerte zur Verhältnisskala. Für die Merkmale auf Basis der Verhältnisskala existieren jeweils zwei Stichproben, deren Ähnlichkeit mit den vorhin beschriebenen statistischen Methoden untersucht werden.

Die Bewertung der Benutzerfreundlichkeit im SUS-Fragebogen findet auf Basis einer Ordinalskala statt (Sauro/Lewis 2012, S. 243). Nach Stevens (1946) sollten keine mathematischen Operationen, bis auf das Abzählen, auf Werte der Ordinalskala angewendet werden. Sauro/Lewis (2012, S. 245) empfehlen trotzdem den Einsatz des t-Test für den Vergleich zweier Stichproben von SUS-Ergebnissen.

Die Merkmale des Qualifikationsniveaus im Sinne des höchsten Abschlusses, des Arbeitsverhältnisses und der Programmierkenntnisse entsprechen auch der Ordinalskala. Im Gegensatz zu einer Likert-Skala können die Unterschiede zwischen den einzelnen Schritten jedoch stärker voneinander abweichen. Der Unterschied zwischen wenigen und vielen besuchten Kurse könnte als geringer eingeschätzt werden, als der Unterschied zwischen vielen Kursen und Industrieerfahrung. Auf diese Merkmale können daher die vorhin beschriebenen statistischen Verfahren nicht angewendet werden.

Für die Untersuchung der Korrelation zwischen zwei Metriken wird der Pearson Correlation Coefficient (PCC) herangezogen.

5.2.8 Sicherstellung der Validität der Ergebnisse

Objekte des Experiments werden im Rahmen einer Pilotdurchführung mit einer Versuchsperson hinsichtlich ihrer Konsistenz, funktionalen Korrektheit und Verständlichkeit geprüft.

Die Durchführung des Experiments wird permanent von dem Koordinator überwacht. Versuchspersonen unterschiedlicher Gruppen können innerhalb einer Reihe keine benachbarten Arbeitsplätze belegen. Eine möglichst weite Entfernung zwischen benachbarten Arbeitsplätzen derselben Reihe wird angestrebt. Benachbarte Reihen von Arbeitsplätzen sind in entgegengesetzte Richtungen aufgestellt. Bildschirme sind damit für eine hintere Reihe noch schwerer einsehbar.

Für die Auswertung der Ergebnisse einer Iteration werden die Protokolldateien zuerst bereinigt. Alle Ereignisse, die außerhalb des Zeitraums der Bearbeitung liegen werden aus der Protokolldatei gelöscht (siehe auch Abschnitt 5.2.5.2). Damit Änderungen am Quelltext, die nach der

erlaubten Bearbeitungszeit gespeichert wurden, von der Auswertung ausgenommen. Die bereinigten Protokolldateien werden mit Hilfe eines eigens dafür entwickelten Werkzeugs automatisch verarbeitet. Das Werkzeug führt folgende Schritte aus:

- Berechnung der Gesamtbearbeitungszeit: Durch die Berechnung der Zeitdauer zwischen dem ersten und letzten Eintrag der Protokolldatei wird die Gesamtbearbeitungszeit ermittelt. Hiermit wird überprüft, ob die Protokolldatei vollständig bereinigt wurde.
- Extraktion des Quelltextes: Die letzte Version zur jeder bearbeiteten Java-Datei wird extrahiert und separat gespeichert.
- Berechnung der Zeiten, die eine Versuchspersonen innerhalb und außerhalb der IDE verbracht hat.
- Berechnung der Anzahl der geänderten bzw. geöffneten Java-Dateien, der Anzahl der Ausgeführten Commands und der Anzahl der durchgeführten Antwortzeitvorhersagen.

Für jede Versuchsperson werden die geänderten Java-Dateien in dem Quelltext der Anwendung Cargo Tracker integriert und hinsichtlich der Kriterien aus Abschnitt 5.2.4.3 bewertet. Die Anzahl der behobenen Performancebugs wird in einer zentralen Tabelle zusammengefasst.

5.2.9 Gegenüberstellung des Experimentaufbaus

Eine Gegenüberstellung des Vorgehens aus der vorliegenden Arbeit und der von Horký et al. (2015) ist in Tabelle 5-4 aufgelistet. Neben der Verfügbarkeit ihres Ansatzes sehen Horký et al. (2015) als drittes Treatment auch die Bereitstellung von absichtlich falschen Performancehinweisen vor. Dies ermöglicht einerseits die Untersuchung, ob beim praktischen Einsatz evtl. auftretende Fehler dazu führen könnten, dass Entwickler zu einem ungewollten Verhalten beeinflusst werden. Andererseits kann dadurch auch hinterfragt werden, ob bessere Implementierungen tatsächlich mit der Verfügbarkeit des Ansatzes zusammenhängen. Der Nachteil eines zusätzlichen Treatments ist, dass für die Erzielung von signifikaten Ergebnisse mehr Teilnehmer notwendig sind. In der vorliegenden Arbeit wird auf dieses dritte Treatment verzichtet. Stattdessen wird nach Durchführung des Experiments ausgewertet, ob einzelne vorhergesagte Antwortzeiten, welche nicht sinnvoll adressiert bzw. optimiert werden können, zu einem unerwünschten Verhalten der Probanden führen.

	Vorliegende Arbeit	**Horký et al. (2015)**
Aufgabenstellung	Optimierung einer existierenden Implementierung	Implementierung einer neuen Methode
Softwaretyp	Java-EE-Anwendung	Java-Algotithmus zum Einlesen von XML
Experimentumgebung	Computerlabor	Hausaufgabe
Versuchspersonen	Studenten, Wissenschaftler und Praktiker	Studenten
Zeitrahmen	40 Minuten	Nicht näher spezifiziert
Abhängige Variable	Anzahl der behobenen Performancebugs	Ausführungszeit des Algorithmus für einen bestimmten Input
Treatments	- Keine Unterstützung - Korrekte Hinweise	- Keine Unterstützung - Korrekte Hinweise - Falsche Hinweise
Instrumentierung	Protokollierung der Aktivitäten durch IDE	Abgelieferter Quelltext

Tabelle 5-4: ***Gegenüberstellung des Experimentaufbaus aus der vorliegenden Arbeit und der von Horký et al. (2015)***
Quelle: Eigene Tabelle

5.3 Durchführung des Experiments

Das Experiment wurde in dem Zeitraum zwischen den 17.03.2016 und dem 18.04.2016 über acht Iterationen durchgeführt. Insgesamt nahmen 26 Versuchspersonen am Experiment teilt. Die Zusammensetzung der Teilnehmer sowie der Verlauf der Iterationen werden im Folgenden beschrieben.

5.3.1 Teilnehmer des Experiments

Potenzielle Versuchspersonen wurden direkt angeschrieben bzw. angesprochen und zur freiwilligen Teilnahme am Experiment eingeladen. Dabei wurden keinerlei Belohnungen angeboten. Auf die Einladung haben 26 Personen positiv reagiert. Die grobe Einteilung der Teilnehmer ist in Tabelle 5-5 aufgelistet. Insgesamt haben sich 21 Studenten und 5 Berufserfahrene für die Teilnahme gemeldet. Bei der Aufteilung in Experimentgruppen ergibt sich dadurch automatisch ein Ungleichgewicht im Umfang von einem Studenten für eine Gruppe und einem Berufserfahrenen für die andere.

Einteilung der Teilnehmer	Unterteilung	Anzahl
Studenten	Bachelor	11
	Master	10
Berufserfahrene	Wissenschaftler	4
	Praktiker	1

Tabelle 5-5: ***Überblick über angemeldete Versuchspersonen***
Quelle: Eigene Tabelle

Teilnehmende Studenten decken die Studiengänge Informatik, Wirtschaftsinformatik und Games Engineering der Fakultät für Informatik der Technischen Universität München ab. Alle teilnehmenden Wissenschaftler sind Mitarbeiter der fortiss GmbH. Bei dem Praktiker handelt es sich um einen angestellten Softwareentwickler eines im Folgenden nicht näher spezifizierten Unternehmens.

5.3.2 Iterationen des Experiments

Ausgehend von der Verfügbarkeit der Teilnehmer und des Computerlabors wurden über einen Zeitraum von einem Monat, an unterschiedlichen Tagen und zu unterschiedlichen Uhrzeiten, insgesamt acht Iterationen durchgeführt. Anhand der Einteilung der Versuchspersonen einer Iteration in Studenten und Berufserfahrene wurden diese über ein Losverfahren der Test- oder Kontrollgruppe zugeordnet. Das Datum, die ungefähre Uhrzeit und die Einteilung der Versuchspersonen zu jeder Iteration sind in Tabelle 5-6 zusammengefasst.

An der ersten Iteration haben vier Studenten teilgenommen. Die Gruppenzuordnung wurde per Losverfahren durchgeführt. Aufgrund der Homogenität der Teilnehmer musste Blockbildung nicht angewendet werden. An der zweiten Iteration haben hingegen ein Student und ein Berufserfahrener teilgenommen. Diese wurden ebenfalls per Losverfahren zugeordnet, jedoch konnte keine gleichmäßige Aufteilung durchgeführt werden. Durch eine feste Zuordnung der Teilnehmer der dritten Iteration wurde das Gleichgewicht dann wiederhergestellt. Teilnehmer wurden dabei entgegengesetzt zu der vorherigen Iteration aufgeteilt. In der vierten Iteration wurden die Versuchspersonen gleichmäßig per Losverfahren eingeteilt. Während der fünften Iteration ist erneut ein Ungleichgewicht entstanden, welches aber in der nächsten Iteration wiederhergestellt wurde.

Aufgrund von unvorhersehbaren Ereignissen standen in der siebten Iteration zu der spezifizierten Uhrzeit nur Arbeitsplätze der Kontrollgruppe zur Verfügung. Beide Teilnehmer dieser Iteration wurden daher derselben Gruppe zugeordnet. Zur Ausbalancierung der Stichproben je Experimentgruppe wurde in der achten Iteration der Umfang der Kontrollgruppe um einen Arbeitsplatz reduziert und der Umfang der Testgruppe um einen erhöht.

Nr.	Datum	Uhrzeit	Ergebnis der Zuordnung		Randomisierung
			Testgruppe	**Kontrollgruppe**	
1	17.03.2016	16 Uhr	Studenten: 2	Studenten: 2	Losverfahren
2	23.03.2016	8 Uhr	Berufserfahrene: 1	Studenten: 1	Losverfahren
3	24.03.2016	17 Uhr	Studenten: 1	Berufserfahrene: 1	Zuordnung vorgegeben, aufgrund des Ungleichgewichts in der vorherigen Runde.
4	28.03.2016	9 Uhr	Studenten: 1	Studenten: 1	Losverfahren
5	31.03.2016	17 Uhr	Berufserfahrene: 1	Studenten: 1	Losverfahren
6	04.04.2016	17 Uhr	Studenten: 1	Berufserfahrene: 1	Zuordnung vorgegeben, aufgrund des Ungleichgewichts in der vorherigen Runde.
7	14.04.2016	17 Uhr	Keine Teilnehmer	Berufserfahrene: 1 Studenten: 1	Zuordnung zu Kontrollgruppe aufgrund der Verfügbarkeit der Rechner.
8	18.04.2016	14 Uhr	Studenten: 6	Studenten: 4	Losverfahren bei Reduzierung der Plätze der Kontrollgruppe zur Ausbalancierung der vorherigen Runde.

Tabelle 5-6: Iterationen bei der Durchführung des Experiments
Quelle: Eigene Tabelle

Alle Versuchspersonen haben das Experiment vollständig absolviert. Die Testgruppe besteht insgesamt aus elf Studenten und einem Berufserfahrenen. Die Kontrollgruppe besteht aus zehn Studenten und zwei Berufserfahrenen.

5.3.3 Sicherstellung der Validität der Ergebnisse

Im Zeitraum der Durchführung des Experiments haben die Rechner des Computerlabors insgesamt ein relativ langsames Antwortzeitverhalten aufgewiesen. Um eine Verzerrung der Ergebnisse zu verhindern, wurden alle Iterationen in demselben Computerlabor durchgeführt. Vor Beginn einer Iteration wurden sowohl die Rechner hochgefahren als auch die IDEs gestartet. Damit konnten Versuchspersonen bei der Bearbeitung der Aufgabe die Entwicklungsumgebung direkt nutzen.

Die Durchführung des Experiments wurde permanent durch den Experimentator überwacht. Auf Probleme oder Fragen konnte sofort reagiert werden. Versuchspersonen wurden vor dem Beginn der Aufgabenbearbeitung darüber informiert, dass jederzeit Fragen gestellt und Probleme gemeldet werden können. Diese wurden aber dazu angehalten so leise wie möglich zu sprechen, damit andere Teilnehmer nicht aufgrund der besprochenen Inhalte in ihrem Verhalten beeinflusst werden. Zu jeder Iteration wurden dieselben Hinweise mitgeteilt.

Während der Bearbeitung der Aufgabe sind einige wenige Probleme aufgetreten. Die IDE einer Versuchsperson ist einmalig abgestürzt, konnte aber ohne Datenverlust neu gestartet werden. Zwei Versuchspersonen konnten zu einem bestimmten Zeitpunkt die Anwendung nicht erneut deployen. Die Ursache lag darin, dass bei einem vorherigen Deployment das Automatisierungsskript nicht richtig geschlossen wurde und der Anwendungsserver im Hintergrund noch lief. Beim erneuten Deployment hatte das Skript dann keine Möglichkeit mehr einen neuen Server zu starten. Diese Probleme wurden durch die Versuchspersonen rechtzeitig gemeldet und schnell gelöst.

5.3.4 Beispielhafter Output des Plugins für Performancebewusstsein

Das Plugin für Performancebewusstsein zielt nicht darauf ab, Bugs zu identifizieren, sondern integriert Messungen im Quelltexteditor und macht Entwickler auf die erwarteten Antwortzeiten von Methoden aufmerksam. Ineffiziente Abfolgen von Anweisungen können jedoch Methodenaufrufe enthalten, die vom Plugin hervorgehoben werden. Eine Übersicht der Hinweise, welche vom Plugin nach der Durchführung einer Antwortzeitvorhersage für die Klasse der obersten Ebene angezeigt werden, ist in Abbildung 5-3 dargestellt.

Beim Ausführen einer Vorhersage für die BatchProcessingBean werden die Methodendeklarationen der wiederverwendeten Klassen innerhalb des aktuellen Projekts ebenfalls besucht und um Hinweise erweitert. Fehler- und Warnmeldungen werden hier als rote oder gelbe Balken gegenüber der Zeilennummer dargestellt, wo diese im Editor auftreten würden. Von den eingeführten Performancebugs betroffene Quelltextzeilen sind daneben als graue Balken dargestellt. Diese ineffizienten Codesequenzen werden jedoch keinem der Teilnehmer bekannt gemacht. Beide Gruppen müssen daher ineffiziente Codesequenzen zuerst identifizieren und dann optimieren.

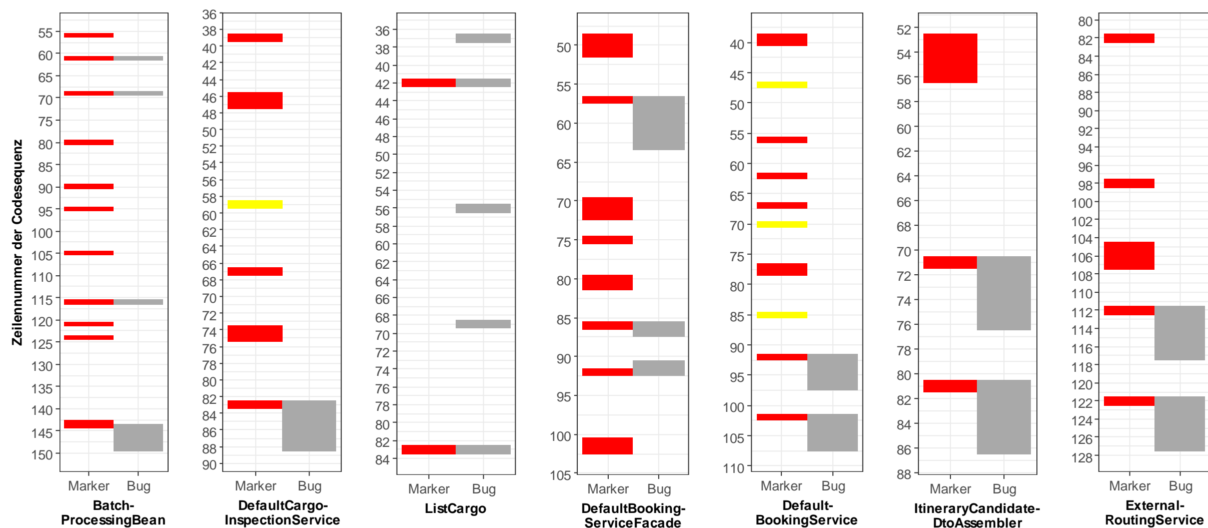


Abbildung 5-3: ***Überblick der Hinweise des Plugins für Performancebewusstsein gegenüber den eingefügten Performancebugs***
Quelle: Eigene Darstellung

Aus Abbildung 5-3 geht hervor, dass nicht alle ineffizienten Codesequenzen durch Hinweise hervorgehoben werden. In diesem konkreten Beispiel werden die Zeilen 37, 56 und 69 der Klasse ListCargo nicht hervorgehoben, da die umschließenden Methoden weder direkt noch indirekt von der Klasse BatchProcessingBean wiederverwendet werden. Andererseits werden viele Codesequenzen durch Hinweise hervorgehoben, obwohl diese nicht um Performancebugs erweitert wurden. Entweder werden dort Methoden aufgerufen, welche eine hohe, jedoch nicht vermeidbare, Antwortzeit aufweisen, oder von den eingeführten Performancebugs direkt oder indirekt betroffen sind. Wie in Abschnitt 5.2.9 beschrieben, können diese Hinweise Aufschluss darüber geben, ob falsche Meldungen das Verhalten der Teilnehmer negativ beeinflussen.

5.4 Auswertung der Eigenschaften der Versuchspersonen

Die Zuordnung der Versuchspersonen auf die Test- oder Kontrollgruppe erfolgte zufällig. Anhand einer Einteilung in Studenten und Berufstätige wurde Blockbildung angewendet (vgl. Abschnitte 5.2.2 und 5.3.2). Nach der Durchführung des Experiments wurde bei den Versuchspersonen anhand eines Fragebogens der höchste Abschluss, das Arbeitsverhältnis und die Kompetenzstufe für relevante Bereiche erhoben. Im Folgenden wird das Ergebnis der Auswertung dieser Eigenschaften zusammengefasst.

5.4.1 Verteilung der Versuchspersonen anhand der Ausbildung und des Arbeitsberhältnisses

Die Verteilung der Versuchspersonen anhand des höchsten Abschlusses auf die Experimentgruppen ist in Abbildung 5-4 dargestellt. In Abbildung 5-5 ist die Verteilung der Versuchspersonen anhand des Arbeitsverhältnisses dargestellt.

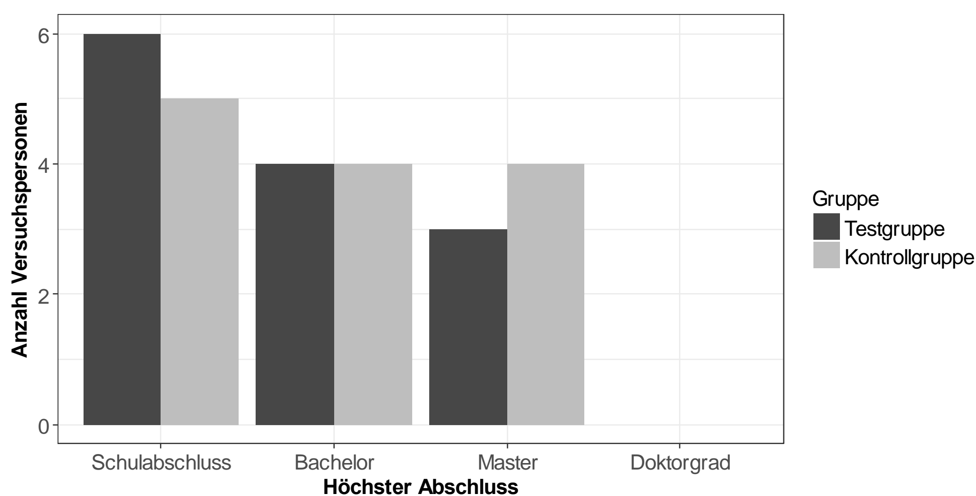


***Abbildung 5-4:* Verteilung der Versuchspersonen anhand des höchsten Abschlusses**
Quelle: In Anlehnung an Danciu/Krcmar (2018)

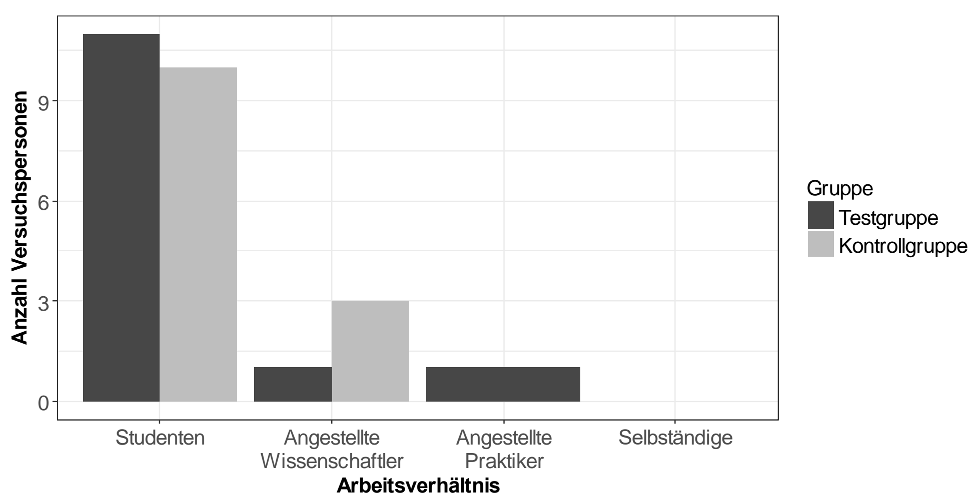


***Abbildung 5-5:* Verteilung der Versuchspersonen anhand des Arbeitsverhältnisses**
Quelle: In Anlehnung an Danciu/Krcmar (2018)

Versuchspersonen mit Schulabschluss bilden den größten Anteil beider Gruppen. Die Testgruppe weist eine Versuchsperson mit Schulausbildung mehr auf als die Kontrollgruppe. Die Kontrollgruppe weist wiederum eine Versuchsperson mit Masterabschluss mehr auf als die Testgruppe. Am Experiment nahmen keine Versuchspersonen mit Doktorgrad teil.

Hinsichtlich des Arbeitsverhältnisses bilden Studenten den größten Anteil der Versuchspersonen beider Gruppen. Die Testgruppe weist einen Studenten mehr auf als die Kontrollgruppe. Die Kontrollgruppe wiederum weist zwei wissenschaftliche Mitarbeiter mehr auf als die Testgruppe. Ein angestellter Praktiker wurde der Testgruppe zugeordnet. Am Experiment nahmen auch keine Versuchspersonen mit dem Arbeitsverhältnis eines Selbständigen teil. Jeweils eine Versuchsperson der Test- und Kontrollgruppe haben den Status eines Studenten obwohl diese schon einen Masterabschluss besitzen.

5.4.2 Verteilung der Versuchspersonen anhand der Programmierkenntnisse

Die Verteilung der Versuchspersonen anhand der Programmierkenntnisse auf die Experimentgruppen ist Abbildung 5-6 dargestellt.

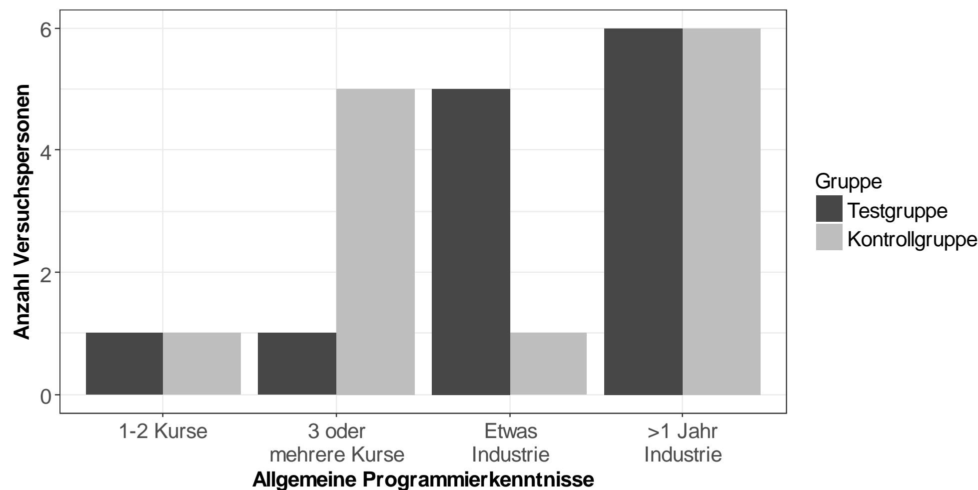


Abbildung 5-6: ***Verteilung der Versuchspersonen anhand der allgemeinen Programmierkenntnisse***
Quelle: Eigene Darstellung

Insgesamt haben zwei Versuchspersonen, deren Programmierkenntnisse auf ein bis zwei besuchte Kurse beschränkt sind, teilgenommen. Versuchspersonen mit über einem Jahr Industrieerfahrung bilden den größten Anteil beider Gruppen. Die Test- und Kontrollgruppe weisen dieselbe Anzahl Versuchspersonen mit der ersten und vierten Erfahrungsstufe auf. Ein Ungleichgewicht zwischen den Gruppen ist hinsichtlich der zweiten und dritten Erfahrungsstufe entstanden. Die Kontrollgruppe weist vier Versuchspersonen mit der zweiten Erfahrungsstufe mehr auf als die Testgruppe. Die Testgruppe weist wiederum vier Versuchspersonen mit der dritten Erfahrungsstufe mehr auf als die Kontrollgruppe.

5.4.3 Verteilung der Versuchspersonen anhand der spezifischen Kenntnisse

Die Verteilung der Versuchspersonen anhand spezifischer Kompetenzbereiche auf die Experimentgruppen ist Abbildung 5-7 dargestellt. Am Experiment haben keine Versuchspersonen ohne Erfahrung im Bereich Java teilgenommen. Insgesamt zehn Teilnehmer hatten noch nie Erfahrung im Bereich Java EE gesammelt. Zwei Teilnehmer, die zur Testgruppe zugeordnet wurden, hatten keine Erfahrung mit Eclipse. Die meiste Industrieerfahrung (vierte Stufe) hatten die Teilnehmer in den Bereichen Java und Eclipse. Die Testgruppe weist keine Versuchspersonen mit der vierten Erfahrungsstufe für Java EE und JPA auf.

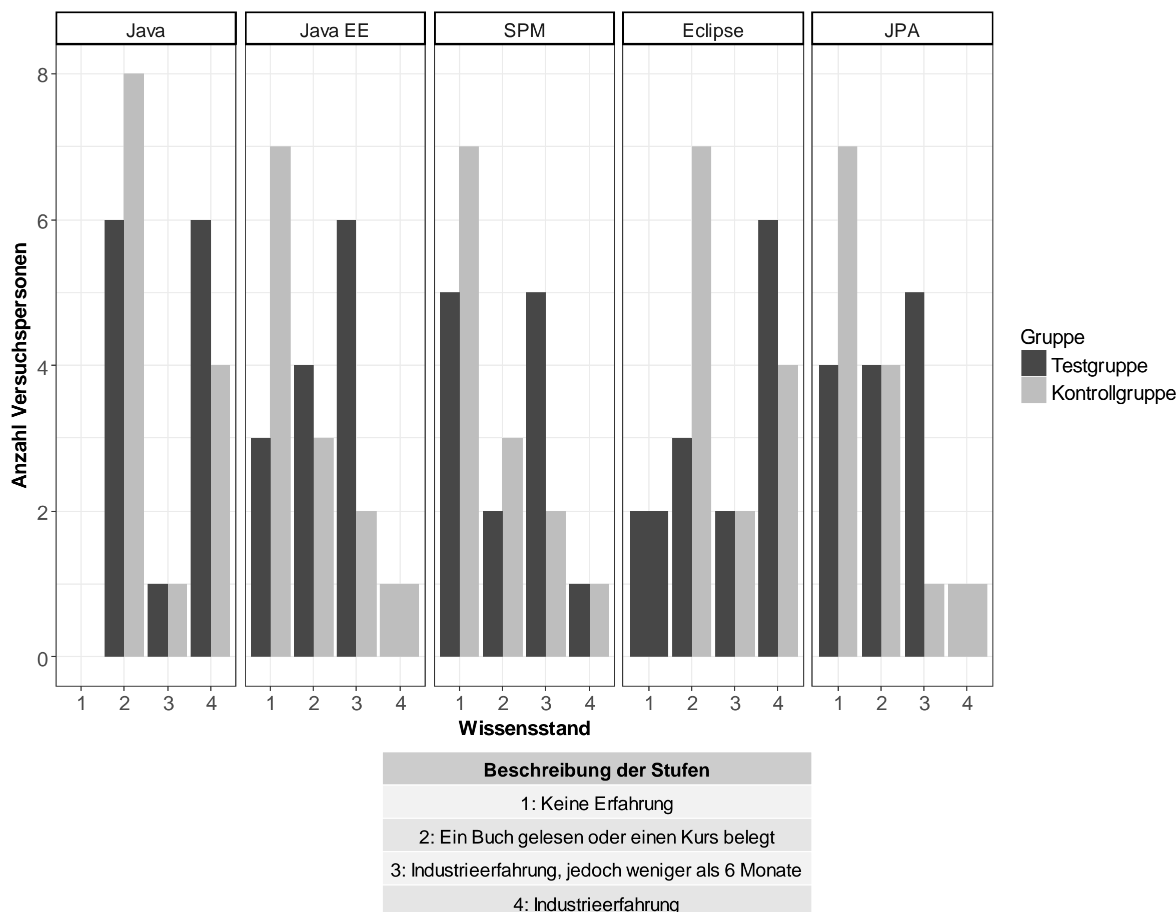


Abbildung 5-7: ***Verteilung der Versuchspersonen anhand spezifischer Kenntnisse***
Quelle: Eigene Darstellung

Versuchspersonen, die bei den allgemeinen Programmierkenntnissen mehr als ein Jahr Industrieerfahrung angaben, hatten tendenziell auch die vierte Erfahrungsstufe im Bereich Java. Wissenschaftler und Praktiker hatten ohne Ausnahme über ein Jahr Industrieerfahrung in der Programmierung und tendenziell auch die vierte Erfahrungsstufe in den Bereichen Java und Java EE. Auch die meisten Teilnehmer mit Masterabschluss hatten über ein Jahr Industrieerfahrung in der Programmierung und tendenziell die vierte Erfahrungsstufe im Bereich Java. Jeweils eine Versuchsperson der Test- und Kontrollgruppe gaben an, über die Anwendung Cargo Tracker informiert zu sein. Alle anderen hatten zuvor keine Kenntnisse darüber.

5.5 Auswertung über die behobenen Performancebugs

Im Folgenden werden die Ergebnisse der Versuchspersonen bei der Behebung von Performancebugs zusammengefasst. Zuerst werden die Gesamtergebnisse der beiden Experimentgruppen beschrieben (siehe Abschnitt 5.5.1). Die Verteilung der behobenen Performancebugs anhand der Erfahrung der Versuchspersonen wird im Abschnitt 5.5.2 beschrieben. Anschließend wird die zugrundeliegende Hypothese getestet (siehe Abschnitt 5.5.3).

5.5.1 Deskriptive Statistik und Datenbereinigung

Versuchspersonen der Testgruppe haben durchschnittlich über drei Mal mehr Performancebugs behoben als die Versuchspersonen der Kontrollgruppe. Neun Versuchspersonen der Kontrollgruppe und drei Teilnehmer der Testgruppe konnten gar keine Bugs beheben. Die maximale Anzahl der behobenen Bugs lag innerhalb der Testgruppe bei 5,5 und innerhalb der Kontrollgruppe bei 3. Statistische Maße zur Beschreibung der beiden Stichproben sind in der Tabelle 5-7 aufgelistet.

Gruppe	Min.	Median	Mittelwert	Max.	StdAbw.
Testgruppe	0,0	1,0	2,192	5,5	2,136376
Kontrollgruppe	0,0	0,0	0,5769	3,0	1,115164

Tabelle 5-7: ***Statistische Maße zur Beschreibung der Stichproben über behobene Performancebugs***
Quelle: Eigene Tabelle

Teilnehmer der Testgruppe haben mindestens eine, maximal fünf und durchschnittlich 2,231 Antwortzeitvorhersagen durchgeführt. Alle Teilnehmer ohne erfolgreiche Optimierungsmaßnahmen haben genau eine Vorhersage durchgeführt. Die meisten Teilnehmer mit mehr als einem behobenen Bug hatten auch mehrere Vorhersagen durchgeführt. Eine Versuchsperson konnte aber bei nur einer durchgeführten Vorhersage fünf Performancebugs beheben. Während dem Experiment hat die Durchführung einer Antwortzeitvorhersage etwa fünf Minuten benötigt.

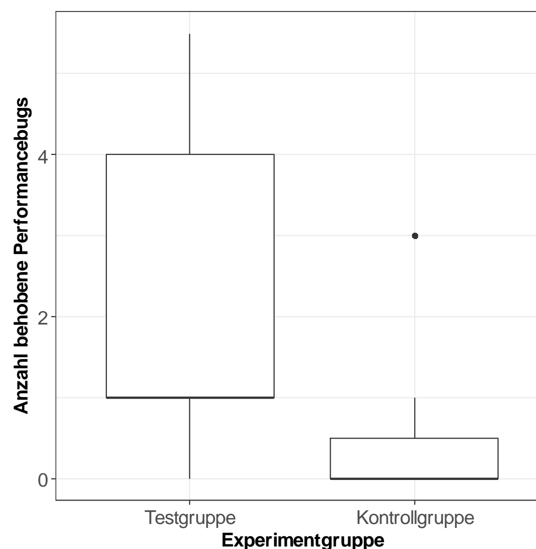


Abbildung 5-8: ***Verteilung der behobenen Performancebugs anhand der Experimentgruppen***
Quelle: In Anlehnung an Danciu/Krcmar (2018)

Eine Zusammenfassung der Lage- und Streuungsmaße der beiden Stichproben ist in Abbildung 5-8 dargestellt. Das untere Quartil der Testgruppe liegt bei 1,0 und das der Kontrollgruppe bei

0,0. Das obere Quartil der Testgruppe liegt bei 4,0 und das der Kontrollgruppe bei 0,5. Das Maximum der Kontrollgruppe wird in der Boxplot-Darstellung als Ausreißer identifiziert, da es das 1,5-fache der Interquartile Range – der Distanz zwischen dem unteren und oberen Quartil – übersteigt (Elliott/Woodward 2007, S. 37). Nach dieser Darstellung liegt das Maximum der Kontrollgruppe bei 1,0, was dem Median der Testgruppe entspricht.

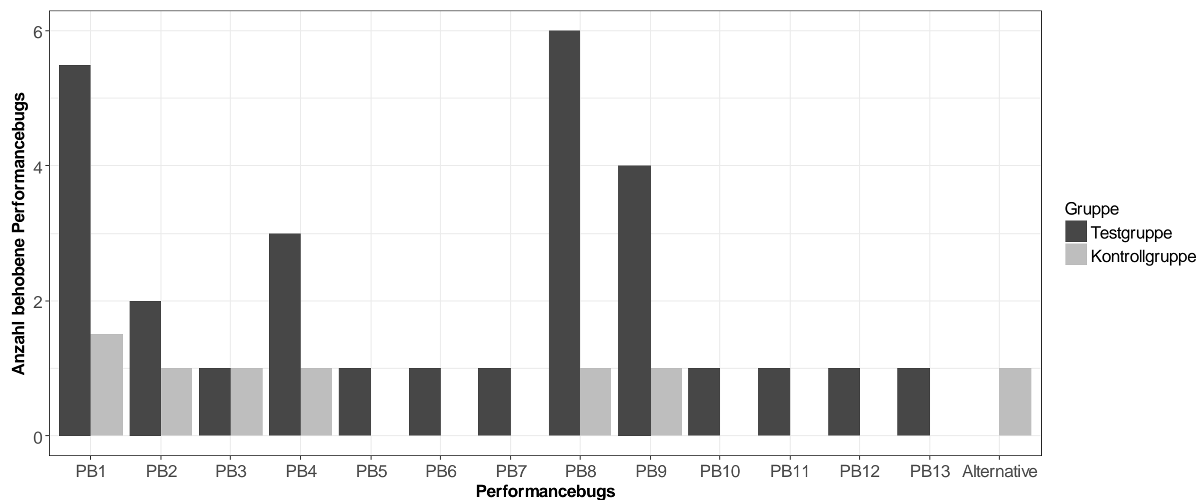


Abbildung 5-9: Behobene Performancebugs je Experimentgruppe
Quelle: In Anlehnung an Danciu/Krcmar (2018)

Die Häufigkeit, mit der Performancebugs durch die Teilnehmer der beiden Experimentgruppen behoben wurden, ist in Abbildung 5-9 dargestellt. Am häufigsten wurde PB1 behoben. Hierbei handelt es sich um eine exzessive Datenabfrage innerhalb der Klasse *BatchProcessingBean*. In der Aufgabenbeschreibung wurde diese Klasse explizit als Einstiegspunkt in die Anwendung genannt. PB1 wurde häufiger identifiziert als PB8, jedoch nicht immer vollständig behoben. Jeweils ein Teilnehmer der Test- und ein Teilnehmer der Kontrollgruppe haben diesen Bug nur teilweise behoben, da ihre Implementierung an dieser Stelle Fehler aufweisen. Ein Teilnehmer der Kontrollgruppe hat eine alternative Lösung für PB1 implementiert – diese wird in Abbildung 5-9 separat aufgeführt. Anstatt die exzessive Datenabfrage mit einer spezifischeren zu ersetzen, wurde der Rückgabewert zwischengespeichert. Dadurch, dass der Wert innerhalb einer Schleife wiederholt benötigt wird, stellt dies auch eine valide Lösung dar.

Am zweithäufigsten wurde PB8 behoben. Insgesamt sieben Teilnehmer haben diesen Bug vollständig behoben. Hierbei handelt es sich auch um eine exzessive Datenabfrage, jedoch in der Klasse *DefaultBookingService*. In derselben Klasse wurde auch PB9 eingebaut, der eine große Ähnlichkeit zu PB8 aufweist. Bis auf die adressierten Objekttypen bestehen die Bugs aus identischen Anweisungen. Die von den Performancebugs betroffenen Methoden folgen im Quelltext auch direkt aufeinander. In über 70% der Fälle haben Versuchspersonen beide Performancebugs behoben. Damit ist PB9 der am dritthäufigsten behobene Bug. PB9 wurde in keinem der Fälle isoliert von PB8 behoben.

Am vierthäufigsten wurde PB4 behoben. Hierbei handelt es sich um eine exzessive Datenabfrage innerhalb der Klasse *DefaultCargoInspectionService*, welche direkt von der *BatchProcessingBean* aufgerufen wird.

Jedes einzelne Performancebug wurde zumindest ein Mal von einem Teilnehmer der Testgruppe behoben. PB5, PB6, PB7, PB10, PB11, PB12 und PB13 wurden von keiner Versuchsperson der Kontrollgruppe behoben. Damit wurden Performancebugs des Typs:

- Auflösung von Caching und
- Einzelabfrage

kein einziges Mal innerhalb der Kontrollgruppe behoben. Sieben Teilnehmer der Kontrollgruppe haben die Klasse *ListCargo* (PB5), drei die Klasse *ExternalRoutingService* (PB12, PB13), einer die Klasse *DefaultBookingServiceFacade* (PB6, PB7) und keiner die Klasse *ItineraryCandidateDtoAssembler* (PB10, PB11) besucht.

Vorhergesagte Antwortzeiten, welche nicht sinnvoll adressiert bzw. reduziert werden können, haben zu keinem unerwünschten Verhalten der Probanden geführt. Bei der Prüfung der Ergebnisse konnten keine gravierenden Anomalien festgestellt werden. Daher werden keine Datensätze aus den Stichproben entfernt.

5.5.2 Verteilung der Ergebnisse anhand der Erfahrung der Versuchspersonen

Die Verteilung der Anzahl behobener Performancebugs anhand des höchsten Abschlusses ist in Abbildung 5-10 dargestellt. Innerhalb der Testgruppe werden mit einem höheren Abschluss mehr Bugs gelöst. Innerhalb der Kontrollgruppe haben Teilnehmer mit Schulabschluss durchschnittlich besser abgeschnitten als Teilnehmer mit Bachelorabschluss. Erst die Stufe des Masterabschlusses schneidete besser ab.

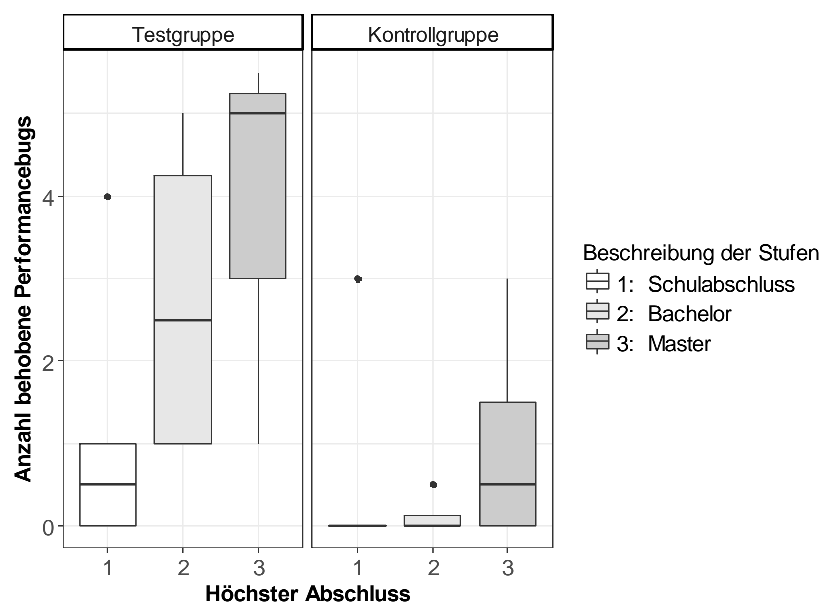


Abbildung 5-10: Verteilung der behobenen Performancebugs anhand des höchsten Abschlusses
Quelle: In Anlehnung an Danciu/Krcmar (2018)

Die Verteilung der Anzahl behobener Performancebugs anhand des Arbeitsverhältnisses ist in Abbildung 5-11 dargestellt. Nur die Testgruppe verfügte über einen Praktiker. Innerhalb beider Gruppen schneideten Wissenschaftler und Praktiker durchschnittlich besser ab als Studenten.

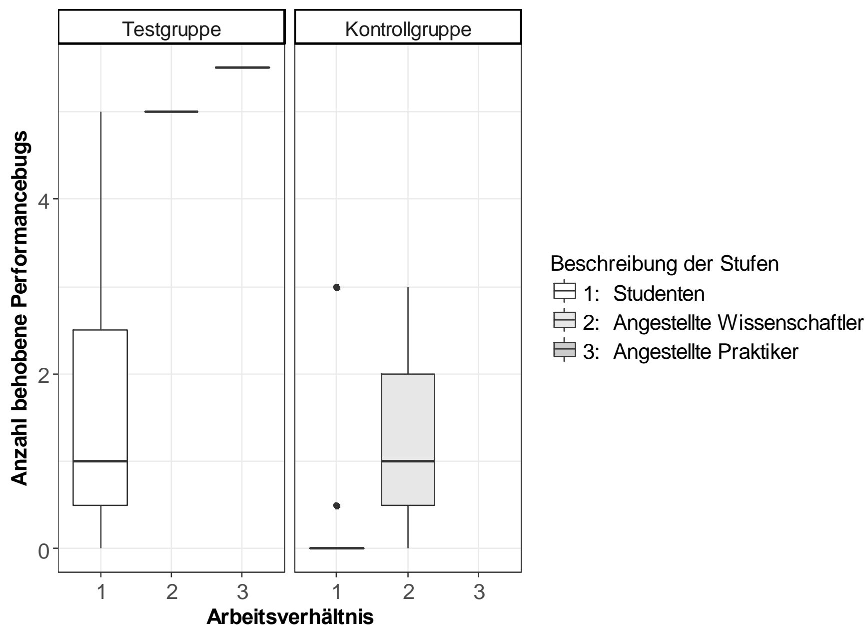


Abbildung 5-11: Verteilung der behobenen Performancebugs anhand des Arbeitsverhältnisses
Quelle: In Anlehnung an Danciu/Krcmar (2018)

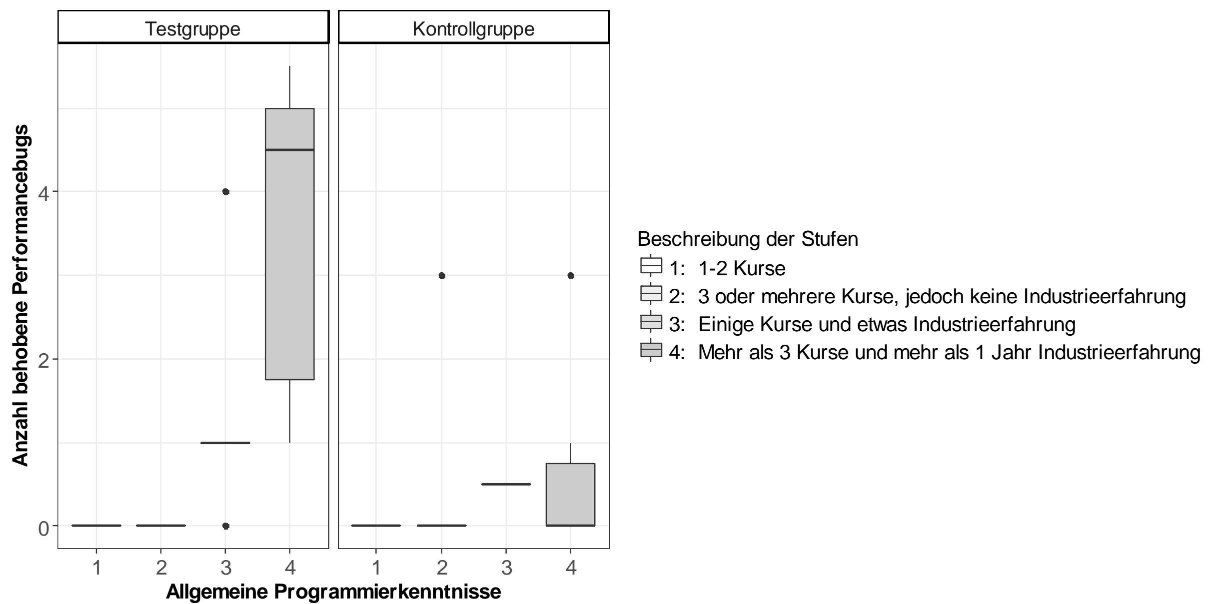


Abbildung 5-12: Verteilung der behobenen Performancebugs anhand der Programmierkenntnisse
Quelle: Eigene Darstellung

Die Verteilung der Anzahl behobener Performancebugs anhand der allgemeinen Programmierkenntnisse ist in Abbildung 5-12 dargestellt. Innerhalb der Testgruppe haben Teilnehmer der ersten und zweiten Erfahrungsstufe gar keine Bugs gelöst. Mit steigender Erfahrung wurden jedoch immer bessere Ergebnisse erzielt. Innerhalb der Kontrollgruppe sind die durchschnittlichen Ergebnisse von Teilnehmer der zweiten, dritten und vierten Erfahrungsstufe sehr ähnlich.

Die Verteilung der Anzahl behobener Performancebugs anhand der spezifischen Kenntnisse innerhalb der Testgruppe ist in Abbildung 5-13 dargestellt. Mit steigender Erfahrung in den Bereichen Java, Java EE und JPA haben Versuchspersonen durchschnittlich bessere Ergebnisse erzielt. Die ersten drei Erfahrungsstufen im Bereich Eclipse haben eine relativ ähnliche Anzahl an Bugs gelöst. Teilnehmer mit Industrieerfahrung in diesem Bereich haben deutlich mehr Bugs gelöst. Die Leistung der Teilnehmer korreliert nicht mit der Erfahrung im Bereich SPM. Teilnehmer der ersten und dritten Erfahrungsstufe haben relativ ähnlich abgeschnitten. Teilnehmer der dritten und vierten Erfahrungsstufe haben weniger Bugs gelöst als Teilnehmer der zweiten Stufe.

Die Verteilung der Anzahl behobener Performancebugs anhand der spezifischen Kenntnisse innerhalb der Kontrollgruppe ist in Abbildung 5-14 dargestellt. Im Gegensatz zur Testgruppe hat die Erfahrungsstufe in den Bereichen Java, Java EE, SPM, Eclipse und JPA weniger Einfluss auf die Leistung der Versuchspersonen. Versuchspersonen der dritten Erfahrungsstufe im Bereich Java hatten schlechtere Ergebnisse als Versuchspersonen der zweiten Stufe. Die Ergebnisse von Teilnehmer der dritten und vierten Stufe sind relativ ähnlich. Dasselbe gilt auch für den Bereich Eclipse. In den Bereichen SPM und JPA ist eine ähnliche Verteilung zwischen der ersten und vierten Stufe sichtbar. Dieselbe Versuchsperson ohne Erfahrung in den Bereichen Java EE, SPM oder JPA hat die maximale Anzahl von drei Bugs innerhalb dieser Gruppe gelöst.

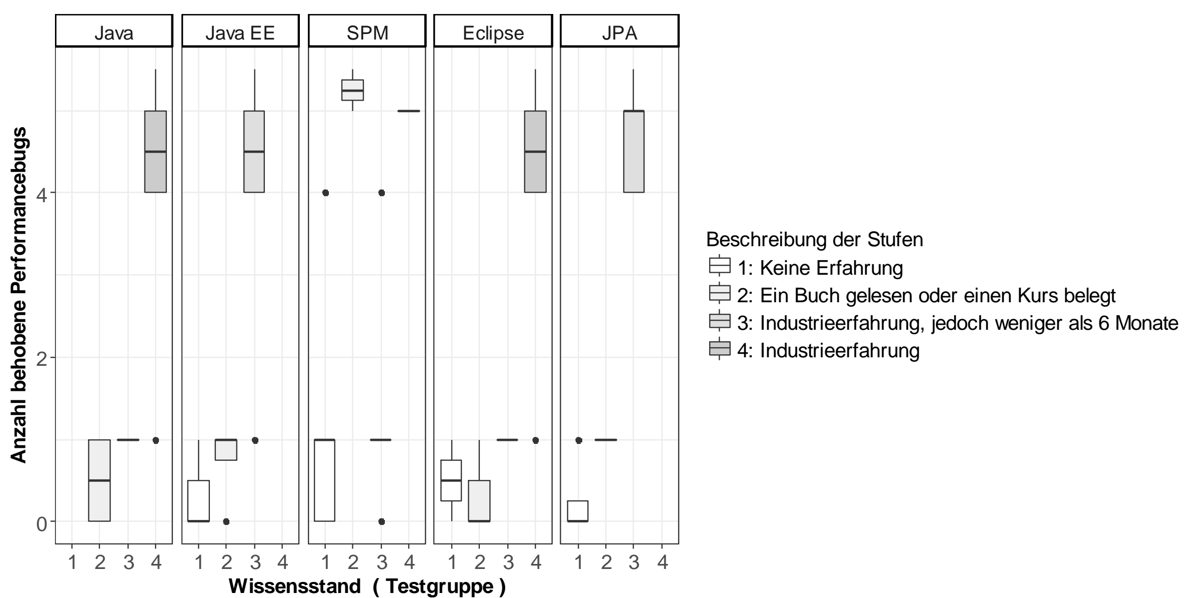


Abbildung 5-13: Verteilung der behobenen Performancebugs anhand der spezifischen Kenntnisse innerhalb der Testgruppe
Quelle: Eigene Darstellung

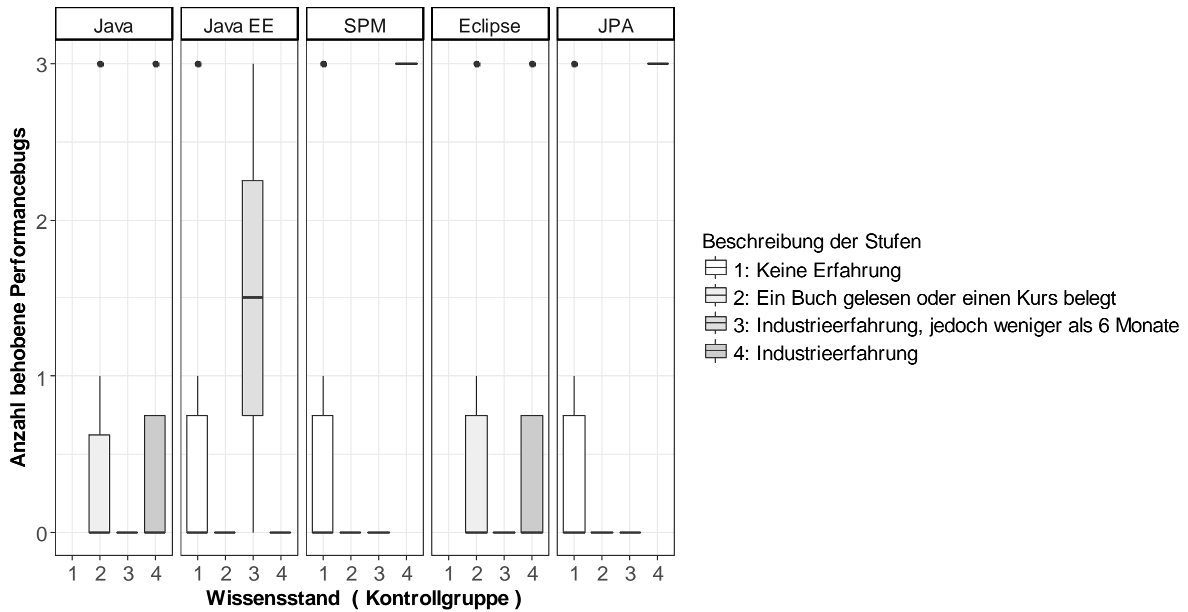


Abbildung 5-14:** **Verteilung der behobenen Performancebugs anhand der spezifischen Kenntnisse innerhalb der Kontrollgruppe
Quelle: Eigene Darstellung

5.5.3 Durchführung des Hypothesentests

Für die Auswahl eines geeigneten Hypothesentests wird zuerst untersucht, ob die Stichproben normalverteilt sind. Das Histogramm der zwei Stichproben über behobene Performancebugs ist in Abbildung 5-15 dargestellt. Für die Kontrollgruppe weist das Histogramm eine eher rechtsschiefe Verteilung auf.

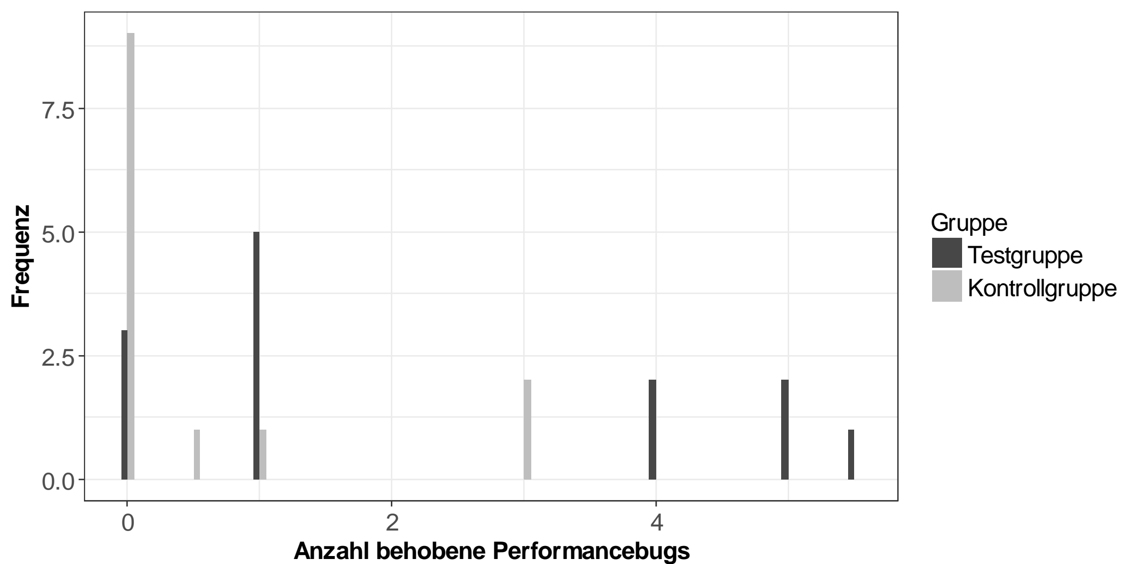


Abbildung 5-15:** **Häufigkeitsverteilung der Stichproben über behobene Performancebugs
Quelle: Eigene Darstellung

Gruppe	Kolmogorov-Smirnov-Test		Shapiro-Wilk-Test	
	D	p-Wert	W	p-Wert
Testgruppe	0,327	0,124	0,8092	0,008795
Kontrollgruppe	0,3898	0,03845	0,5759	0,00004089

Tabelle 5-8: ***Tests auf Normalverteilung der Stichproben über behobene Performancebugs***
Quelle: In Anlehnung an Danciu/Krcmar (2018)

Wie im Abschnitt 5.2.7 beschrieben, wird der Kolmogorov-Smirnov- und der Shapiro-Wilk-Test für die Prüfung auf Normalverteilung angewendet (siehe Tabelle 5-8). Die Nullhypothese beider Tests geht davon aus, dass eine Normalverteilung vorliegt. Der Kolmogorov-Smirnov-Test weist für die Testgruppe einen nicht signifikanten p-Wert von 0,124. Dies deutet darauf hin, dass die Stichprobe normalverteilt ist. Für die Kontrollgruppe weist der Test einen p-Wert kleiner als 0,05 auf und deutet darauf hin, dass die Stichprobe nicht normalverteilt ist. Der Shapiro-Wilk-Test deutet darauf hin, dass keine der Stichproben normalverteilt ist. Aufgrund der nicht vorliegenden Normalverteilung wird der Mann-Whitney-Test (auch Mann-Whitney U genannt) für die Prüfung der Hypothese angewendet. Zum Vergleich sind in Tabelle 5-9 auch die Ergebnisse des t-Test aufgelistet.

Richtung	Mann-Whitney-Test		t-Test		
	U	p-Wert	t	df	p-Wert
Gerichtet (größer)	131	0,005723	2,4168	18,087	0,01322
Ungerichtet	131	0,01145	2,4168	18,087	0,02644

Tabelle 5-9: ***Hypothesentest über behobene Performancebugs***
Quelle: Eigene Tabelle

Der gerichtete Mann-Whitney-Test weist einen p-Wert von 0,005723 und deutet darauf hin, dass die Verteilung der Anzahl der behobenen Performancebugs der Testgruppe sich signifikant von der Verteilung der Kontrollgruppe unterscheidet. Auch die Ergebnisse des t-Test deuten auf einen signifikanten Unterschied zwischen den beiden Verteilungen. Die Hypothese $H_{1\text{-}0}$ wird abgelehnt und die Alternative angenommen.

5.6 Auswertung der Zeitaufwände von Probanden

Im Folgenden wird das Verhalten der Versuchspersonen hinsichtlich der verbrachten Zeit während der Bearbeitung der Aufgabe zusammengefasst. Insbesondere wird der Aspekt der innerhalb der IDE verbrachten Zeit beleuchtet (siehe Abschnitt 5.6.1). Darüber hinaus wird das Verhalten der Versuchspersonen anhand der Besuche und Änderungen von Dateien beschrieben. Abschließend wird die Hypothese, dass die Verfügbarkeit des Plugins für Performancebewusstsein keinen Einfluss auf die innerhalb der IDE verbrachten Zeit hat, aufgestellt und getestet (siehe Abschnitt 5.6.2).

5.6.1 Deskriptive Statistik und Datenberinigung

Versuchspersonen der Testgruppe haben durchschnittlich 10,6 % mehr Zeit innerhalb der IDE verbracht als Versuchspersonen der Kontrollgruppe. Die minimale Zeit in der Testgruppe war dabei 78%. Statistische Maße zur Beschreibung der beiden Stichproben sind in der Tabelle 5-10 aufgelistet.

Gruppe	Min.	Median	Mittelwert	Max.	StdAbw.
Testgruppe	78	91	87,92	96	6,17065
Kontrollgruppe	50	80	77,31	94	12,22282

Tabelle 5-10: ***Statistische Maße zur Beschreibung der Stichproben über die Zeit innerhalb der IDE***
Quelle: Eigene Tabelle

Eine Zusammenfassung der Lage- und Streuungsmaße der beiden Stichproben ist in Abbildung 5-16 dargestellt. Die Testgruppe weist insgesamt eine relativ geringe Streuung auf. Das untere Quartil liegt bei 82% und das obere bei 92%. Die erfolgreichsten Versuchspersonen der Kontrollgruppe haben durchschnittlich mehr Zeit als der Gesamtdurchschnitt der Gruppe innerhalb der IDE verbracht.

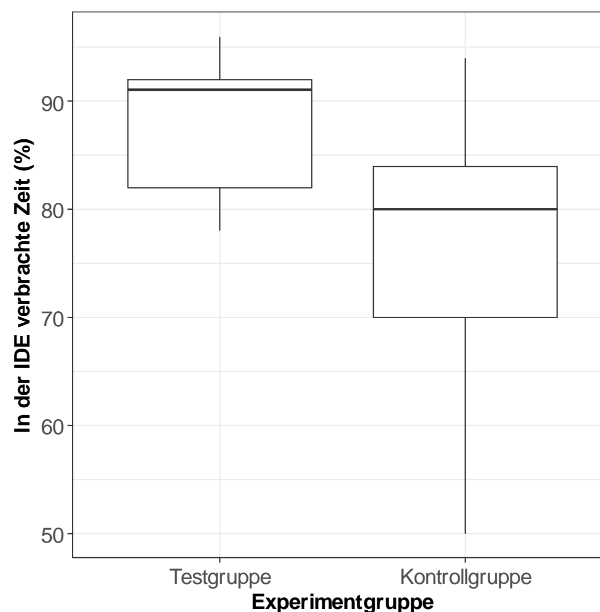


Abbildung 5-16: ***Verteilung der Zeit innerhalb der IDE anhand der Experimentgruppen***
Quelle: Eigene Darstellung

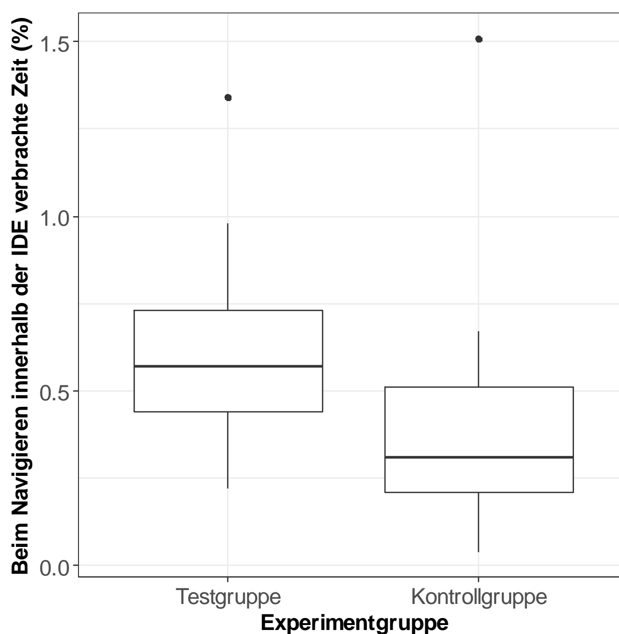


Abbildung 5-17: Verteilung der Zeit beim Navigieren innerhalb der IDE anhand der Experimentgruppen
Quelle: Eigene Darstellung

Eine Zusammenfassung der Lage- und Streuungsmaße über die Zeit für das Navigieren innerhalb der IDE ist in Abbildung 5-17 dargestellt. Probanden der Testgruppe haben durchschnittlich 0,63% ihrer Zeit für das Navigieren innerhalb der IDE verbracht (Median 0,57%). Der Durchschnitt und der Median der Kontrollgruppe betragen 0,41% und 0,31%. Die für das Editieren des Quelltextes verbachte Zeit konnte anhand der protokollierten Ereignisse nicht bestimmt werden. Hierfür hätten alle Tastaturaktionen erfasst werden müssen.

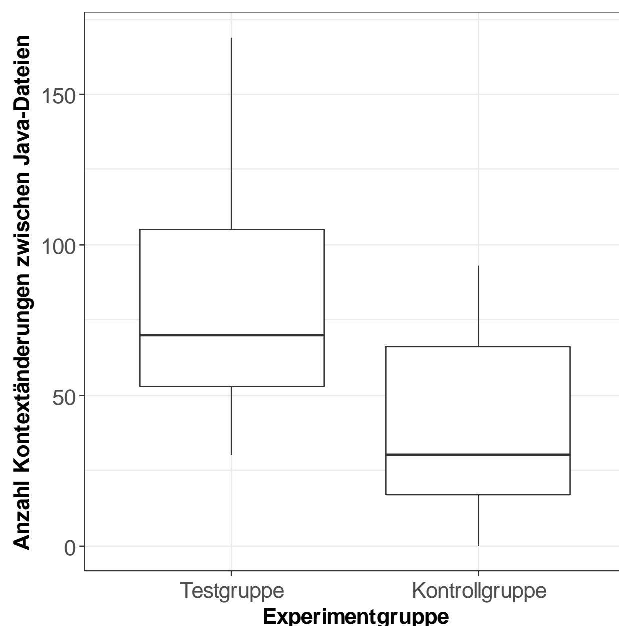


Abbildung 5-18: Verteilung der Anzahl der Kontextänderungen zwischen Java-Dateien
Quelle: Eigene Darstellung

Die Anzahl der Kontextänderungen zwischen Java-Dateien war innerhalb der Testgruppe höher (vgl. Abbildung 5-18). Dasselbe gilt auch für die Anzahl der Kontextänderungen über alle Arten von Dateien.

5.6.2 Durchführung des Hypothesentests

Aufgrund der Alternativhypothese, die eine positive Beeinflussung impliziert, ist die Hypothese gerichtet. Für die Auswahl eines geeigneten Hypothesentests wird zuerst untersucht, ob die Stichproben normalverteilt sind. Die Ergebnisse des Kolmogorov-Smirnov- und des Shapiro-Wilk-Tests sind in der Tabelle 5-11 aufgelistet.

Gruppe	**Kolmogorov-Smirnov-Test**		**Shapiro-Wilk-Test**	
	D	**p-Wert**	**W**	**p-Wert**
Testgruppe	0,2294	0,5005	0,8934	0,1085
Kontrollgruppe	0,138	0,9656	0,9456	0,5333

Tabelle 5-11: ***Tests auf Normalverteilung der Stichproben über der Zeit innerhalb der IDE***
Quelle: Eigene Tabelle

Sowohl der Kolmogorov-Smirnov- als auch der Shapiro-Wilk-Test weisen für beide Stichproben einen nicht signifikanten p-Wert von über 0,05 auf. Damit kann die Normalverteilung der beiden Stichproben angenommen werden. Zur Prüfung der Hypothese wird der t-Test angewendet. Als Vergleich sind in der Tabelle 5-12 auch die Ergebnisse des Mann-Whitney-Tests dargestellt.

Richtung	**t-Test**			**Mann-Whitney-Test**	
	t	**df**	**p-Wert**	**U**	**p-Wert**
Gerichtet (größer)	2,7954	17,744	0,006035	128	0,01268
Ungerichtet	2,7954	17,744	0,01207	128	0,02537

Tabelle 5-12: ***Hypothesentest über die Zeit innerhalb der IDE***
Quelle: Eigene Tabelle

Der gerichtete t-Test weist einen p-Wert von unter 0,025 auf und deutet damit auf einen signifikanten Unterschied zwischen der Verteilung der Zeitanteile beider Stichproben hin. Dieselbe Schlussfolgerung kann auch durch die Anwendung des Mann-Whitney-Tests gezogen werden. Die Hypothese $H_{2\text{-}0}$ wird abgelehnt und die Alternative angenommen.

5.7 Auswertung des methodischen Vorgehens von Probanden

Im Folgenden wird das Verhalten der Versuchspersonen hinsichtlich des methodischen Vorgehens während der Bearbeitung der Aufgabe zusammengefasst. Neben der Anzahl der relevanten Java-Dateien, auf die zugegriffen wurde, wird auch die Anzahl der Änderungen an Dateien beleuchtet (siehe Abschnitt 5.7.1). Abschließend wird die Hypothese, dass die Verfügbarkeit des Plugins für Performancebewusstsein keinen Einfluss auf das methodische Vorgehen hat, getestet (siehe Abschnitt 5.7.2).

5.7.1 Deskriptive Statistik und Datenbereinigung

Versuchspersonen der Testgruppe haben im Durchschnitt annähernd doppelt so viele relevante Java-Dateien besucht wie die Versuchspersonen der Kontrollgruppe. Statistische Maße zur Beschreibung der beiden Stichproben sind in der Tabelle 5-13 aufgelistet. Eine Zusammenfassung der Lage- und Streuungsmaße über die Anzahl der Besuche von relevanten Java-Dateien ist in Abbildung 5-19 dargestellt.

Gruppe	Min.	Median	Mittelwert	Max.	StdAbw.
Testgruppe	2	5	4,385	6	1,38675
Kontrollgruppe	1	2	2,462	4	0,9674179

Tabelle 5-13: ***Statistische Maße zur Beschreibung der Stichproben über die Anzahl der Besuche von relevanten Java-Dateien***
Quelle: Eigene Tabelle

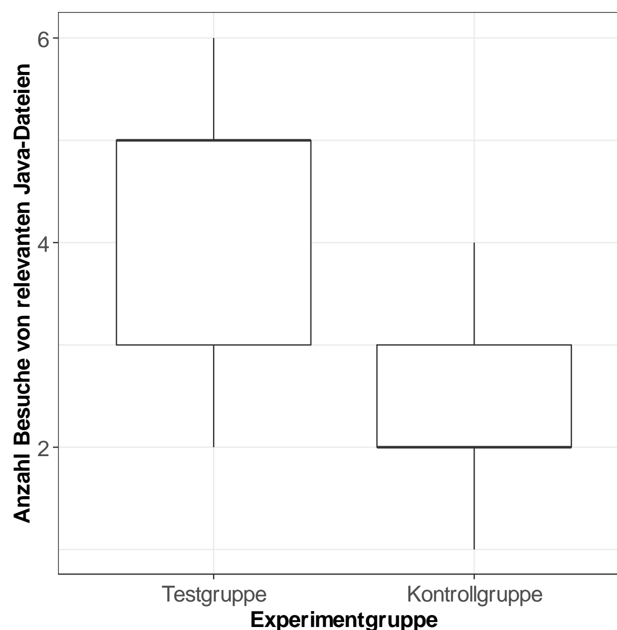


Abbildung 5-19: ***Verteilung der Anzahl der Besuche von relevanten Java-Dateien anhand der Experimentgruppen***
Quelle: Eigene Darstellung

Ein weiterer Indikator für das Vorgehen der Versuchspersonen während der Bearbeitung der Aufgabe ist die gesamte Anzahl der besuchten und die Anzahl der geänderten Dateien. Eine geringe Anzahl besuchter Dateien deuten auf eine Fokussierung auf bestimmte Bereiche der Anwendung hin. Wenige Änderungen an den Dateien deuten auf die fehlende Fähigkeit Optimierungspotenziale zu identifizieren und zu adressieren. Das Öffnen oder Ändern einer Datei im Editor wird durch die IDE protokolliert (vgl. Abschnitt 5.2.5.2).

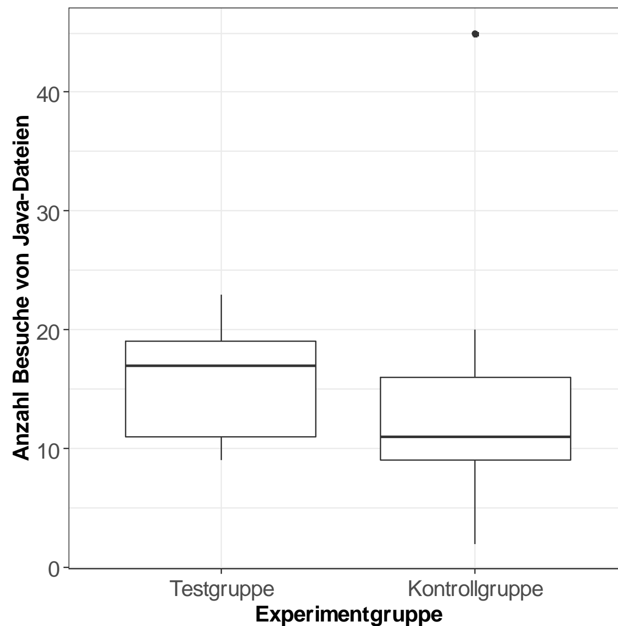


Abbildung 5-20: Verteilung der Anzahl der besuchten Java-Dateien anhand der Experimentgruppen
Quelle: Eigene Darstellung

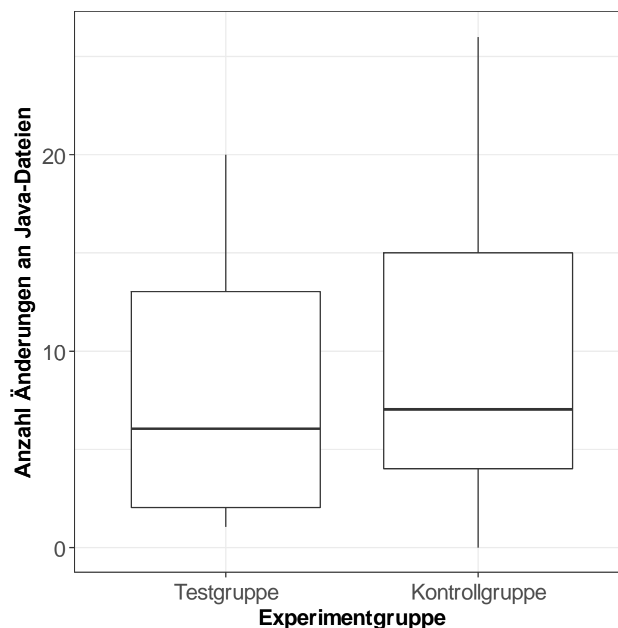


Abbildung 5-21: Verteilung der Anzahl der Änderungen an Dateien anhand der Experimentgruppen
Quelle: Eigene Darstellung

Eine Zusammenfassung der Lage- und Streuungsmaße der beiden Stichproben über die insgesamt besuchten Java-Dateien ist in Abbildung 5-20 dargestellt. Das wiederholte Besuchen ein und derselben Java-Datei wird in diesem Fall nicht berücksichtigt. Die beiden Gruppen weisen eine relativ ähnliche Aktivität auf. Versuchspersonen der Testgruppe haben mindestens neun und durchschnittlich 15 unterschiedliche Dateien im Editor abgerufen. Versuchspersonen mit vier oder mehr behobenen Bugs weisen mindestens 19 Besuche auf. Versuchspersonen der Kontrollgruppe haben mindestens zwei und durchschnittlich 14 unterschiedliche Dateien im Editor abgerufen.

Dem Besuch von Ressourcen wird auch die tatsächliche Aktivität, bzw. die Anzahl der Änderungen an Java-Dateien, gegenübergestellt. Eine Zusammenfassung der Lage- und Streuungsmaße der beiden Stichproben über die Anzahl der Änderungen an Dateien ist in Abbildung 5-21 dargestellt. Auch hier weisen die beiden Gruppen eine ähnliche Aktivität auf. Versuchspersonen der Testgruppe haben mindestens eine Änderung vorgenommen. Durchschnittlich weist die Testgruppe jedoch weniger Änderungen auf. Innerhalb der Testgruppe konnten Versuchspersonen auch mit nur einer Änderung ein Performancebug beheben. Andererseits benötigten Versuchspersonen der Kontrollgruppe mindestens sieben Änderungen, um mindestens ein Bug zu lösen. Innerhalb der Testgruppe waren für die Lösung eines Bugs durchschnittlich etwa vier Änderungen notwendig. Innerhalb der Kontrollgruppe waren etwa 17 Änderungen notwendig.

Ein weiterer Indikator für das Vorgehen der Versuchspersonen während der Bearbeitung der Aufgabe ist die Anzahl der ausgeführten Commands. Eine geringe Anzahl ausgeführter Commands kann dabei auf fehlende Kenntnisse im Umgang mit der IDE oder die fehlende Fähigkeit Optimierungspotenziale zu identifizieren und zu adressieren hinweisen.

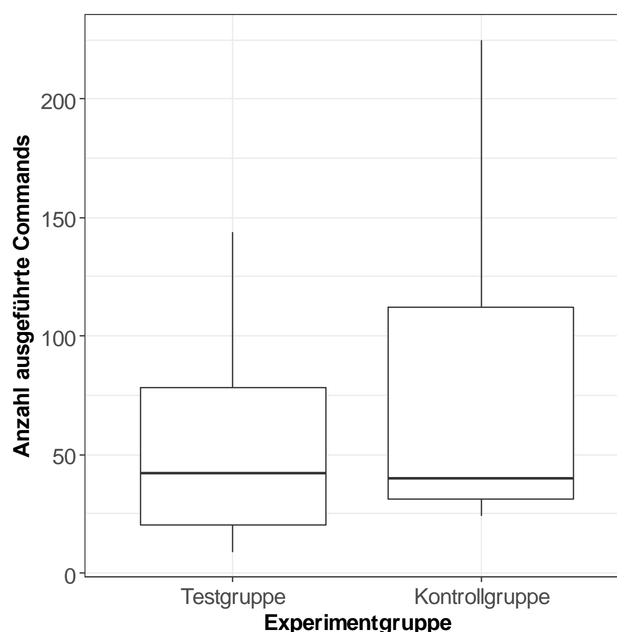


Abbildung 5-22: Verteilung der Anzahl ausgeführter Commands anhand der Experimentgruppen
Quelle: Eigene Darstellung

Eine Zusammenfassung der Lage- und Streuungsmaße der beiden Stichproben über die ausgeführten Commands ist in Abbildung 5-22 dargestellt. Versuchspersonen der Kontrollgruppe haben durchschnittlich mehr Commands ausgeführt. Erfolgreiche Teilnehmer dieser Gruppe haben dabei überdurchschnittlich viele Commands ausgeführt. Innerhalb der Testgruppe wurden für die Lösung eines Bugs durchschnittlich etwa 31 Commands ausgeführt. Innerhalb der Kontrollgruppe waren etwa 137 Commands nötig.

5.7.2 Durchführung des Hypothesentests

Für die Auswahl eines geeigneten Hypothesentests wird zuerst untersucht, ob die Stichproben normalverteilt sind. Das Histogramm der zwei Stichproben über die Anzahl der relevanten Klassen, auf die zugegriffen wurde, ist in Abbildung 5-23 dargestellt. Für die Testgruppe weist das Histogramm eine eher linksschiefe Verteilung auf.

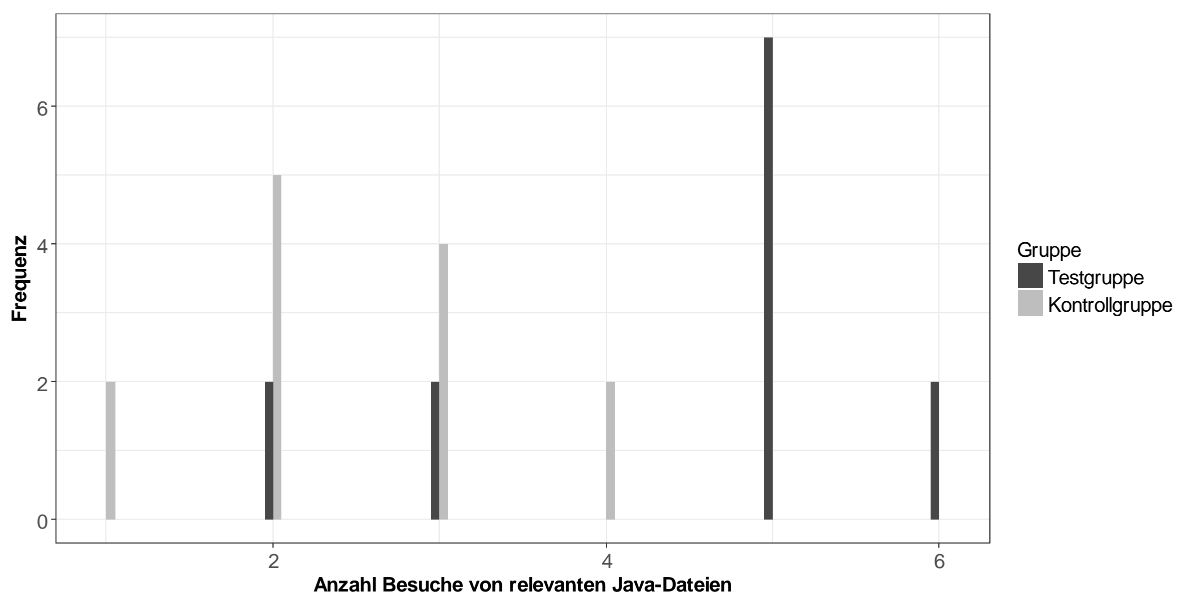


Abbildung 5-23: Häufigkeitsverteilung der Stichproben über die Anzahl relevanter Java-Dateien, auf die zugegriffen wurde
Quelle: Eigene Darstellung

Gruppe	**Kolmogorov-Smirnov-Test**		**Shapiro-Wilk-Test**	
	D	**p-Wert**	**W**	**p-Wert**
Testgruppe	0,3637	0,06418	0,80024	0,006861
Kontrollgruppe	0,22181	0,5446	0,90113	0,1386

Tabelle 5-14: Tests auf Normalverteilung der Stichproben über die Anzahl der Besuche von relevanten Java-Dateien
Quelle: Eigene Tabelle

Für die Prüfung auf Normalverteilung wird der Kolmogorov-Smirnov- und der Shapiro-Wilk-Test angewendet (siehe Tabelle 5-14). Die Nullhypothese beider Tests geht davon aus, dass eine Normalverteilung vorliegt. Der Kolmogorov-Smirnov-Test weist für beide Stichproben einen nicht signifikanten p-Wert von über 0,05 auf und damit auf eine Normalverteilung hin. Der Shapiro-Wilk-Test weist für die Testgruppen einen p-Wert kleiner als 0,05 auf und deutet darauf hin, dass die Stichprobe nicht normalverteilt ist. Aufgrund der nicht vorliegenden Normalverteilung wird der Mann-Whitney-Test für die Prüfung der Hypothese angewendet. Zum Vergleich sind in Tabelle 5-15 auch die Ergebnisse des t-Test aufgelistet.

Richtung	**Mann-Whitney-Test**		**t-Test**		
	U	**p-Wert**	**t**	**df**	**p-Wert**
Gerichtet (größer)	144	0,0008661	4,1007	21,443	0,0002464
Ungerichtet	144	0,001732	4,1007	21,443	0,0004927

Tabelle 5-15: ***Hypothesentest über die Anzahl relevanter Java-Dateien, auf die zugegriffen wurde***
Quelle: Eigene Tabelle

Der gerichtete Mann-Whitney-Test weist einen p-Wert von 0,0008661 und deutet darauf hin, dass die Verteilung der Anzahl der Besuche relevanter Java-Dateien der Testgruppe sich signifikant von der Verteilung der Kontrollgruppe unterscheidet. Auch die Ergebnisse des t-Test deuten auf einen signifikanten Unterschied zwischen den beiden Verteilungen hin. Die Hypothese $H_{3\text{-}0}$ wird abgelehnt und die Alternative angenommen.

5.8 Auswertung der Benutzerfreundlichkeit

Im Folgenden wird die Bewertung der Versuchspersonen über die Benutzerfreundlichkeit des Plugins und der IDE zusammengefasst. Zuerst werden die Gesamtergebnisse im Abschnitt 5.8.1 beschrieben. Abschließend wird die Hypothese, dass der Einsatz des Plugins für Performancebewusstsein keinen Einfluss auf die Benutzerfreundlichkeit hat, aufgestellt und getestet (siehe Abschnitt 5.8.2).

5.8.1 Deskriptive Statistik und Datenberinigung

Ein Item zur Bewertung der Benutzerfreundlichkeit wurde durch eine Versuchsperson der Kontrollgruppe nicht beantwortet. Der entsprechende Datensatz wird daher aus der Stichprobe entfernt. Im Vergleich zur IDE im Allgemeinen wurde die Benutzerfreundlichkeit des Plugins für Performancebewusstsein durchschnittlich besser bewertet. Statistische Maße zur Beschreibung der beiden Stichproben sind in der Tabelle 5-16 aufgelistet.

Gruppe	Min.	Median	Mittelwert	Max.	StdAbw.
Testgruppe	36	50	49,17	60	7,444746
Kontrollgruppe	22	38	39	55	9,293204

Tabelle 5-16: Statistische Maße zur Beschreibung der Stichproben über die Benutzerfreundlichkeit
Quelle: Eigene Tabelle

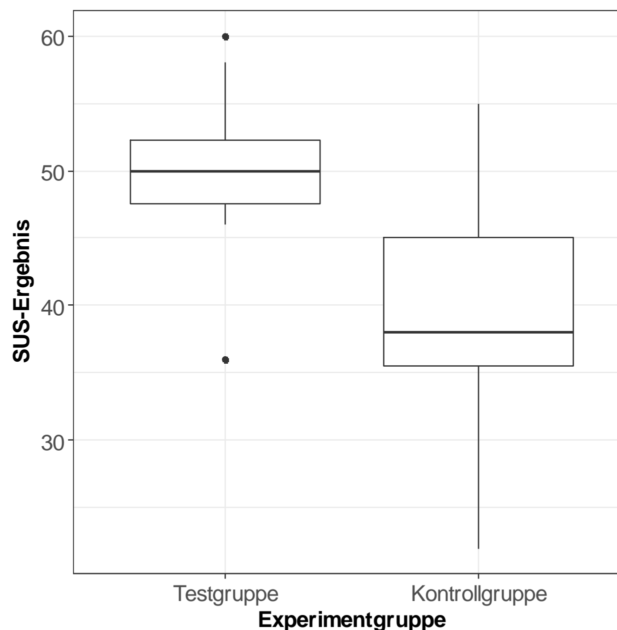


Abbildung 5-24: Verteilung der Benutzerfreundlichkeit anhand der Experimentgruppen
Quelle: Eigene Darstellung

Eine Zusammenfassung der Lage- und Streuungsmaße der beiden Stichproben ist in Abbildung 5-24 dargestellt. Das untere Quartil der Testgruppe liegt bei 47,5 und das obere bei 52,25. Der minimale und der maximale Wert werden als Ausreißer ausgewiesen. Das untere Quartil der Kontrollgruppe liegt bei 35,5 und das obere bei 45.

5.8.2 Durchführung des Hypothesentests

Über den SUS-Fragebogen wurde die Zufriedenheit der Versuchspersonen beim Einsatz eines Werkezugs – des Plugins oder der IDE – zur Bearbeitung derselben Aufgabe erhoben. Ausgehend von den zwei Stichproben wird folgende Hypothese aufgestellt:

- H_0: Die Nutzung des Ansatzes für Performancebewusstsein hat keinen Einfluss auf die Zufriedenheit der Versuchspersonen während der Optimierung von Komponenten.
- H_1: Die Nutzung des Ansatzes für Performancebewusstsein hat einen positive Einfluss auf die Zufriedenheit der Versuchspersonen während der Optimierung von Komponenten.

Gruppe	Kolmogorov-Smirnov-Test		Shapiro-Wilk-Test	
	D	p-Wert	W	p-Wert
Testgruppe	0,1877	0,7915	0,91	0,2135
Kontrollgruppe	0,1648	0,9003	0,9804	0,9851

Tabelle 5-17: ***Tests auf Normalverteilung der Stichproben über Benutzerfreundlichkeit***
Quelle: Eigene Tabelle

Richtung	t-Test			Mann-Whitney-Test	
	t	df	p-Wert	U	p-Wert
Gerichtet (größer)	2,9577	21	0,003757	114	0,007557
Ungerichtet	2,9577	21	0,007513	114	0,01511

Tabelle 5-18: ***Hypothesentest über Benutzerfreundlichkeit***
Quelle: Eigene Tabelle

Aufgrund der Alternativhypothese, die eine positive Beeinflussung adressiert, ist die Hypothese gerichtet. Für die Auswahl eines geeigneten Hypothesentests wird zuerst untersucht, ob die Stichproben normalverteilt sind. Die Ergebnisse des Kolmogorov-Smirnov- und des Shapiro-Wilk-Tests sind in der Tabelle 5-17 aufgelistet. Sowohl der Kolmogorov-Smirnov- als auch der Shapiro-Wilk-Test weisen für beide Stichproben einen nicht signifikanten p-Wert von über 0,05 auf. Damit kann die Normalverteilung der beiden Stichproben angenommen werden. Zur Prüfung der Hypothese wird der t-Test angewendet. Als Vergleich sind in der Tabelle 5-18 auch die Ergebnisse des Mann-Whitney-Tests dargestellt.

Der gerichtete t-Test weist einen p-Wert von unter 0,025 auf deutet damit auf einen signifikanten Unterschied zwischen der Verteilung der SUS-Ergebnisse beider Stichproben hin. Dieselbe Schlussfolgerung kann auch durch die Anwendung des Mann-Whitney-Tests gezogen werden. Die Hypothese H_0 wird abgelehnt und die Alternative angenommen.

5.9 Interpretierung der Ergebnisse

Im Folgenden werden die Ergebnisse des Experiments interpretiert und deren Validität diskutiert.

5.9.1 Evaluation der Ergebnisse und Implikationen

Bei Verfügbarkeit von Performancebewusstsein haben Versuchspersonen mit einer höheren Qualifizierung und einer höheren Erfahrungsstufe auch bessere Ergebnisse erzielt. Die einzige Ausnahme stellt die Erfahrung im Bereich SPM dar. Innerhalb der Kontrollgruppe konnte dieser Zusammenhang nicht eindeutig festgestellt werden. Die fehlende Vertrautheit mit der Anwendung hindert Versuchspersonen ihr Wissen produktiv einzusetzen. Durch die Verfügbarkeit von Hinweisen über die Antwortzeit von Methoden im Quelltext können diese fokussierter

vorgehen. Auch unerfahrene Teilnehmer werden damit befähigt Optimierungspotenziale zu identifizieren. Die Merkmale des Qualifikationsniveaus im Sinne des höchsten Abschlusses, des Arbeitsverhältnisses und der Programmierkenntnisse entsprechen der Ordinalskala und werden daher nicht im Zusammenhang mit mathematischen Operationen eingesetzt (siehe auch Abschnitt 5.2.7). Unter Annahme eines gleichen Abstandes zwischen den Werten des Qualifikationsniveaus, wie bei der Intervallskala, weist der Korrelationstest eine hohe Korrelation zwischen der Anzahl der behobenen Bugs und der Qualifizierung nur bei der Verfügbarkeit des Awareness Plugins.

Teilnehmer der Kontrollgruppe haben vor allem Bugs auf der obersten Ebene, dem Einstiegspunkt in die Anwendung, behoben. PB4 und die benachbarten PB8 und PB9 bilden dabei die Ausnahme. Durch den direkten Aufruf der Klasse *DefaultCargoInspectionService* (PB4) durch die *BatchProcessingBean*, stich diese auch stärker hervor, als andere Klassen. Andere direkt aufgerufene Klassen sind *ListCargo* und *DefaultBookingServiceFacade*. Keins der hier vororteten Performancebugs wurde durch die Kontrollgruppe behoben. Ein möglicher Grund dafür ist, dass die Mehrheit dieser Bugs keine exzessive Datenabfrage darstellen und komplizierter zu identifizieren sind. Die einzige exzessive Datenabfrage wird in der Methode *loadCargoForRouting* der Klasse *DefaultBookingServiceFacade* durchgeführt. Diese einzelne Methode wird aber nur indirekt von der *BatchProcessingBean* aufgerufen. PB8 und PB9 wurden jeweils ein Mal von demselben Teilnehmer der Kontrollgruppe identifiziert.

Bei Verfügbarkeit von Performancebewusstsein haben Versuchspersonen durchschnittlich 10,6% mehr Zeit innerhalb der IDE verbracht. Die Auswertung der Änderungen innerhalb der Kontrollgruppe hat ergeben, dass einige Versuchspersonen durch das Einsetzten von Messpunkten im Quelltext die Antwortzeit der Komponenten analysiert haben. Messergebnisse wurden extern auf die Systemkonsole oder auf die Benutzeroberfläche ausgegeben. Durch die Suche und Interpretierung dieser Nachrichten verbringen Versuchspersonen zusätzliche Zeit außerhalb der IDE. Der Test auf Korrelation zwischen der Anzahl der behobenen Fehler und der Zeit, die innerhalb der IDE verbracht wird, zeigt jedoch keine signifikante Korrelation an. Minelli et al. (2015) melden eine durchschnittliche Zeit außerhalb der IDE von 8% (im Gegensatz zu ~ 18%, die im Rahmen dieses Experiments gemessen wurden) und identifizieren eine Korrelation mit der Zeit, die für das Verständnis aufgewendet wurde. Aufgrund ihrer Beobachtung könnte argumentiert werden, dass die Probanden der Kontrollgruppe mehr Zeit für das Verstehen aufgewendet haben und ihnen somit weniger Zeit für die Behebung von Fehlern geblieben ist. Dem steht jedoch entgegen, dass eine Korrelation zwischen der Effektivität von Entwicklern und dem Zeitaufwand für Navigation oder der Anzahl der Kontextänderungen zwischen Java-Klassen nicht feststellbar ist.

Probanden der Testgruppe besuchten signifikant mehr relevante Klassen als Probanden der Kontrollgruppe. Der Test für die Korrelation zwischen der methodischen Untersuchung der relevanten Klassen und der Effektivität der Probanden der Testgruppe ergibt einen PCC von 0,5636469 und einen p-Wert von 0,04484, was auf eine signifikante lineare Korrelation hinweist. Da viele Probanden der Kontrollgruppe überhaupt keine Performancebugs behoben haben, sind die Ergebnisse des Korrelationstests für diese Gruppe nicht belastbar. Auf der Grundlage der Ergebnisse des Korrelationstests wird abgeleitet, dass das Awareness Plugin eine methodische Untersuchung von Quellcode im Rahmen von Wartungsarbeiten unterstützt und somit die Effektivität der Entwickler erhöht.

Versuchspersonen der Testgruppe weisen eine viel höhere Effizienz auf. Je erzielte Optimierung haben diese durchschnittlich nur 5 Änderungen benötigt. Innerhalb der Kontrollgruppe war mehr als die doppelte Anzahl an Änderungen notwendig um ein Bug zu lösen. Auch hinsichtlich der ausgeführten Commands konnten Teilnehmer der Testgruppe mehr Bugs bei geringerer Aktivität lösen.

Eine mögliche Erklärung für die längere Verweildauer innerhalb der IDE und die höhere Effizienz bei Teilnehmer der Testgruppe ist, dass durch die Verfügbarkeit von Performancebewusstsein die Aufwände für die Identifizierung von Optimierungspotenzialen und für die Verifizierung der Auswirkungen von Optimierungsmaßnahmen reduziert werden.

Aufgrund der relativ schlechten Performance der Rechner im Computerlabor hat eine Antwortzeitvorhersage etwa fünf Minuten gedauert. Innerhalb der verfügbaren Zeit zur Bearbeitung der Aufgabe konnten somit maximal acht Vorhersagen durchgeführt werden. Auf einem handelsüblichen Geschäftslaptop erfolgt dieselbe Vorhersage mehr als doppelt so schnell. Es ist davon auszugehen, dass die Versuchspersonen auf leistungsfähigeren Rechnern bessere Ergebnisse erzielt hätten. Bei der reinen Bearbeitung des Quelltextes weisen die Rechner des Computerlabors nicht derart gravierende Unterschiede auf.

5.9.2 Validität des Experiments

Einflüsse auf die Validität des Experiments nach Wohlin et al. (2012, S. 104ff) werden im Folgenden beschrieben.

5.9.2.1 Statistische Validität

Die statistische Validität bezieht sich auf den Einfluss der gewählten mathematischen Methoden auf die abgeleiteten Schlussfolgerungen.

Aussagekraft und Annahmen der Hypothesentests

Die Auswahl der statistischen Tests für die Prüfung von Hypothesen orientiert sich an den Richtlinien von Wohlin et al. (2012, S. 137). Zugrundeliegende Annahmen der Hypothesentests wurden dabei berücksichtigt. Die Annahme der Normalverteilung des t-Test wurde in allen Fällen durch die Anwendung des Kolmogorov-Smirnov- und des Shapiro-Wilk-Tests geprüft. Annahmen über das Skalenniveau der verarbeiteten Metriken wurden explizit adressiert. Bei der Prüfung von Hypothesen wurden jeweils zwei unterschiedliche Tests angewendet und die Ergebnisse einander gegenübergestellt.

Zuverlässigkeit der Messungen

Zur Sicherstellung der Zuverlässigkeit der Messung des Wissensstandes von Versuchspersonen orientiert sich die Formulierung der Fragen und die Spezifikation der Antwortmöglichkeiten am Beispiel von Wohlin et al. (2012, S. 205). Zur Erhebung der Benutzerzufriedenheit wurde der standardisierte SUS-Fragebogen eingesetzt. Die Aktivität der Versuchspersonen und deren Änderungen am Quelltext wurden automatisiert durch die IDE erfasst. Metriken wurden mit Hilfe eines Werkzeugs automatisiert berechnet. Optimierungsmaßnahmen wurden anhand eines festgelegten Schemas bewertet. Dabei wurden alternative und unvollständige Lösungsansätze explizit berücksichtigt.

Zuverlässigkeit der Treatments

Die Zuverlässigkeit des Treatments über die zwei Experimentgruppen und über den verschiedenen Iterationen wird durch mehrere Maßnahmen adressiert. Die IDE wird mit Hilfe eines automatisierten Build-Prozesses erstellt. Beide Treatments basieren auf denselben Binärdateien. Für die Kontrollgruppe wurde das Plugin für Performancebewusstsein jedoch deaktiviert. Bei der Installation der IDEs auf den Rechnern werden immer dieselben Binärdateien verwendet. Nach einer Iteration wird die komplette Entwicklungsumgebung gelöscht und anschließend neu aufgesetzt. Für die konsistente Anwendung der Entwicklungsumgebung wurde eine Anleitung bereitgestellt.

Zufällige externe Einflüsse

Externe Einflüsse können die Ergebnisse des Experiments beeinflussen. Bei der Durchführung mehrerer Iterationen können diese Einflüsse auch unregelmäßig auftreten. Das Experiment wurde in einem Zeitraum von einem Monat an verschiedenen Tagen und zu unterschiedlichen Uhrzeiten im selben Computerlabor wiederholt. Ein Einfluss der Uhrzeit kann nicht ausgeschlossen werden. Bei dem Computerlabor handelt es sich um einen abgeschlossenen Raum. Während einer Iteration haben zwischen zwei und zehn Versuchspersonen teilgenommen. Alle Iteration wurden von derselben Person koordiniert. Teilnehmer wurden angewiesen beim Stellen von Fragen so leise wie möglich zu sprechen, damit andere Teilnehmer nicht aufgrund der besprochenen Inhalte in ihrem Verhalten beeinflusst werden. Keine der acht Iterationen wurde durch besondere Zwischen- bzw. Vorfälle unterbrochen. Sieben dieser Iterationen wurden mit Versuchspersonen beider Gruppen durchgeführt.

5.9.2.2 Interne Validität

Die interne Validität bezieht sich auf Einflüsse auf die unabhängige Variable des Experiments.

Selektion der Versuchspersonen

Die Selektion von Versuchspersonen stellt einen wichtigen Einflussfaktor auf die interne Validität dar. Durch die Anwendung von Convenience Sampling wurden Studenten der Fakultät für Informatik der Technischen Universität München, wissenschaftliche Mitarbeiter derselben Fakultät und der fortiss GmbH und Softwareentwickler zur freiwilligen Teilnahme am Experiment aufgefordert. Für die Teilnahme wurden zumindest grundlegende Programmierkenntnisse vorausgesetzt. Am Experiment haben Bachelor- und Masterstudenten aller relevanten Studiengänge teilgenommen. Aufgrund der freiwilligen Teilnahme kann von einer relativ hohen Motivation der Versuchspersonen ausgegangen werden (Wohlin et al. 2012, S. 107). Teilnehmer wurden zufällig auf Experimentgruppen verteilt. Durch die Anwendung von Blockbildung wurde der Einfluss der Qualifikation der Versuchspersonen kontrolliert. Die Auswertung der nachgelagerten Erhebung der Kenntnisse in relevanten Bereichen zeigt, dass die ausgewählten Teilnehmer alle Erfahrungsstufen abdecken. Ein Einfluss der Vertrautheit mit der Beispielanwendung auf die erzielten Ergebnisse konnte bei der Auswertung ausgeschlossen werden.

Mehrfache Zuordnung von Treatments

Versuchspersonen dürfen einmalig am Experiment teilnehmen. Durch die Existenz einer Kontrollgruppe wird jede Versuchsperson genau einem Treatment zugeordnet. Während dem Experiment sind keine Versuchspersonen ausgefallen.

Instrumentierung

Eine negative Beeinflussung der Ergebnisse durch die verwendeten Objekte und die Instrumentierung wird durch die Auswahl der bespielhaften Anwendung und der Gestaltung der Dokumente und des Treatments adressiert. Die Anwendung implementiert eine moderne Benutzeroberfläche und basiert auf der neuesten Java-EE-Version. Die Kompilierung und das Deployment der Anwendung sind mit Hilfe eines Skriptes vollständig automatisiert. Die Anleitung beschreibt die Funktionen der Anwendung und des Plugins anschaulich durch Screenshots. Die Inhalte sind nach Themenfelder abgegrenzt und mit Symbolen zur Beschreibung der jeweiligen Aktivität versehen. Die IDE basiert auf der neuesten Eclipse-Version 4.5.2. Die Bedrohung einer zu anspruchsvollen Aufgabenstellung kann dadurch wiederlegt werden, dass einige Versuchspersonen der Test- und Kontrollgruppe bis über 40% bzw. 20% der Performancebugs beseitigen konnten.

Interaktion zwischen Subjekten

Nehmen Versuchspersonen beider Gruppen gleichzeitig im selben Raum am Experiment teil, können sich diese gegenseitig beeinflussen. Personen aus der Kontrollgruppe könnten die Existenz des Plugins entdecken oder das Verhalten der Personen aus der Testgruppe imitieren. Die Existenz zweier Gruppen wurde den Teilnehmern nicht kommuniziert. Zur Verhinderung der Diffusion und Imitation des Verhaltens zwischen den beiden Experimentgruppen wurden zwei unterschiedliche Versionen der Anweisung und des Fragebogens ausgeteilt. Versuchspersonen der Kontrollgruppe waren sich dadurch nicht über die Existenz eines Plugins für die Durchführung von Antwortzeitvorhersagen bewusst. Auch durch die Verteilung der Personen im Computerlabor wurden die Effekte der Diffusion und Imitation adressiert. Versuchspersonen unterschiedlicher Gruppen konnten innerhalb einer Reihe keine benachbarten Arbeitsplätze belegen. Benachbarte Reihen von Arbeitsplätzen waren in entgegengesetzte Richtungen aufgestellt. Bildschirme waren damit für eine hintere Reihe schwerer einsehbar.

Die interne Validität wird auch durch eine mögliche Recherche des Quelltextes der Anwendung im Internet bedroht. Versuchspersonen wurden explizit angewiesen, nicht im Internet nach Optimierungsmaßnahmen für Cargo Tracker zu suchen. Die Aktivität der Versuchspersonen wurde permanent von dem Experimentator beaufsichtigt.

5.9.2.3 Konstruktvalidität

Die Konstruktvalidität bezieht sich auf die Übertragung der Ergebnisse auf die zugrundeliegenden Konstrukte.

Angemessene Beschreibung der Konstrukte

Die Unterstützung von Performancebewusstsein und das Antwortzeitverhalten von Komponenten bilden die grundlegenden Konstrukte dieses Experiments. Diese werden auf den Ansatz aus Kapitel 4 und auf das Vorhandensein von Performancebugs abgebildet. Eine detaillierte Beschreibung dieser Konzepte existiert in der Literatur und wurde als Grundlage für dieses Experiment verwendet. Die Operationalisierung des Antwortzeitverhaltens als Menge beseitigter Performancebugs repräsentiert eine Vereinfachung, denn Bugs können sich in ihrer Auswirkung stark unterscheiden. Die Auswirkung hängt im Kontext von Java-EE-Anwendungen von der Befüllung der Datenbank ab und kann nicht allgemeingültig beziffert werden. Eine vereinfachte Abbildung des Antwortzeitverhaltens ist dadurch notwendig.

Betrachtung eines einziges Objekts

Das Experiment basiert auf einer einzigen Beispielanwendung. Dadurch besteht das Risiko, dass das zugrundeliegende Java-EE-Technologie durch das Experiment nicht vollständig abgebildet wird. Die Anwendung besteht jedoch aus einer Vielzahl an Komponenten und deckt das breite Spektrum der Java-EE-Technologien ab. Nur zwei der in der Literatur identifizierten Performance Antipattern werden als Performancebugs instanziiert. Einerseits können nur Antipattern auf Implementierungsebene durch den Ansatz für Performancebewusstsein adressiert werden. Andererseits treffen andere Typen nicht auf die Anwendung zu. Insgesamt fünf Arten von Performancebugs werden instanziiert und auf sieben unterschiedliche Klassen verteilt.

Betrachung einer einzigen Methode

Die Bewertung der Optimierungsmaßnahmen durch eine einzige Person kann die Konstruktvalidität stark beeinflussen. Dadurch, dass Performancebugs synthetisch konstruiert und in die Anwendung eingebaut wurden, ist deren Existenz und Form ex ante bekannt. Durch den Vergleich der Änderungen am Quelltext mit dem Ausgangszustand der Anwendung und die Berücksichtigung von alternativen Lösungsansätzen wird der Einfluss der subjektiven Einschätzung während der Bewertung verringert. Durch die automatisierte Messung des Verhaltens der Versuchspersonen im Sinne der verbrachten Zeit, der besuchten und geänderten Dateien und der ausgeführten Commands werden die Ergebnisse gegengeprüft.

Ausprägung von Konstrukte

Die reine Betrachtung der Existenz von Konstrukte ohne Berücksichtigung ihrer Ausprägung kann auch die Konstruktvalidität beeinflussen (Wohlin et al. 2012, S. 109). Die zugrundeliegenden Konstrukte dieses Experiments werden jedoch auch hinsichtlich ihrer Ausprägung abgebildet. Die Qualifikation und der Wissensstand der Versuchspersonen werden anhand von vier Stufen gemessen. Das Ergebnis der Teilnehmer wird durch die genaue Bestimmung der Anzahl und des Typs der behobenen Performancebugs gemessen.

Interaktion mit dem Treatment

Versuchspersonen wurden genau einem Treatment zugeordnet. Dadurch kann eine Interaktion zwischen Treatments ausgeschlossen werden. Eine Interaktion zwischen dem Treatment und seiner Anwendung durch die Versuchspersonen ist gegeben. Bei einer solchen Interaktion ist die Versuchsperson noch empfänglicher für den Effekt des Treatments als sonst (Wohlin et al. 2012, S. 109). Als Ziel der Aufgabe wird die Optimierung des Antwortzeitverhaltens von Komponenten vorgegeben. In der Praxis würde ein Entwickler das Plugin jedoch auch mit dem Ziel einer Performanceevaluation aufrufen und derselbe Effekt wäre gegeben. Dadurch, dass beide Gruppen dieses explizite Ziel verfolgen ist der Effekt auf beiden Seiten gleich.

Eingeschränkte Übertragbarkeit auf andere Konstrukte

Die Anwendung des Treatments zielt auf die Verbesserung des Antwortzeitverhaltens von Komponenten ab. Bei der Optimierung der Antwortzeit können jedoch andere Aspekte der Performance, wie z.B. die Ressourcenauslastung oder der Durchsatz, verschlechtert werden. Bei der Zwischenspeicherung von Rückgabewerten wird die Wiederholung eines Aufrufs vermieden, dafür aber mehr Hauptspeicher verbraucht. Neben Aspekten der Performance können auch andere Qualitätsmerkmale, wie z.B. die Wartbarkeit von Quelltext, betroffen sein. Dadurch sind die Ergebnisse dieses Experiments eingeschränkt auf andere Konstrukte verallgemeinerbar.

Erraten von Hypothesen

Es besteht die Möglichkeit, dass Versuchspersonen die zugrundeliegende Hypothese des Experiments erraten und dadurch ihr Verhalten beeinflusst wird. Die Aufgabenstellung macht die Zielgröße dieses Experiments für die Teilnehmer transparent und schafft dadurch bei allen dieselbe Voraussetzung. Einzig die persönliche Einstellung einer Person führt zu einer positiven oder negativen Beeinflussung des Verhaltens (Wohlin et al. 2012, S. 110). Aufgrund der freiwilligen Teilnahme ist von einer relativ hohen Motivation auszugehen und dementsprechend von einer positiven Beeinflussung.

Einfluss der Evaluation

Personen deren Leistung gemessen wird, können dazu tendieren bessere Ergebnisse zu berichten, als tatsächlich erzielt wurden (Wohlin et al. 2012, S. 110). Durch eine automatisierte Erfassung und eine standardisierte Bewertung der Änderungen am Quelltext können Versuchspersonen ihre Ergebnisse nicht besser darstellen.

Erwartungen des Experimentators

Der Experimentator kann aufgrund seiner Erwartungen an das Experiment die Ergebnisse ungewollt beeinflussen. Das Risiko wird durch die Existenz eines einzigen Experimentators über alle Phasen dieses Experiments noch erhöht. Um eine Beeinflussung der Teilnehmer zu vermeiden, erhalten diese eine schriftliche Anleitung zur Bearbeitung der Aufgabe. Andere Einflüsse der Erwartungen des Experimentators können nicht ausgeschlossen werden.

5.9.2.4 Externe Validität

Die externe Validität adressiert Einflüsse auf die Übertragbarkeit der Ergebnisse auf die Praxis.

Einfluss der Selektion

Die Verallgemeinerbarkeit der Ergebnisse wird vor allem durch die Auswahl von Probanden, der Anwendung und der Aufgabe beeinflusst. Am Experiment nahmen Studenten, Wissenschaftler und ein Praktiker teil. Verschiedene Studien untersuchen die Eignung von Studenten als Subjekte für die Durchführung von Experimenten im Bereich der Softwaretechnik (Höst et al. 2000; Svahnberg et al. 2008; Runeson 2003). Höst et al. (2000) vergleichen die Aufwandschätzungen von Studenten und professionellen Softwareentwicklern und kommen zu der Schlussfolgerung, dass diese unter bestimmten Bedingungen sehr ähnliche Ergebnisse erzielen. Svahnberg et al. (2008) kommen zu dem Ergebnis, dass Studenten für Experimente im Bereich des Anforderungsmanagements geeignet sein können. Nach Kitchenham et al. (2002) eignen sich Studenten allgemein für die Evaluierung der Nutzung einer Technik durch Softwareingenieure mit weniger Erfahrung. Der Großteil der Versuchspersonen dieses Experiments verfügte jedoch schon über einen Bachelorabschluss und Programmiererfahrung in der Industrie. Über mehr als ein Jahr Industrieerfahrung haben etwa die Hälfte der Teilnehmer verfügt. Hinsichtlich der Erfahrungsstufe deckt die Stichprobe der Versuchspersonen das Spektrum der professionellen Softwareentwickler mit wenig Erfahrung bis zum Experten ab. Hinsichtlich der fachlichen Spezialisierung decken die Teilnehmer die Bereiche Informatik, Wirtschaftsinformatik und Games Engineering ab.

Einfluss der Umgebung

Die im Experiment verwendete Hardware ist für die Industrie nicht repräsentativ. Eine Antwortzeitvorhersage hat hier mehr als drei Mal länger gedauert als auf einem modernen Rechner. Die verwendete Entwicklungsumgebung repräsentiert jedoch den aktuell Stand der Technik. Eingesetzt wurden die neuesten Versionen von Java EE, Eclipse und Glassfish.

Einfluss der Anwendung

Der Einsatz der Anwendung Cargo Tracker als Objekt im Experiment stellt aufgrund der Technologie, der Komplexität und der Vertrautheit ein weiteres Risiko für die externe Validität dar. Cargo Tracker implementiert alle relevanten Java-EE-Technologien und ist in dieser Hinsicht repräsentativ für die Industrie. Diese weist jedoch eine geringe Komplexität auf. Insgesamt verfügt die Benutzeroberfläche über wenige Funktionen. Die Anwendung besteht aus relativ wenigen Komponenten, die eine einfache Logik implementieren. Die Datenbank weist nur einige wenige Strukturen auf. Es ist jedoch zu erwarten, dass der Nutzen von Performancebewusstsein bei einer komplexeren Anwendung noch höher ausfällt.

Versuchspersonen haben die Implementierung von Cargo Tracker vor dem Experiment nicht gekannt. Nur zwei Teilnehmer waren über die Existenz der Anwendung informiert. Es kann auch angenommen werden, dass keiner der Teilnehmer über Erfahrungen in der Domäne des Schiffsfrachttransports verfügte. In der Praxis sind Entwickler jedoch mit der Anwendung, die sie implementieren, vertraut. Diese kennen die Rolle und die Realisierung einzelner Komponenten und finden sich im Quelltext einfacher zurecht. Performancebugs können dadurch evtl. schneller identifiziert werden. Damit existieren nicht implizit Einsichten über das Antwortzeitverhalten von Diensten. Ziel des Ansatzes für Performancebewusstsein ist es Entwicklern von Java-EE-Anwendungen kontextspezifische Einsichten über die erwartete Antwortzeit von Komponenten automatisiert bereitzustellen. Antwortzeiten werden dabei auf Basis der wiederverwendeten Dienste berechnet. Im Kontext komplexer Abhängigkeiten und Kontrollflüsse sollen Entwickler auf potenzielle Performanceprobleme hingewiesen werden. Bei der Anwendung des Ansatzes in der Praxis wird auch davon ausgegangen, dass der Entwickler einer Komponente nur bedingt über die Implementierung der wiederverwendeten Dienste und deren Antwortzeitverhalten informiert ist. Die Bedrohung der externen Validität durch die Vertrautheit der Anwendung wird teilweise dadurch wiederlegt, dass einige Versuchspersonen der Test- und Kontrollgruppe bis über 40% bzw. 20% der Performancebugs beseitigen konnten.

Einfluss der Aufgabe

Die Aufgabe besteht in der Optimierung von existierenden Komponenten durch die Beseitigung von Performancebugs. Ähnliche Wartungsarbeiten können auch in der Praxis vorkommen. Die Ergebnisse des Experiments sind jedoch nicht unbedingt auf Entwicklungsarbeiten übertragbar. Diese Bedrohung der externen Validität wurde bewusst eingegangen, um in Gegenzug einen kurzen Ablauf mit einer möglichst hohen Teilnehmerzahl und niedriger Abbrecherquote zu ermöglichen. Im Gegensatz zu der Entwicklung einfacher Algorithmen, setzt die Implementierung von Java-EE-Komponenten mehr Wissen voraus. Überschaubare Entwicklungsaufgaben implizieren das Risiko vieler identischer Lösungen. Kompliziertere Aufgaben können zu mehr unvollständigen Lösungen führen.

Die ausgewählten Antipatterns und die eingebauten Performancebugs sind für die in der Praxis angetroffenen Performanceprobleme evtl. nicht repräsentativ. Andere Arten von Performancebugs, die z.B. aufgrund der Konfiguration bestimmter Komponenten auftreten, können jedoch durch den Ansatz nicht identifiziert werden. Auf der Abstraktionsebene der Implementierung können andere Antipattern nicht direkt auf die Anwendung übertragen werden.

Hinsichtlich der Interaktion zwischen Vergangenheit und Treatment wird das hier beschriebene Experiment von keinen besonderen Ereignissen beeinflusst.

5.9.3 Verallgemeinerbarkeit der Ergebnisse

In der Anwendung wurden fünf Arten von Performancebugs eingebaut. Jede einzelne Instanz davon wurde während dem Experiment bei Verfügbarkeit von Performancebewusstsein mindestens ein Mal identifiziert und beseitigt. Ein ähnlicher Effekt kann auch für andere Arten von Performancebugs erwartet werden, solange diese im Zusammenhang mit einem langsamen Methodenaufruf stehen. Ein solcher Aufruf weist eine Antwortzeit auf, die erstens in der Performancedatenbank hinterlegt ist und zweitens einen der Schwellenwerte übersteigt. Dieser Aufruf muss in der Menge der Anweisungen, welche den Performancebug aufweisen, enthalten sein. Methodenaufrufe, deren Antwortzeit den Schwellenwert überschreiten, werden im Quelltext hervorgehoben. Mit Hilfe von Markierungen wird dem Entwickler ein Pfad zu diesem Aufruf angezeigt. Performancebugs, die aus vielen schnelleren Methodenaufrufe bestehen, können bei der Berechnung der Antwortzeitvorhersage berücksichtigt werden. Der Entwickler kann dadurch auf die Existenz des Bugs aufmerksam gemacht werden. Im Quelltext würde die vom Bug betroffene Stelle nicht hervorgehoben werden. Die Ergebnisse können nicht auf Performancebugs, die sich auf die Ressourcennutzung oder den Durchsatz beziehen, verallgemeinert werden.

Im Experiment wurde sichergestellt, dass keine der Versuchspersonen Kenntnisse über die Implementierung von Cargo Tracker besitzt. Damit sollte verhindert werden, dass künstlich eingebaute Performancebugs aufgrund der Vertrautheit mit dem Quellcode identifiziert werden. Es ist jedoch zu erwarten, dass mit der Implementierung vertraute Entwickler auch von dem Ansatz für Performancebewusstsein profitieren. Die Auswertung der Ergebnisse hat gezeigt, dass je erfahrener die Versuchspersonen der Testgruppe waren, desto mehr Bugs sie beseitigen konnten. Es wird erwartet, dass dieser Effekt auch auf sehr erfahrene Entwickler zutrifft.

Cargo Tracker weist eine relativ einfache Implementierung auf. Komponenten verwenden oft wenige oder gar keine Dienste wieder. Für Implementierungen, die mehr Wiederverwendung und mehr Schichten aufweisen, werden bei der Nutzung des Ansatzes gleiche oder bessere Effekte erwartet.

5.9.4 Gewonnene Erkenntnisse

Die Anleitung für Teilnehmer hat die Funktionen und Oberfläche der Anwendung textuell und mit Hilfe von Screenshots beschrieben. Versuchspersonen hatten aber erst während der Bearbeitung der Aufgabe erstmalig Zugriff auf die Anwendung. Damit wurde ein Teil der Bearbeitungszeit für das Kennenlernen der Oberfläche genutzt. Bei der Gestaltung eines neuen

Experiments sollte eine separater Agendapunkt für die Interaktion mit der Benutzeroberfläche vorgesehen werden.

Die Fragebögen zur Erhebung der Kenntnisse der Teilnehmer wurden nach der Bearbeitung der Aufgabe ausgeteilt. Es ist vorstellbar, dass sich Versuchspersonen über ihre erzielten Ergebnisse bewusst waren und dies ihre Selbsteinschätzung beeinflusst hat. Beispielsweise könnte eine Versuchsperson, die keine Bugs gelöst hat, dazu verleitet werden, seine Kenntnisse vorsichtiger zu bewerten als sonst. Bei der Gestaltung eines neuen Experiments sollte das Ausfüllen der Fragebögen vor der Bearbeitung der Aufgabe eingeplant werden.

Das erneute Deployen der Anwendung hat bei einigen Versuchspersonen Fehler zurückgeliefert. Diese hatten ein zuvor ausgeführtes Skript nicht korrekt abgeschlossen. Dadurch lief der Anwendungsserver noch im Hintergrund und hat das erneute Deployen gestört. Obwohl das Skript selbst die korrekte Terminierung beschreibt, sollte ein entsprechender Hinweis auch in der Anleitung der Teilnehmer hinzugefügt werden.

Bei der Erhebung des Wissensstandes werden für die adressierten Bereiche teilweise unterschiedliche Einstufungen verwendet. Allgemeine Programmierkenntnisse, spezielle Kenntnisse und Kenntnisse über Cargo Tracker bieten jeweils vier Abstufungen an, jedoch mit unterschiedlichen Beschreibungen. Zur Herstellung der Vergleichbarkeit sollten für alle Bereiche dieselben Abstufungen verwendet werden.

Im Kommentarfeld des Fragebogens zur Erhebung der Benutzerfreundlichkeit haben Versuchspersonen der Testgruppe positives Feedback zum Plugin für Performancebewusstsein abgegeben. Als Erweiterung wurde die Bereitstellung einer hierarchischen Darstellung der Abhängigkeiten zu langsamen Diensten vorgeschlagen.

5.10 Zusammenfassung und Ausblick

Ziel dieses Experiments war es den Einfluss des Ansatzes für Performancebewusstsein zu untersuchen. Versuchspersonen sollten im Rahmen einer Aufgabe das Antwortzeitverhalten existierender Komponenten optimieren. Zu diesem Zweck wurden 13 Performancebugs in der Anwendung Cargo Tracker eingebaut. Die Effektivität der Optimierungsmaßnahmen wurde anhand der Anzahl der beseitigten Performancebugs gemessen. In Zuge des Experiments wurde die Hypothese, dass die Nutzung des Ansatzes für Performancebewusstsein keinen Einfluss auf die Anzahl der beseitigten Performancebugs hat, getestet. Die unabhängige Variable des Experiments ist die Verfügbarkeit des Ansatzes in der Entwicklungsumgebung der Versuchspersonen, woraus zwei Treatments resultieren. Rechner eines Computerlabors wurden mit Java EE IDEs ausgestattet. Für eine Teilmenge der IDEs war der Ansatz für Performancebewusstsein verfügbar. Bei den restlichen IDEs war das Plugin deaktiviert. Die abhängige Variable ist die Anzahl der behobenen Performancebugs. Studenten, Wissenschaftler und Praktiker wurden unter Anwendung von Convenience Sampling zur freiwilligen Teilnahme am Experiment eingeladen. Die adressierten Studenten belegen eines der Studiengänge Informatik, Wirtschaftsinformatik oder Games Engineering im Bachelor oder Master an der Technischen Universität München. Eingeladen wurden auch wissenschaftliche Mitarbeiter der Fakultät für Informatik der Technischen Universität München und der fortiss GmbH. Die adressierten Praktiker haben ein Studium an der Fakultät für Informatik der Technischen Universität München absolviert

und arbeiten in der Industrie als Softwareentwickler. Als Voraussetzung mussten diese mindestens grundlegende Programmierkenntnisse besitzen. Das Experiment wurde in dem Zeitraum zwischen den 17.03.2016 und dem 18.04.2016 über acht Iterationen durchgeführt. Insgesamt nahmen 26 Versuchspersonen, davon 21 Studenten, vier Wissenschaftler und ein Praktiker, am Experiment teil. Teilnehmer wurden über ein Losverfahren auf Experimentgruppen zugeordnet. Zur Kontrolle des Faktors der Erfahrung in der Softwareentwicklung wurde Blockbildung angewendet. Als Indikator für die Erfahrung wurde das Arbeitsverhältnis der Versuchspersonen angenommen. Nach der Bearbeitung der Aufgabe wurden Informationen über die Ausbildung und die Kenntnisse der Versuchspersonen mit Hilfe eines Fragebogens erhoben. Die Zufriedenheit bei der Nutzung des Plugins bzw. der IDE wurde mit Hilfe eines SUS-Fragebogens abgefragt.

Versuchspersonen mit Studentenstatus und Schulabschluss bildeten den größten Anteil beider Gruppen. Die meisten Versuchspersonen verfügen hinsichtlich der allgemeinen Programmierkenntnisse über mehr als ein Jahr Industrieerfahrung. Dabei weisen die Teilnehmer vor allem in den Bereichen Java und Eclipse Industrieerfahrung auf. Jeweils eine Versuchsperson der Test- und Kontrollgruppe gaben an, über die Anwendung Cargo Tracker informiert zu sein. Alle anderen hatten zuvor keine Kenntnisse darüber. Versuchspersonen der Testgruppe haben durchschnittlich über drei Mal mehr Performancebugs behoben als die Versuchspersonen der Kontrollgruppe. Jedes einzelne Performancebug wurde zumindest einmal von einem Teilnehmer der Testgruppe behoben. Innerhalb beider Gruppen schneideten Wissenschaftler und Praktiker durchschnittlich besser ab als Studenten. Mit steigender Erfahrung wurden in der Testgruppe immer bessere Ergebnisse erzielt. Dieser Effekt kann innerhalb der Kontrollgruppe nicht eindeutig festgestellt werden. Die durchgeführten Hypothesentests deuten darauf hin, dass sich die Verteilung der Anzahl der behobenen Performancebugs der Testgruppe signifikant von der Verteilung der Kontrollgruppe unterscheidet. Im Vergleich zur IDE im Allgemeinen wurde die Benutzerfreundlichkeit des Plugins für Performancebewusstsein durchschnittlich besser bewertet. Auch hier kann ein statistisch signifikanter Unterschied festgestellt werden. Versuchspersonen der Testgruppe haben durchschnittlich 10,6 % mehr Zeit innerhalb der IDE verbracht als Versuchspersonen der Kontrollgruppe. Beide Gruppen weisen eine relativ ähnliche Anzahl an besuchten Dateien und Quelltextänderungen auf. Es wurde eine signifikante lineare Korrelation zwischen der methodischen Untersuchung der relevanten Klassen und der Effektivität der Probanden der Testgruppe festgestellt. Hieraus wird abgeleitet, dass das Awareness Plugin eine methodische Untersuchung von Quellcode im Rahmen von Wartungsarbeiten unterstützt und somit die Effektivität der Entwickler erhöht.

Zukünftige Forschung sollte den Einfluss des Ansatzes für Performancebewusstsein in neuen Szenarien untersuchen. Das vorliegende Experiment hat nur die Verfügbarkeit des Ansatzes betrachtet. Erstrebenswert wäre auch die Gegenüberstellung der Ergebnisse unter Anwendung verschiedener Werkzeuge. Hierbei kommen sowohl andere Ansätze für Performancebewusstsein als auch klassische Werkzeuge für Performanceevaluationen in Frage. Beispielsweise könnten Versuchspersonen aufgefordert werden, einen Java Profiler für die Optimierung von Komponenten einzusetzen. Darüber hinaus sollte der Ansatz für Performancebewusstsein auch im Rahmen einer Entwicklungsaufgabe evaluiert werden.

Entwickler mussten im hier beschriebenen Experiment Antwortzeitvorhersagen explizit anfordern. Aufgrund der Dauer, die eine Vorhersage in Anspruch genommen hat, konnte der Ansatz

keine implizite Vorhersage, die z.B. automatisch nach jeder Speicheroperation angestoßen wird, unterstützen. Der Ansatz sollte für dieses Szenario optimiert werden. Im Rahmen eines weiteren Experiments könnte dann untersucht werden, ob der Einfluss auf Entwickler genauso hoch bleibt, wenn Vorhersagen unaufgefordert angezeigt werden.

6 Performance Model Management Repository

In diesem Kapitel wird das Konzept und die Umsetzung eines Performance Model Management Repository (PMMR) vorgestellt. Als erstes wird die Relevanz der adressierten Fragestellungen motiviert (siehe Abschnitt 6.1). Das PMMR wird im Abschnitt 6.2 konzeptuell beschrieben. Die technische Realisierung des PMMR wird im Abschnitt 6.3 vorgestellt. Abschnitt 6.3.4 bietet einen Überblick über die Integration des PMMR mit dem Ansatz für Performancebewusstsein aus Kapitel 4. Eine Evaluation des PMMR wird in Abschnitt 6.4 beschrieben. Abschließend werden die Ergebnisse zusammengefasst und weitere Forschungsrichtungen aufgezeigt (siehe Abschnitt 6.5).

Eine erste Vision des Performance Model Management Repository wurde in (Brunnert et al. 2015a) veröffentlicht. Die in diesem Kapitel beschriebene Umsetzung des PMMR auf Basis von PCM wurde in (Danciu et al. 2015a) veröffentlicht.

6.1 Motivation

6.1.1 Problemstellung

Performancemodelle können in der Praxis abhängig von dem Entwicklungsstadium der abgebildeten Komponenten bzw. Anwendungen sowohl manuell erstellt als auch automatisch generiert werden. Die Erstellung, Verwaltung und Bearbeitung dieser Modelle wird aufgrund von komplexen Architekturen, Lebenszyklen und Governancestrukturen von Unternehmensanwendungen (Brunnert et al. 2014) zunehmend aufwändiger.

Bei erhöhter Komplexität von Architekturen nehmen die Anzahl der Komponenten und deren Beziehungen zu. Für die Untersuchung des Verhaltens von Anwendungen müssen somit zunehmend mehr Aspekte modelliert werden. Zur effizienten Abbildung von wiederverwendeten Implementierungen müssen auch Modellelemente besser wiederverwendbar sein.

Einzelne Komponenten dieser komplexen Architekturen unterliegen einer ständigen Evolution, von der Erhebung von Anforderungen, über den Entwurf und die Implementierung, bis hin zum produktiven Betrieb. Dabei können sich die Entwicklungsstände individueller Komponenten voneinander unterscheiden. Beispielsweise kann der Entwurf einer Komponente schon die Wiederverwendung einer anderen, schon produktiven Komponente vorsehen. Andererseits können individuelle Komponenten auch unterschiedliche Releasezyklen aufweisen. Einzelne Komponenten können beispielsweise gar nicht, in langen Abständen oder permanent weiterentwickelt werden. Als Ergebnis der losgelösten Entwicklungszyklen und der komplexen Abhängigkeiten kann der parallele Betrieb unterschiedlicher Versionsstände derselben Komponente notwendig werden. Dabei können die Implementierungen zweier Versionen stark voneinander abweichen. Schnittstellen unterliegen einer ähnlichen Entwicklung und können auch in unterschiedlichen Versionen angetroffen werden. Während eine Schnittstelle konstant bleibt, kann sich die Implementierung der entsprechenden Komponenten verändern. Ein und dieselbe Komponente kann gleichzeitig unterschiedliche Versionen einer Schnittstelle unterstützen. Performancemodelle müssen daher die Existenz mehrerer Versionen derselben Komponente bzw. Schnittstelle unterstützen.

Aus organisatorischer Perspektive können Rechte und Pflichten hinsichtlich der Entwicklung und der Verwaltung von Komponenten unterschiedlichen Entitäten zugeordnet sein. Komponenten können beispielsweise durch unterschiedliche Teams weiterentwickelt werden. Der Zugang zu Informationen über die Implementierung und den Entwicklungsstand von Komponenten kann dadurch erschwert werden. Werkzeuge für die Modellierung der Performance müssen in diesem Kontext den effizienten Austausch von Modellinhalten unterstützen und dabei gleichzeitig Zugriffsrechte prüfen. Bei der Zusammenarbeit unterschiedlicher Entitäten müssen Artefakte parallel bearbeitet werden können. Performancemodelle müssen daher einen Mehrfachzugriff unterstützen und Werkzeuge für die Auflösung von Konflikten anbieten.

Die Modellierung der Performance von Komponenten kann auch die Spezifikation von benötigten Ressourcen, wie z.B. CPU, Festplatte oder Netzwerk, abbilden. Neben der Art wird auch der Umfang der benötigten Ressourcen spezifiziert. Bedarfe können sowohl unabhängig von der Ausprägung der Ressource, beispielsweise in Form von CPU-Zyklen, als auch spezifisch für einen bestimmten Hardwaretyp beschrieben werden. Im Kontext komplexer Architekturen, Lebenszyklen und Governancestrukturen müssen Performancemodelle verschiedene Ressourcentypen abbilden können. Einerseits müssen Performancemessungen aus unterschiedlichen Hardwareumgebungen in Komponentenspezifikationen integriert werden können. Andererseits muss das Verhalten von Komponenten für unterschiedliche Hardwareumgebungen evaluiert werden können.

Bei der automatisierten Generierung von Performancemodellen, wie beispielsweise durch den Ansatz für Performancebewusstsein, werden Unternehmensanwendungen bzw. Komponenten relativ detailliert abgebildet. Einige Ansätze sehen auch eine relativ häufige Generierung von Modellen vor (Brunnert/Krcmar 2015). Dabei wird einerseits Rechenleistung in Anspruch genommen und andererseits Zeit aufgewendet. Für eine effizientere Generierung von Performancemodellen sollten diese durch Werkzeuge unternehmensweit zwischengespeichert und wiederverwendet werden können.

6.1.2 Forschungsziele

Für die effiziente Verwaltung und Bearbeitung von Performancemodellen in komplexen Unternehmensumgebungen soll ein PMMR entworfen werden. Ziel des PMMR ist die Verwaltung und Versionierung von Spezifikation für Komponenten und Schnittstellen. Durch die Integration mit dem PMMR soll der Funktionsumfang des Ansatzes für Performancebewusstsein erweitert werden. Die Entwicklung des PMMR soll folgende Aspekte adressieren:

- Versionierung: Es soll untersucht werden, wie unterschiedliche Versionen von Komponenten und Schnittstellen abgebildet und im PMMR verwaltet werden können.
- Management von Modellelementen: Es soll untersucht werden, wie Spezifikationen von Komponenten und Schnittstellen gespeichert, ausgetauscht und gleichzeitig bearbeitet werden können. Zusätzlich sollen die Aspekte der Strukturierung der Inhalte und des Zugriffsschutzes adressiert werden.
- Übertragung von Änderungen: Spezifikationen von Komponenten und Schnittstellen sollen aus dem PMMR ausgelesen und für die Evaluierung beliebiger Szenarien in andere Performancemodelle integriert werden können. Es soll untersucht werden, wie Änderungen an

den Spezifikationen innerhalb des PMMR auf andere Performancemodelle, welche diese wiederverwenden, übertragen werden können.

- Management von Ressourcenbedarfe: Es soll untersucht werden, wie aus unterschiedlichen Hardwareumgebungen abgeleitete Ressourcenbedarfe durch das PMMR verwaltet werden können.
- Integration mit dem Ansatz für Performancebewusstsein: Es soll untersucht werden, wie die Erstellung der Performancemodelle durch den Ansatz für Performancebewusstsein mit Hilfe des PMMR unterstützt werden kann.

Das Konzept des PMMR soll anschließend technisch realisiert und in eine Modellierungsumgebung integriert werden.

6.1.3 Kontext

Das Konzept des PMMR fokussiert die Verwaltung spezifischer Elemente von Performancemodellen für komponentenbasierte Anwendungen. Genauer soll die Spezifikation von Komponenten und Schnittstellen verwaltet werden. Andere Elemente, wie z.B. die Spezifikation des Workloads oder der Instanziierung von Komponenten werden von dem hier beschriebenen PMMR nicht berücksichtigt.

Das PMMR soll sowohl die Interaktion mit Nutzern als auch mit anderen Werkzeugen unterstützen. Nutzer sollen mit Hilfe einer grafischen Oberfläche die Funktionen des PMMR einsetzen. Über eine API sollen Funktionen programmatisch gesteuert werden können. Zur Unterstützung der Wiederverwendung von Spezifikationen in Performancemodellen wird das PMMR mit einem Modellierungswerkzeug integriert.

Als Meta-Modell für das PMMR dient das PCM. Als Modellierungswerkzeug wird daher die Palladio-Bench eingesetzt. Wichtige Aspekte des PMMR werden jedoch nicht durch PCM unterstützt. Durch die Speicherung von Performancemodellen als Datei wird die parallele Bearbeitung erschwert. Spezifikationen von Komponenten oder Schnittstellen können nicht über mehrere Modelle wiederverwendet werden. Die Versionierung von Komponenten und Schnittstellen wird durch PCM nicht unterstützt. Ressourcenbedarfe werden innerhalb der Komponentenspezifikation dokumentiert. Einige Ansätze leiten diese Bedarfe von hardwarespezifischen Performancemessungen, wie z.B. Antwortzeitmessungen, ab (Brunnert et al. 2013). Dadurch können die resultierenden Modelle nur für bestimmte Hardwareumgebungen eingesetzt werden. PCM wird für die Unterstützung der Funktionen eines PMMR erweitert.

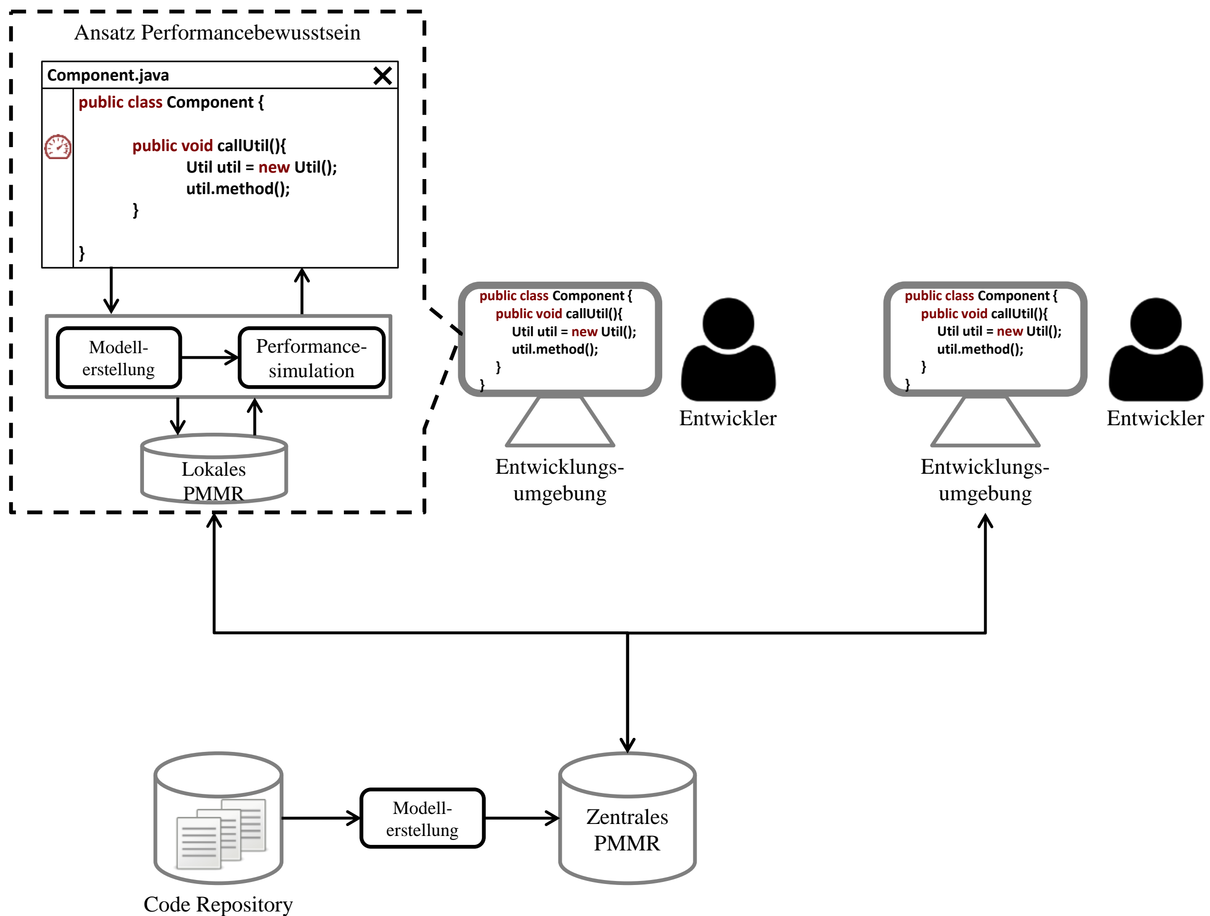


Abbildung 6-1: Ansatz für Performancebewusstsein als Anwendungsfall des PMMR
Quelle: Eigene Darstellung

Der Ansatz für Performancebewusstsein aus Kapitel 4 dient als Anwendungsfall für den Einsatz des PMMR (siehe Abbildung 6-1). Für die Berechnung einer Antwortzeitvorhersage wird der Quelltext einer Menge von Komponenten eingelesen und als Performancemodell abgebildet. Die Anzahl der modellierten Komponenten hängt von dem Umfang des entsprechenden Java-Projektes und den verfügbaren Antwortzeitmessungen ab (vgl. auch Abschnitt 4.2.2). Der aktuelle Stand des Ansatzes sieht für jede Vorhersage die Erstellung eines vollständig neuen Modells vor, unabhängig von den vorangegangenen Änderungen am Quelltext. Zwischen zwei Antwortzeitvorhersagen kann es jedoch zu relativ wenigen Änderungen kommen. Beispielsweise können wiederholte Änderungen an der fokussierten Komponente vorgenommen werden. In diesem Fall würden auch alle restlichen Komponenten erneut verarbeitet werden. Insgesamt nehmen die einzelnen Antwortzeitvorhersagen unnötig viel Zeit in Anspruch. Zur Optimierung der Antwortzeitvorhersage können Komponentenspezifikationen mit Hilfe des PMMR zwischengespeichert werden. Bei der Modellerstellung kann geprüft werden, ob eine bestimmte Version einer Java-Klasse schon als Repository-Komponente abgebildet wurde. In diesem Fall könnte die Komponentenspezifikation im aktuellen Performancemodell wiederverwendet werden. Ansonsten müsste zuerst eine neue Spezifikation erstellt und im PMMR abgelegt werden. Modellelemente sollen während der Erstellung der PCM-Instanz nicht von einer entfernten Ablage abgerufen, sondern lokal zwischengespeichert werden. Hierfür wird zwischen einem lokalen und einem zentralen PMMR unterschieden. Unterschiedliche Entwickler benötigen für die Durchführung von Antwortzeitvorhersagen evtl. dieselben

Komponentenspezifikationen. Durch die Synchronisation des lokalen und zentralen PMMR können Modellelemente ausgetauscht werden. Die Erstellung der Komponentenspezifikationen kann sowohl dezentral auf den Entwicklerrechnern als auch zentral für den vollständigen Quelltext einer Anwendung durchgeführt werden.

6.2 Konzept für das Management von Performancemodellen

Ein Überblick über das Konzept des PMMR ist in Abbildung 6-2 dargestellt. Spezifikationen von Komponenten und Schnittstellen werden auf einer zentralen Ablage abgespeichert. Inhalte des PMMR können durch unterschiedliche Teams bzw. Nutzer erstellt, bearbeitet und ausgelesen werden. Zur Abbildung des Lebenszyklus von Komponenten und Schnittstellen können die entsprechenden Spezifikationen mit Versionsinformationen versehen werden. Inhalte werden durch Nutzer lokal zwischengespeichert und Änderungen auf die zentrale Ablage festgeschrieben.

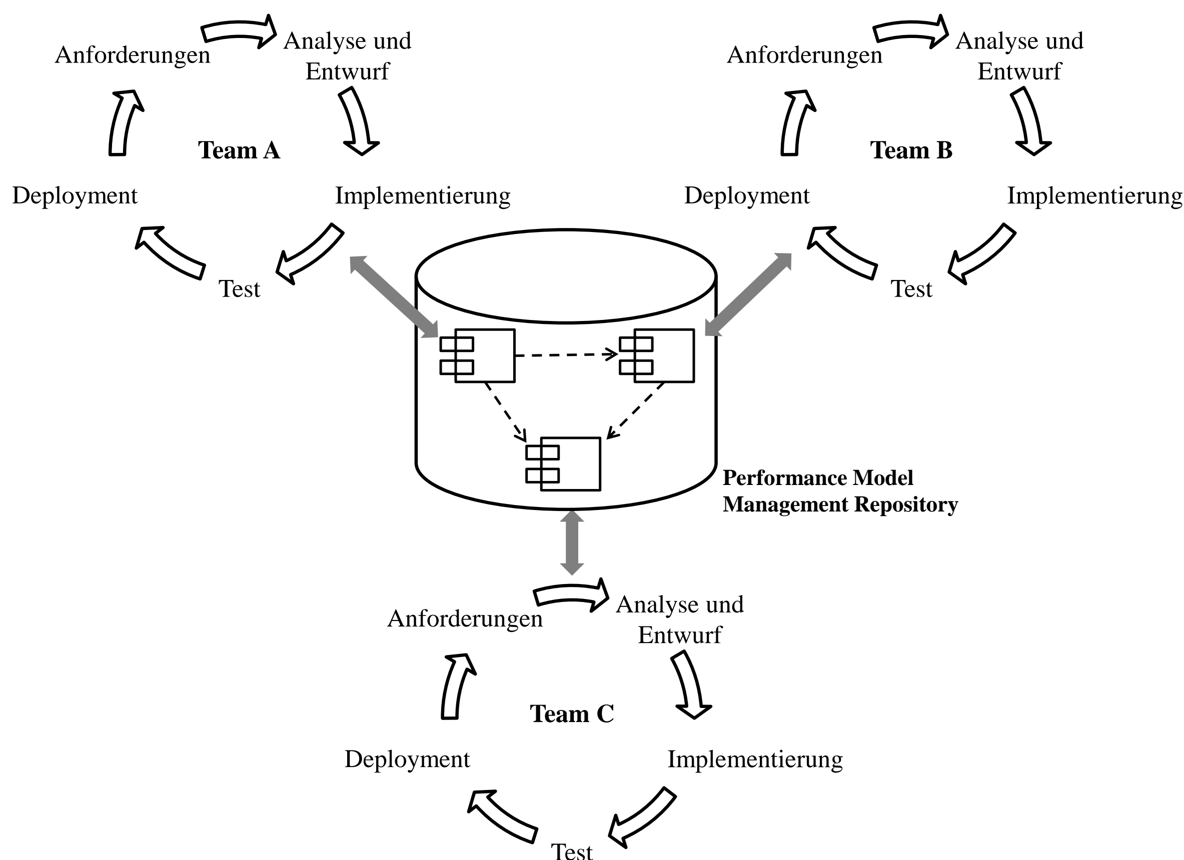


Abbildung 6-2: ***Konzeptuelle Darstellung eines Performance Model Management Repository***
Quelle: In Anlehnung an Brunnert et al. (2015a)

Nutzer des PMMR können Inhalte zur Durchführung von Performanceevaluationen in lokale Performancemodelle integrieren. Änderungen am Quellelement im PMMR werden auf die betroffenen Performancemodelle übertragen. Die wichtigsten Aspekte des PMMR-Konzepts werden im Folgenden beschrieben.

6.2.1 Versionierung von Komponenten und Schnittstellen

Die Kontrolle der Evolution komplexer Systeme steht im Fokus des Software Configuration Management (SCM) (Tichy 1988). Im SCM wird der Begriff der Versionierung auf unterschiedliche Abstraktionsebenen verwendet. IEEE (1988) unterscheidet zwischen *Version* und *Revision*. Während hier mit einer Version die Weiterentwicklung der Funktionalität gekennzeichnet wird, bündelt eine Revision nur beseitigte Bugs bei gleichbleibender Funktionalität. Bei der Abbildung der Performance von Komponenten ist diese Unterscheidung jedoch nicht sinnvoll. Das Verhalten einer Komponente kann einerseits durch die Beseitigung eines Bugs stark beeinflusst werden. Die Erweiterung der Funktionalität muss andererseits nicht unbedingt eine Änderung der Performance herbeiführen.

Nach Conradi/Westfechtel (1997) und Conradi/Westfechtel (1998) können Versionen in folgende Kategorien unterteilt werden:

- Revisionen: Bezeichnen die Weiterentwicklung vorheriger Versionen und können sowohl die Beseitigung von Bugs als auch Erweiterungen der Funktionalität beinhalten.
- Varianten: Bezeichnen alternative Versionen eines Artefakts, welche parallel zum Einsatz kommen.

Versionskontrollsysteme, wie beispielsweise Subversion[29] oder EMFStore[30], unterstützen die Verwaltung von Änderungen an Quelltext oder Modellen (Tichy 1984; Kögel 2011). Änderungen führen hier implizit zu einer neuen Version, die nach der Definition von Conradi/Westfechtel (1997) einer Revision entspricht. Im Rahmen eines Release Managements können hingegen explizite Versionen spezifiziert werden (Stuckenholz 2005). Diese entsprechen den Varianten nach Conradi/Westfechtel (1997).

Während der Entwicklung von Diensten können folgende Dimensionen der Versionierung unterschieden werden (Andrikopoulos et al. 2012):

- Versionierung von Schnittstellen: Unterstützung für die Verwaltung der Evolution von Artefakten, welche die Interaktion eines Dienstes mit seiner Umwelt beschreiben.
- Versionierung von Implementierungen: Unterstützung für die Verwaltung der Evolution des Quelltextes, der Ressourcen, der Konfiguration und der Dokumentation von Diensten.

Mit Hilfe des PMMR sollen Versionen von Komponenten und Schnittstellen verwaltet werden. Daher muss sowohl die Versionierung von Implementierungen als auch von Schnittstellen unterstützt werden. Für die Verwaltung von parallel verwendeten Versionsstände von Komponenten und Schnittstellen unterstützt das PMMR die Spezifikation von expliziten Versionen. Diese werden im Folgenden Versionsspezifikation genannt und entsprechen der Definition für Varianten nach Conradi/Westfechtel (1997). Versionsspezifikationen sind entweder Komponentenversionen oder Schnittstellenversionen. Änderungen an einer Versionsspezifikation werden über implizite Versionen verwaltet. Implizite Versionen werden im Folgenden Revision

[29] https://subversion.apache.org
[30] http://www.eclipse.org/emfstore/

genannt und entsprechen der Definition nach Conradi/Westfechtel (1997). Ein Beispiel für die Spezifikation von Versionen von Komponenten und Schnittstellen im PMMR ist in Abbildung 6-3 dargestellt.

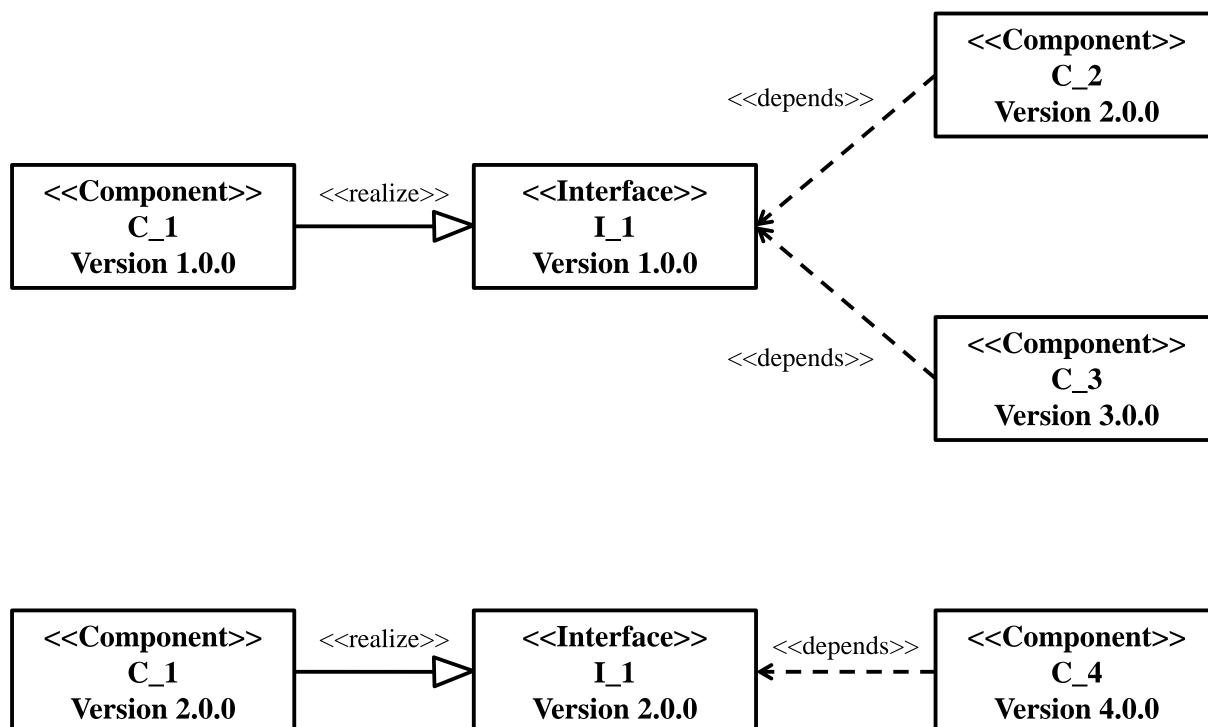


Abbildung 6-3:* *Beispiel für die Versionierung von Komponenten und Schnittstellen
Quelle: In Anlehnung an Danciu et al. (2015a)

Alle Komponenten und Schnittstellen, die im PMMR verwaltet werden, besitzen eine Versionsnummer. Durch die Weiterentwicklung, Verschmelzung oder Aufteilung von Versionsspezifikationen kann die Ausprägung von Komponenten und Schnittstellen verändert werden. Das Konzept der einzigartigen Identität von Komponenten und Schnittstellen wird abgelöst und durch die Existenz verschiedener paralleler Versionen ersetzt. Schnittstellenversionen werden von Komponentenversionen angeboten oder benötigt.

6.2.2 Verwaltung von Versionsspezifikationen

Bei der Verwaltung von PMMR-Inhalten werden drei Werkzeuge unterschieden (siehe Abbildung 6-4). Versionsspezifikationen werden auf einem zentralen Server persistent gespeichert. Das *zentrale PMMR* spiegelt den aktuellen Zustand der Daten wieder. Versionsspezifikationen sind in *Projekte* organisiert. Nutzer können die Inhalte des zentralen PMMR nicht direkt bearbeiten. Zum Erstellen, ändern oder auslesen von Versionsspezifikationen müssen Nutzer ein bestimmtes Projekt auf ein *lokales PMMR* duplizieren (engl. check-out). Eine Benutzerverwaltung unterstützt die Spezifikation von Lese- und Schreibrechte der Nutzer für einzelne Projekte. Auf dem lokalen PMMR haben Nutzer die Möglichkeit neue Versionsspezifikationen anzulegen oder existierende zu bearbeiten. Änderungen werden in Form einer Revision auf das zentrale PMMR festgeschrieben (engl. check-in). Über eine Aktualisierung des lokalen PMMR werden die aktuellsten Revisionen vom zentralen Server abgerufen. Die Abläufe der Interaktion

zwischen dem lokalen und dem zentralen PMMR orientieren sich an die Gestaltung von Versionskontrollsystemen (Tichy 1984).

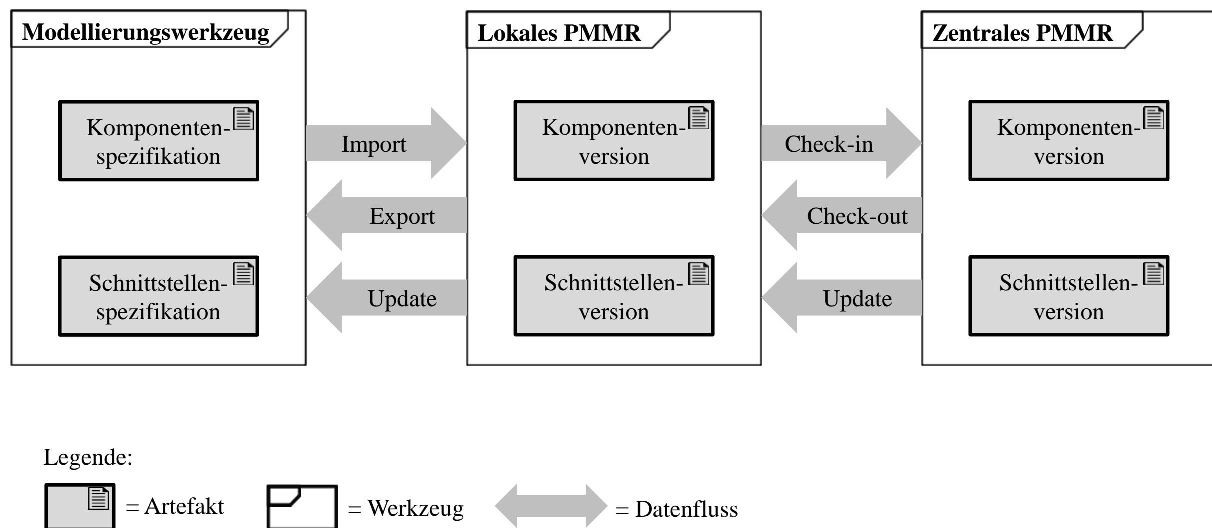


Abbildung 6-4: ***Verwaltung von Komponenten und Schnittstellen***
Quelle: Eigene Darstellung

Nutzer haben die Möglichkeit Inhalte des PMMR mit einem Modellierungswerkzeug auszutauschen. Das Modellierungswerkzeug wird für die Erstellung und Bearbeitung von Performancemodellen eingesetzt. Diese Modelle spezifizieren Komponenten und Schnittstellen ohne Versionsinformation. Über eine Importfunktion können Nutzer Spezifikationen von Komponenten und Schnittstellen aus einem Performancemodell in das PMMR übertragen und als Versionsspezifikation abbilden. In diesem Zuge kann eine beliebige Versionsinformation zugeordnet werden. Andersherum können Versionsspezifikationen auf beliebige Performancemodelle übertragen werden. Die entsprechenden Versionsinformationen gehen dabei verloren. Änderungen an Versionsspezifikationen, die vom zentralen PMMR abgerufen werden, können auf die exportierten Komponenten und Schnittstellenspezifikationen übertragen werden (siehe Abschnitt 6.2.3).

Schnittstellenspezifikationen beschreiben wie Komponenten untereinander interagieren können. Ohne die Spezifikation von Schnittstellen können Komponenten nicht in Performancemodellen verwendet werden. Daher werden beim Import von Komponentenspezifikationen aus einem Performancemodell automatisch auch alle angebotenen und benötigten Schnittstellenspezifikationen in das PMMR übernommen. Dasselbe gilt auch für den Export von Komponentenspezifikationen.

Schnittstellen können von mehreren Komponenten benötigt oder angeboten werden. Beim Einsatz einer Schnittstelle in einem Performancemodell sind jedoch nicht alle diese Komponenten zwingend notwendig. Beim Austausch von Schnittstellen zwischen dem lokalen PMMR und einem Modellierungswerkzeug werden die verbundenen Komponenten nicht automatisch übernommen. Diese müssen explizit importiert bzw. exportiert werden.

6.2.3 Übertragung von Änderungen an Versionsspezifikationen

Beliebige Mengen von Versionsspezifikationen aus dem PMMR können zu einer beliebigen Anzahl von Performancemodellen exportiert werden. Änderungen an den Versionsspezifikationen, die nach dem Export stattfinden, werden auf die betroffenen Performancemodelle automatisch übertragen. Während dem Export wird zu jeder Versionsspezifikation eine Zuordnung auf das entsprechende Performancemodell gespeichert. Die Zuordnung spezifiziert folgende Elemente:

- Quellobjekt: Beschreibt die eindeutige Referenz auf eine Versionsspezifikation, die aus dem PMMR exportiert wurde.
- Zielobjekt: Beschreibt die eindeutige Referenz auf eine Schnittstellen- oder Komponentenspezifikation, die im Performancemodell auf der Grundlage einer Versionsspezifikation aus dem PMMR erstellt wurde.
- Performancemodell: Beschreibt eine eindeutige Referenz auf ein Performancemodell, zu dem eine Versionsspezifikation exportiert wurde.
- Versionsstand: Beschreibt welche Revision einer Versionsspezifikation die entsprechende Schnittstellen- oder Komponentenspezifikation abbildet.

Änderungen an Versionsspezifikationen werden nach jeder Aktualisierung eines Projekts im lokalen PMMR auf die entsprechenden Performancemodelle übertragen. Dafür iteriert das PMMR über alle Zuordnungen und gruppiert diese anhand des Performancemodells. Für jedes Performancemodell wird die Menge der Quell- und Zielobjekte abgeleitet. Für jedes Zielobjekt aus dieser Menge werden die Unterschiede zum Quellobjekt berechnet. Alle identifizierten Unterschiede werden anschließend auf das Zielobjekt übertragen.

Änderungen an den Schnittstellen- und Komponentenspezifikationen im Performancemodell werden nicht auf die Versionsspezifikationen im PMMR übertragen.

6.2.4 Verwaltung von Ressourcenbedarfe

Komponenten können Ressourcenbedarfe, die in einer bestimmten Hardwareumgebung gemessen wurden, spezifizieren. Bedarfe beziehen sich dabei auf bestimmte Ressourcentypen, wie CPU, Festplatte oder Netzwerk. Beispielsweise kann der CPU-Bedarf einer Komponente als Zeitdauer, die auf einem bestimmten CPU-Typ gemessen wurde, angegeben sein. Bei der Ausführung der Komponenten auf einem anderen Hardwaretyp kann die Komponente ein unterschiedliches Verhalten aufweisen. Durch die Verwaltung von Komponentenspezifikationen im PMMR sollen diese in verschiedenen Performancemodellen für unterschiedliche Evaluierungsszenarien wiederverwendbar sein. Diese Performancemodelle können andere Hardwareumgebungen adressieren und sind dadurch nicht mit den spezifizierten Ressourcenbedarfen kompatibel.

Für die Unterstützung von heterogenen Hardwareumgebungen sieht das PMMR die Umrechnung von Ressourcenbedarfen vor. Alle hardwarespezifischen Ressourcenbedarfe werden im PMMR relativ zu einem Hardware-Benchmark-Ergebnis abgespeichert (Menascé/Almeida 2002, S. 270). Beim Auslesen der Komponentenspezifikation wird der Ressourcenbedarf für

eine Zielumgebung umgerechnet. Der Ablauf der Umrechnung der Ressourcenbedarfe ist in Abbildung 6-5 dargestellt. In einem ersten Schritt erstellt ein Komponentenentwickler eine Komponentenspezifikation. Der Ressourcenbedarf wird auf dem *Hardwaretyp A* gemessen und spezifiziert die Auslastung verschiedener Ressourcentypen durch die Komponente. Das PMMR spezifiziert für jeden Ressourcentyp einen Referenz-Benchmark-Ergebnis. Dabei handelt es sich um das Ergebnis eines standardisierten Benchmarks auf einer Referenz-Hardware. Für CPUs kann beispielsweise der Benchmark *SPEC CPU2006*[31] eingesetzt werden. Beim Check-in der Komponentenspezifikation gibt der Komponentenentwickler das Benchmark-Ergebnis des Hardwaretyps A an und der Ressourcenbedarf wird für die Referenz-Hardware umgerechnet. Zu einem späteren Zeitpunkt führt ein Komponentenentwickler ein Check-out der Komponente durch. Diese soll für eine Performanceevaluation für den *Hardwaretyp B* eingesetzt werden. Dabei gibt der Entwickler das Benchmark-Ergebnis dieses Hardwaretyps an und der Ressourcenbedarf wird für die Zielumgebung umgerechnet.

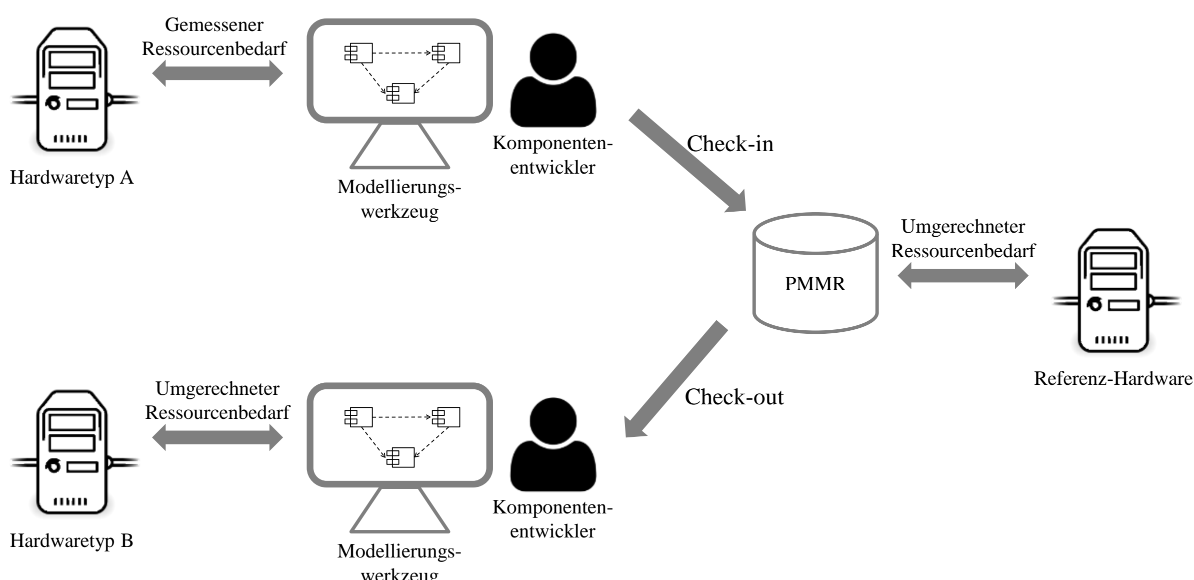


Abbildung 6-5: Umrechnung der Ressourcenbedarfe während dem Check-in und Check-out
Quelle: Eigene Darstellung

Im PMMR wird der Ressourcenbedarf $r_{baseline}$ einer Komponente gespeichert. Das PMMR verfügt über das Benchmark-Ergebnis $b_{baseline}$ für eine Referenz-Hardware. Der Ressourcenbedarf $r_{baseline}$ wird während dem Check-in wie folgt berechnet (Brunnert et al. 2015a):

$$r_{baseline} = \frac{b_{baseline}}{b_{checkinbenchmarkvalue}} * r_{checkinvalue}$$

Der gemessene Ressourcenbedarf wird als $r_{checkinvalue}$ übergeben. Das Benchmark-Ergebnis des Hardwaretyps, auf dem die Messung durchgeführt wurde, wird als $b_{checkinbenchmarkvalue}$ übergeben.

[31] http://www.spec.org/cpu2006/

Beim Exportieren einer Komponentenspezifikation aus dem PMMR wird der Ressourcenbedarf $r_{baseline}$ für eine bestimmte Zielumgebung wie folgt als $r_{checkoutvalue}$ berechnet (Brunnert et al. 2015a):

$$r_{checkoutvalue} = \frac{b_{checkoutbenchmarkvalue}}{b_{baseline}} * r_{baseline}$$

Das Benchmark-Ergebnis des Hardwaretyps der Zielumgebung wird als $b_{checkoutbenchmarkvalue}$ übergeben.

Das PMMR sieht die Umrechnung der Bedarfe für beliebige Arten von Ressourcen vor. Für hardwareunabhängige Ressourcenbedarfe wird für die Benchmark-Ergebnisse $b_{chekinbenchmarkvalue}$ und $b_{checkoutbenchmarkvalue}$ jeweils das Referenzergebnis $b_{baseline}$ verwendet.

6.3 Technische Umsetzung

Die Implementierung des PMMR wird in eine existierende Modellierungsumgebung eingebettet. Die technologische Plattform sowie das zugrundeliegende Meta-Modell werden im Folgenden beschrieben.

6.3.1 Grundlegende Technologien

Für die technische Umsetzung des PMMR werden existierende Technologien wiederverwendet und teilweise erweitert. Ein Überblick über die technologische Plattform des PMMR ist in Abbildung 6-6 dargestellt. Das PMMR verwendet PCM in der Version 3.4.1 als Meta-Modell. Die Modellierungsumgebung für PCM-Instanzen wird durch die Palladio-Bench implementiert. Diese basiert auf der Eclipse-Plattform und wird um das PMMR erweitert.

PCM unterstützt keine Versionierung von Komponenten- und Schnittstellenspezifikationen. Im Repository-Modell können Ressourcenbedarfe unabhängig von der Hardware spezifiziert werden. Eine Umrechnung von hardwarespezifischen Ressourcenbedarfe wird hingegen nicht unterstützt. Für diese Zwecke wird PCM erweitert (siehe Abschnitt 6.3.2). Die Spezifikation des PCM basiert auf EMF (vgl. auch Abschnitt 2.7). Erweiterungen an PCM werden daher im zugrundeliegenden EMF-Modell eingeführt.

PCM-Instanzen werden durch die Palladio-Bench als Datei abgespeichert. Der Austausch und die kollaborative Bearbeitung von Performancemodellen werden dadurch erschwert. Konventionelle Versionskontrollsysteme, wie beispielsweise Subversion, unterstützen die parallele Bearbeitung graphbasierter Modelle nur bedingt (Kögel 2011, S. 6). Für die Verwaltung von Versionsspezifikationen setzt das PMMR aus diesem Grund EMFStore[32] ein. EMFStore implementiert ein Versionskontrollsystem für EMF-basierte Modelle und besteht aus dem *Workspace* und einem *Repository*.

[32] http://www.eclipse.org/emfstore/

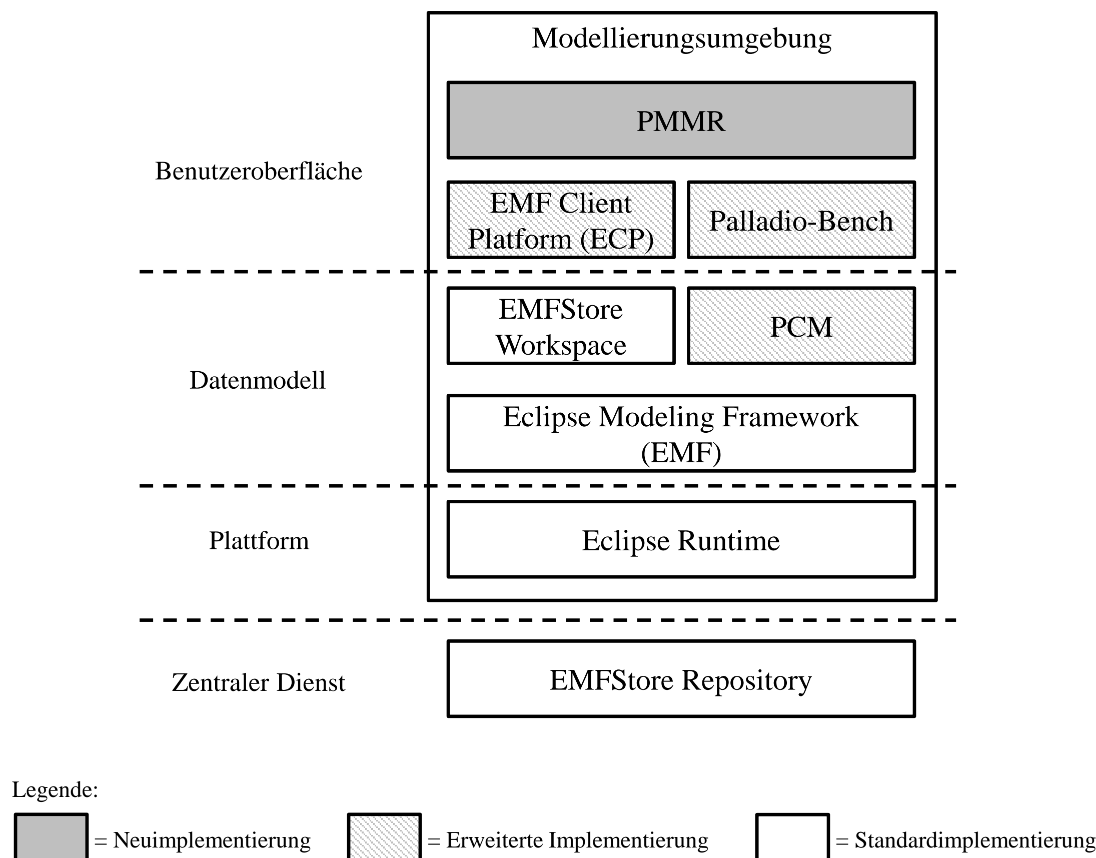


Abbildung 6-6: Technologische Plattform des Performance Model Management Repository
Quelle: Eigene Darstellung

Modellinhalte werden dezentral im Workspace erzeugt und bearbeitet. Eine Menge an lokalen Änderungen kann auf dem Repository festgeschrieben (engl. commit) werden und führt zu einer neuen Revision des Modells. Während dem Commit wird das lokale Workspace aktualisiert. Konflikte zwischen den lokalen Änderungen und der aktuellsten Revision vom Repository werden dem Nutzer gemeldet. Dieser hat die Möglichkeit Maßnahmen für die Beseitigung der Konflikte einzuleiten. Nach dem Festschreiben von lokalen Änderungen auf dem Repository stehen diese auch für andere Nutzer zur Verfügung.

Eine Benutzeroberfläche für die Interaktion mit EMFStore wird durch die EMF Client Platform (ECP)[33] bereitgestellt. Das PMMR verwendet ECP unter anderem für die Herstellung der Verbindung zum Repository sowie die Verwaltung und Bearbeitung von Modellinhalten.

6.3.2 Erweiterungen des Palladio Component Model

PCM wird über eine Ecore-Instanz, die in Pakete organisiert ist, spezifiziert (Becker 2008). Für die Umsetzung des PMMR notwendige Erweiterungen an PCM werden in dem neu hinzugefügten Paket *pmmr* gebündelt (siehe Abbildung 6-7). Dieses verwendet die Pakete *repository* und *resourcetype* wieder.

[33] http://www.eclipse.org/ecp/

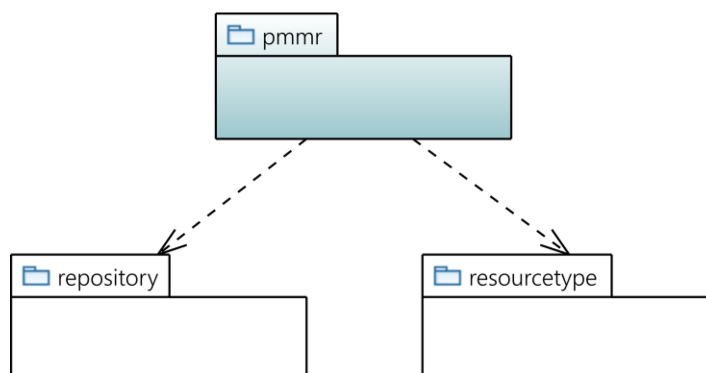


Abbildung 6-7: Erweiterung der Ecore-Instanz der PCM-Spezifikation
Quelle: Eigene Darstellung

An den existierenden Ecore-Paketen und deren Inhalte werden keine Änderungen vorgenommen. Vorhandene PCM-Instanzen und Editoren der Palladio-Bench müssen aufgrund der Erweiterung nicht angepasst werden und bleiben nutzbar.

6.3.2.1 Versionierung von Komponenten- und Schnittstellenspezifikationen

Die Erweiterungen an PCM für die Unterstützung der Versionierung von Komponenten- und Schnittstellenspezifikationen sind in Abbildung 6-8 dargestellt. Alle Inhalte eines PMMR-Projekts sind dem Wurzelknoten *Application* untergeordnet und in *Packages* organisiert. Packages repräsentieren Verzeichnisse und sind hierarchisch aufgebaut. Auf der obersten Ebene besteht eine Application aus genau einem *Root Package*. Ein Package kann Unterverzeichnisse und eine Menge von inhaltlich zusammenhängenden Versionsspezifikationen beinhalten. Nutzer können Packages beliebig anlegen, benennen und ineinander verschachteln. Paketstrukturen können beispielsweise den Aufbau einer Anwendung oder der Unternehmensorganisation abbilden. Versionsspezifikationen können nach verschiedenen Kriterien, wie z.B. nach Release, Entwicklungsstand, oder Organisationszugehörigkeit, in Packages gruppiert werden.

Die abstrakte Klasse *VersionSpec* definiert die Attribute und Beziehungen einer Versionsspezifikation. Eine VersionSpec verfügt über eine Versionsnummer, ein Versionsdatum, eine Beschreibung und drei Arten von Beziehungen zur Abbildung der Evolution einer Versionsspezifikation. Über eine Vorgänger-/Nachfolgerbeziehung wird die Weiterentwicklung einer Komponente oder Schnittstelle mit ihrer Vorgängerversion verknüpft. Die Aufteilung einer Komponente oder Schnittstelle in mehrere getrennte Spezifikationen wird über die *Branch-Beziehung* abgebildet. Die Zusammensetzung einer neuen Spezifikation aus mehreren Vorgängerversionen kann über die *Merge-Beziehung* abgebildet werden. *ComponentVersion* und *InterfaceVersion* implementieren die Klasse VersionSpec und bilden Versionen von Komponenten- bzw. Schnittstellenspezifikationen ab. Durch die Spezifikation von generischen Typen können die Beziehungen der VersionSpec nur zwischen Objekten derselben Klasse, z.B. zwei Objekte vom Typ ComponentVersion, existieren.

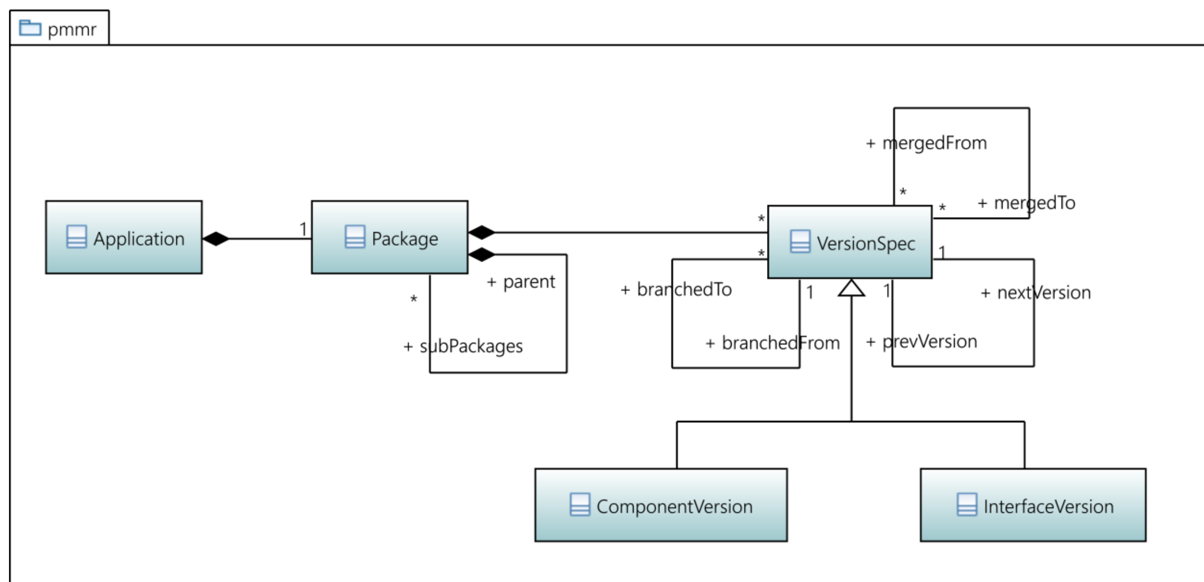


Abbildung 6-8: ***Erweiterung des PCM für die Versionierung von Komponenten und Schnittstellen***
Quelle: Danciu et al. (2015a)

6.3.2.2 Verwaltung von Komponenten- und Schnittstellenspezifikationen

Der Entwurf eines Datenmodells für das PMMR zur Verwaltung von Komponenten- und Schnittstellenspezifikationen wird durch die von PCM, der Palladio-Bench und EMFStore auferlegten technischen Einschränkungen maßgeblich beeinflusst. Das Datenmodell des Repository-Diagramms wird im gleichnamigen Ecore-Paket spezifiziert und besteht aus genau einem gleichnamigen Wurzelknoten. Komponenten- und Schnittstellenspezifikationen werden in PCM mit Hilfe der Klassen *BasicComponent* bzw. *OperationInterface* abgebildet. Durch die Erweiterung der Superklassen *ImplementationComponentType* und *RepositoryComponent* sowie *Interface* stehen diese implizit in einer Containment-Beziehung mit dem Repository-Element. Ziel des PMMR ist es Elemente vom Typ RepositoryComponent und OperationInterface um Versionsinformationen zu erweitern und innerhalb eines eigenen Datenmodells zu verwalten. Die Definition einer neuen Datenstruktur für die Spezifikation von Komponenten und Schnittstellen würde erhebliche Nachteile implizieren. Diese könnten in PCM-Instanzen nicht direkt wiederverwendet werden. Beim Import und Export von Spezifikationen müssten Inhalte zwischen den Datenstrukturen transformiert werden.

In EMF dürfen Klassen höchstens an einer Containment-Beziehung teilnehmen. Mit Hilfe von EMF generierte Standardeditoren setzten für die Erstellung und Änderung von Elementen eine Containment-Beziehung voraus. Bei der Bearbeitung eines Wurzelknotens unterstützten diese beispielsweise das Hinzufügen eines Unterknotens nur wenn dieser in einer Containment-Beziehung mit dem Wurzelknoten steht. Als Ergebnis muss das PMMR auch die Containment-Beziehung und das Repository-Element wiederverwenden. Die entsprechende Erweiterung der PCM-Spezifikation ist in Abbildung 6-9[34] dargestellt.

[34] Die Version der Abbildung in (Danciu et al. 2015a) hat zur Vereinfachung nur die implzite Beziehung zwischen Repository und BasicComponent bzw. OperationInterface dargestellt.

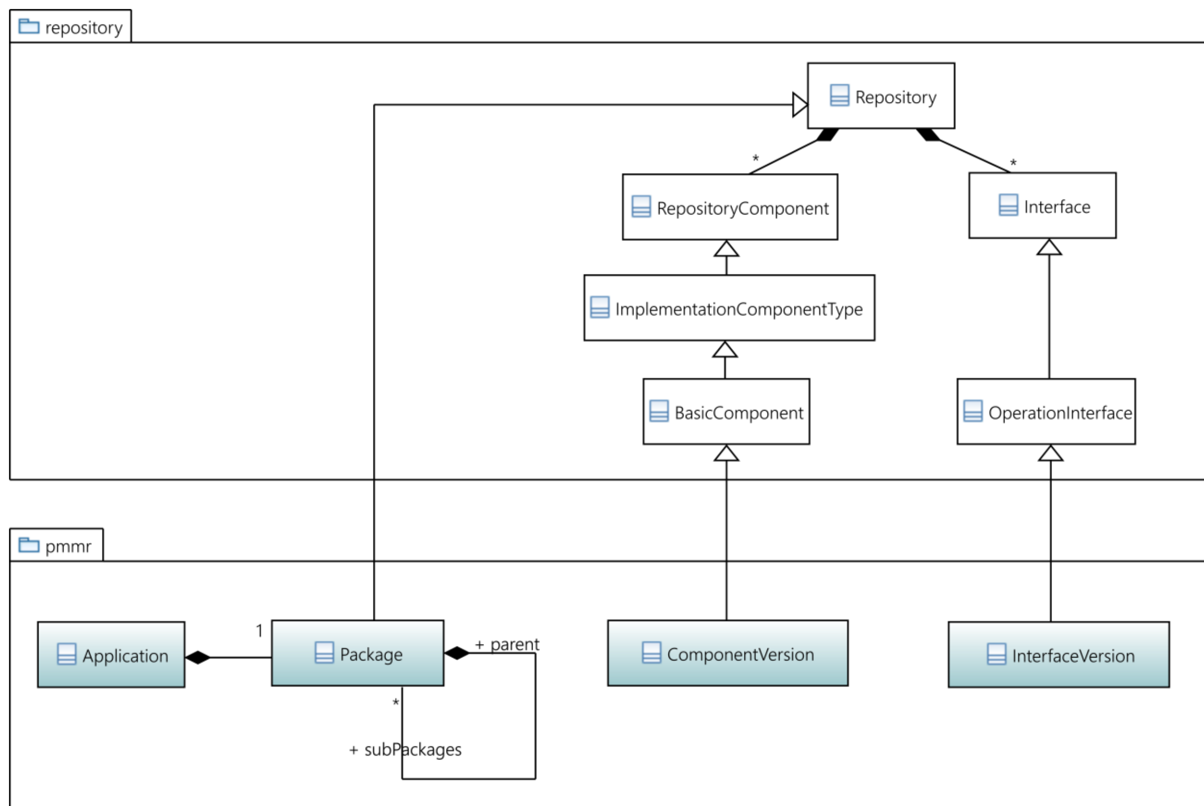


Abbildung 6-9: Erweiterung des PCM für die Verwaltung von Komponenten und Schnittstellen
Quelle: In Anlehnung an Danciu et al. (2015a)

Die Klassen ComponentVersion und InterfaceVersion erweitern BasicComponent bzw. OperationInterface. Damit erhalten Komponenten- und Schnittstellenspezifikationen die Eigenschaften und Beziehungen einer Versionsspezifikation. Da VersionSpec-Elemente zusammengesetzt bzw. aufgeteilt werden können wird durch diese Erweiterung die eindeutige Identität der BasicComponent- und OperationInterface-Elemente aufgelöst. Im resultierenden Datenmodell kann beispielsweise dieselbe BasicComponent innerhalb zweier Versionen vorkommen.

Die Palladio-Bench sieht die Speicherung von Spezifikationen innerhalb eines großen Repository-Diagramms vor. Bei der Aufteilung von Komponenten und Schnittstellen auf mehrere Repositories können diese nicht mehr einander referenzieren. Durch die Erweiterung des Repository durch die Klasse Package unterstützt das PMMR die Verwaltung von Spezifikationen innerhalb einer hierarchischen Struktur. VersionSpec-Elemente unterschiedlicher Pakete können dabei einander referenzieren.

6.3.2.3 Übertragung von Änderungen an Versionsspezifikationen

Für die Übertragung von Änderungen an exportierten VersionSpec-Instanzen kommen unterschiedliche Implementierungen in Frage. Einerseits kann eine Instanz von beliebig vielen EMF-basierten Modellen referenziert werden. Hierbei existieren keine Kopien der Instanz und Änderungen müssen nicht explizit übertragen werden. Aufgrund der Containment-Beziehung können Instanzen von BasicComponent nur höchstens einem Repository zugeordnet werden (vgl. auch Abschnitt 6.3.2.2). VersionSpec-Instanzen aus dem PMMR stehen schon in der

Containment-Beziehung zu Package und können daher nicht in einem anderen Performancemodell hinzugefügt werden. Sobald dies passiert, wird die Beziehung der VersionSpec-Instanz zum PMMR-Paket mit der neuen Beziehung zu einer Repository-Instanz ersetzt.

Eine andere Alternative ist die Spezifikation einer Proxy-Klasse für die Abbildung von exportierten VersionSpec-Instanzen. Der Proxy muss die Klasse BasicComponent erweitern und kann dadurch einem Repository hinzugefügt werden. Anstatt eine Kopie der Attribute und Beziehungen der VersionSpec anzubieten, könnte der Proxy die aktuellen Werte bei Bedarf von der VersionSpec abrufen. Hierfür benötigt der Proxy eine einfache Beziehung zu der VersionSpec-Instanz. Bei der Erweiterung der BasicComponent erbt der Proxy die Containment-Beziehung zu der Klasse *ServiceEffectSpecification*. Anstatt eigene Unterelemente zu spezifizieren muss der Proxy jedoch die ServiceEffectSpecification-Instanzen der VersionSpec abrufen. Als Ergebnis kann der Proxy durch EMF nicht serialisiert werden. Ein anderer Nachteil dieses Vorgehens ist, dass alle Methoden der Proxy-Klasse, die für den Abruf sowie das Setzen von Werten und Referenzen zuständig sind, manuell überschrieben werden müssen. Bei der erneuten Generierung des Quelltextes aus dem Ecore-Modell würden diese Änderungen verloren gehen.

Für die Übertragung von Änderungen an VersionSpec-Instanzen, werden Kopien angelegt und mit Hilfe eines Synchronisationsmechanismus auf dem aktuellen Stand gehalten. Die entsprechende Erweiterung der PCM-Spezifikation ist in Abbildung 6-10 dargestellt.

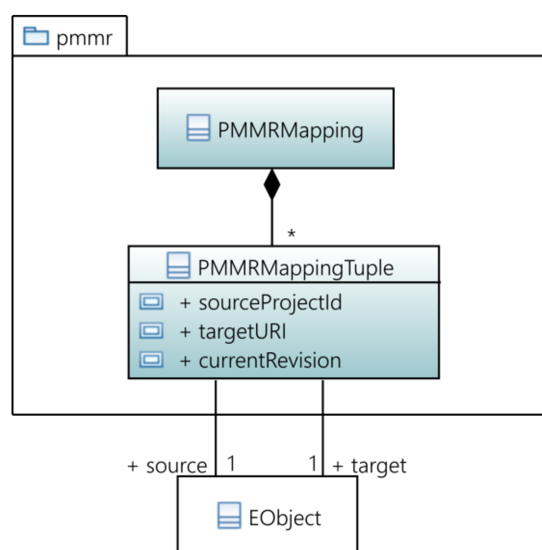


Abbildung 6-10: Erweiterung des PCM für die Übertragung von Änderungen an Versionsspezifikationen
Quelle: Eigene Darstellung

Ein lokales PMMR verwaltet mit Hilfe der Klasse *PMMRMappingTuple* die Zuordnung von VersionSpec-Instanzen auf exportierte Kopien. Das Quell- und Zielobjekt wird jeweils mit der generischen Ecore-Klasse EObject referenziert. Zielobjekte können BasicComponent- oder OperationInterface-Instanzen sein. Die Zuordnung spezifiziert in welchem Projekt die VersionSpec-Instanz angesiedelt ist und welche Revision dieser Instanz das Zielobjekt aktuell abbildet. Über das Attribut *targetURI* wird die Ressource, in der das Zielobjekt angesiedelt ist,

referenziert. Beim Export einer VersionSpec legt das lokale PMMR jeweils einen Eintrag für jedes Zielobjekt an. Die Klasse *PMMRMapping* dient als Wurzelknoten für alle Zuordnungen.

Nach der Aktualisierung eines Projekts im lokalen PMMR werden alle betroffenen Zuordnungen identifiziert und anhand der targetURI gruppiert. Mit Hilfe von EMF Compare[35] wird die Menge der exportierten VersionSpec-Instanzen mit der Menge der Zielobjekte verglichen. Für jedes zugeordnete Paar werden dabei die Werte der Attribute und die Ausprägung der Beziehungen gegenübergestellt. Alle im Zielobjekt abweichenden Eigenschaften werden mit den aktuellen Werten der VersionSpec-Instanz überschrieben. Abschließend wird der Wert des Attributs currentRevision in der entsprechenden PMMRMappingTuple-Instanz aktualisiert.

6.3.2.4 Verwaltung von Ressourcenbedarfe

Ressourcenbedarfe einer Komponentenspezifikation werden auf der Ebene von *InternalAction* spezifiziert. Dabei wird eine Auslastung in Form eines Werts mit einer *ProcessingResourceType* verknüpft. Die ProcessingResourceType beschreibt einen Typ von Ressource bzw. eine Hardwarekomponente, wie beispielsweise die CPU. Die Palladio-Bench gibt eine feste Menge von Ressourcentypen, die über Ihren Namen eindeutig unterschieden werden, vor. Dazu gehören die Typen *CPU*, *HDD* und *DELAY*. Während der Simulation wird anhand des Namens des Ressourcentyps jeweils ein unterschiedliches Verhalten abgebildet. VersionSpec-Instanzen sollten diese Ressourcentypen wiederverwenden. Ansonsten besteht die Möglichkeit, dass im PMMR abweichende Ressourcentypen spezifiziert werden, die während der Simulation nicht verarbeitet werden können. Um dem wiederum entgegenzuwirken müsste beim Export und Import von Spezifikationen eine Transformation durchgeführt werden.

Die Erweiterungen an PCM für die Unterstützung von heterogenen Ressourcenbedarfe sind in Abbildung 6-11 dargestellt. Die von der Palladio-Bench vordefinierten Ressourcentypen werden automatisch in jedem PMMR-Projekt importiert. Innerhalb des PMMR können je Ressourcentyp beliebig viele konkrete Ressourcen, eines bestimmten Typs und Herstellers, als *ResourceInstance* spezifiziert werden. Damit kann beispielsweise eine konkrete Variante einer CPU eines bestimmten Herstellers abgebildet werden. ResourceInstance-Objekte können nur vom Typ der vordefinierten ProcessingResourceTypes spezifiziert werden und sind in einem *BenchmarkRepository* eingegliedert.

[35] https://www.eclipse.org/emf/compare/

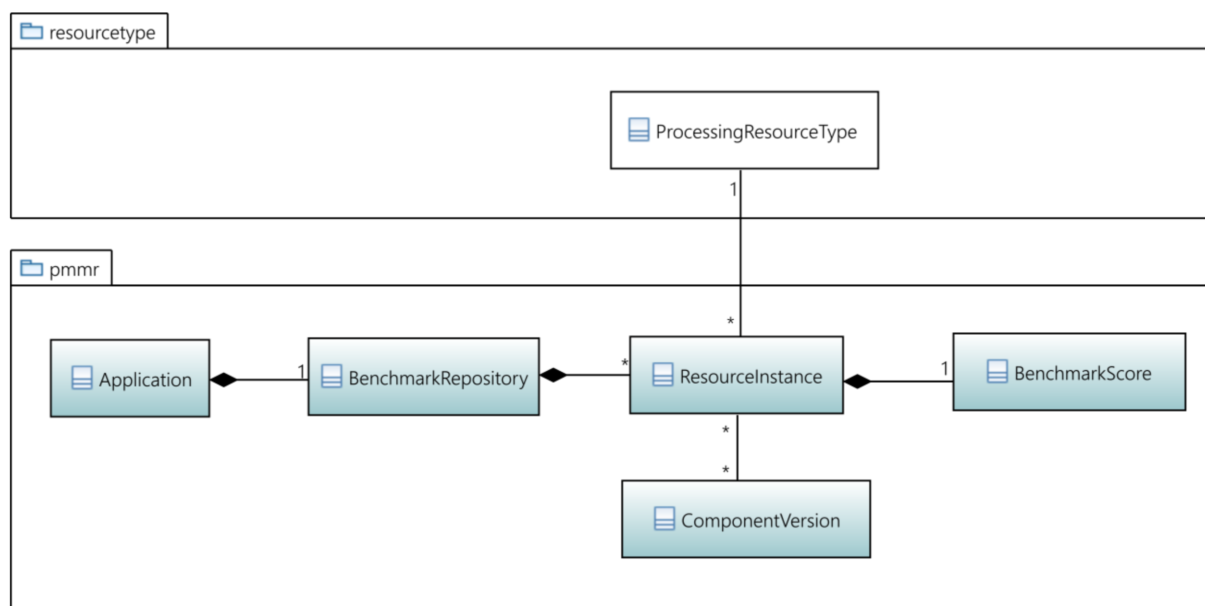


Abbildung 6-11: Erweiterung des PCM für die Verwaltung von Ressourcenbedarfe
Quelle: Danciu et al. (2015a)

Eine ResouceInstance verfügt über genau einem *BenchmarkScore*, der die Leistung von Komponenten während der Bearbeitung einer bestimmten Aufgabe spezifiziert. Das PMMR sieht vor, dass zur Erhebung der BenchmarkScores verschiedener ResourceInstances desselben Typs der gleiche Benchmark verwendet wird.

Bei der Erstellung oder dem Import einer Komponentenspezifikation muss der Nutzer für jeden Ressourcentyp jeweils eine Ressourceninstanz angeben. Damit wird die Hardwareumgebung, in der entsprechende Ressourcenbedarfe gemessen wurden, spezifiziert. Beim Export einer Komponentenspezifikation muss der Nutzer für jeden Ressourcentyp jeweils eine Zielhardware auswählen und die Ressourcenbedarfe werden umgerechnet.

6.3.3 Verwaltung und Bearbeitung von Versionsspezifikationen

Das zentrale PMMR wird mit Hilfe des EMFStore-Repository umgesetzt. Hierbei handelt es sich um eine Serverkomponente, welche für die Speicherung und Versionierung von EMF-basierten Modellen sowie den Zugriffsschutz zuständig ist (Kögel 2011, S. 34). Modellinhalte werden in Projekte strukturiert. Für den Zugriff auf Projekte können Benutzerkonten angelegt und konfiguriert werden. Das Zugriffsrecht für ein Projekt kann einzelnen Benutzer oder Benutzergruppen vergeben werden. Das lokale PMMR wird mit Hilfe des EMFStore-Workspace umgesetzt. Projekte werden lokal im Workspace dupliziert und können dort bearbeitet werden. Das Workspace unterstützt die Nachverfolgung von Änderungen sowie die Erkennung und Behebung von Änderungskonflikten (Kögel 2011, S. 35).

ECP bietet für die Interaktion mit EMFStore verschiedene Benutzeroberflächen. Mit dem *Repository Explorer* können die Verbindung zu einem EMFStore-Repository hergestellt, Benutzerkonten verwaltet und Projekte dupliziert werden. Der *Model Explorer* unterstützt die Navigation durch Projekte und deren Inhalte. EMF-Objekte werden anhand von Contains-Beziehungen in einer hierarchischen Baumstruktur dargestellt. Der *Editor* implementiert eine Oberfläche

für die Bearbeitung von einzelnen EMF-Objekten. Attribute und Beziehungen eines Objekts werden als Felder eines generischen Formulars dargestellt. Für die Unterstützung einer grafischen Modellierung von RDSEFFs wurde ECP um einen neuen Editor erweitert.

Versionsspezifikationen können im PMMR mit Hilfe des Model Explorer erstellt oder aus einer existierenden PCM-Instanz importiert werden. Der Import und Export von Versionsspezifikationen werden im Folgenden beschrieben.

6.3.3.1 Import von Versionsspezifikationen

Nutzer haben die Möglichkeit existierende Komponenten- und Schnittstellenspezifikationen aus PCM-Instanzen zu importieren (siehe Abbildung 6-12). Der Vorgang kann im Editor für Repository-Modelle nach der Selektion einer beliebigen Menge BasicComponent- bzw. OperationInterface-Elemente initiiert werden. Über eine grafische Oberfläche spezifiziert der Nutzer das Zielpaket innerhalb eines PMMR-Projekts und ggf. die Hardwareumgebung, in der Ressourcenbedarfe gemessen wurden. Für hardwareunabhängige Ressourcenbedarfe kann die Spezifikation einer Hardwareumgebung übersprungen werden. Für alle selektierten Komponentenspezifikationen werden die angebotenen und benötigten Schnittstellenspezifikationen ermittelt und in die Menge der Elemente, die importiert werden sollen, aufgenommen. Ein- und Ausgabeparameter von Schnittstellen können mit Hilfe vordefinierter oder selbst erstellter Elemente des Typs *DataType* spezifiziert werden. Für alle Schnittstellenspezifikationen werden die referenzierten DataType-Objekte ermittelt und in die Menge der Elemente, die importiert werden sollen, aufgenommen.

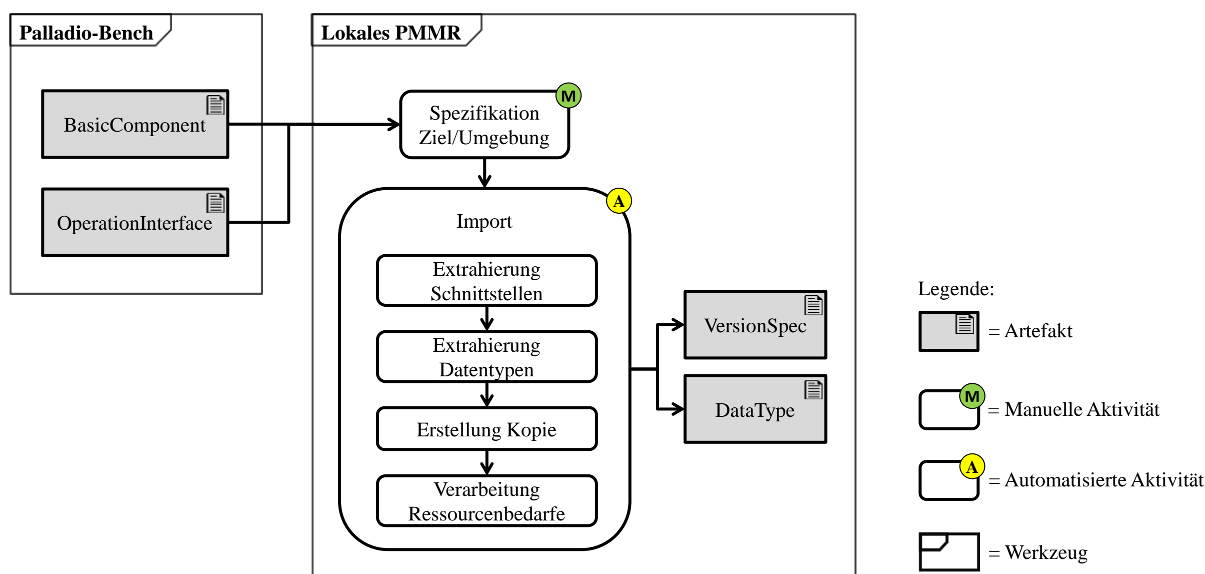


Abbildung 6-12: Import von Versionsspezifikationen aus existierenden PCM-Instanzen
Quelle: Eigene Darstellung

Von der resultierenden Menge der Komponenten-/Schnittstellenspezifikationen und Datentypen wird eine Kopie erstellt. Dabei werden Objekte vom Typ BasicComponent als ComponentVersion und Objekte vom Typ OperationInterface als InterfaceVersion instanziiert. Für die Spezifikation der Versionsinformationen werden Standardwerte verwendet. Das PMMR iteriert

über alle ComponentVersion-Instanzen und verarbeitet die Spezifikation der Ressourcenbedarfe. Referenzen auf Ressourcentypen aus der Quelldatei werden mit Verweisen auf entsprechende PMMR-Objekte ausgetauscht. ComponentVersion-Instanzen werden anhand der spezifizierten Hardwareumgebung mit entsprechenden ResourceInstance-Objekten verknüpft. Abschließend werden die Inhalte dem spezifizierten Zielpaket hinzugefügt.

6.3.3.2 Export von Versionsspezifikationen

Nutzer haben die Möglichkeit Versionsspezifikationen aus dem PMMR nach einer PCM-Instanz zu exportieren (siehe Abbildung 6-13). Der Vorgang kann im Model Explorer nach der Selektion einer beliebigen Menge VersionSpec-Elemente initiiert werden. Bevor Versionsspezifikationen exportiert werden können, muss der Nutzer alle lokalen Änderungen auf das zentrale PMMR festschreiben. Über eine grafische Oberfläche spezifiziert der Nutzer als Ziel ein Repository-Modell und eine Hardwareumgebung. Für alle selektierten ComponentVersion-Instanzen werden die angebotenen und benötigten Schnittstellenspezifikationen ermittelt und in die Menge der Elemente, die exportiert werden sollen, aufgenommen. In diese Menge werden anschließend auch die von InterfaceOperation-Instanzen referenzierten DataType-Objekte hinzugefügt.

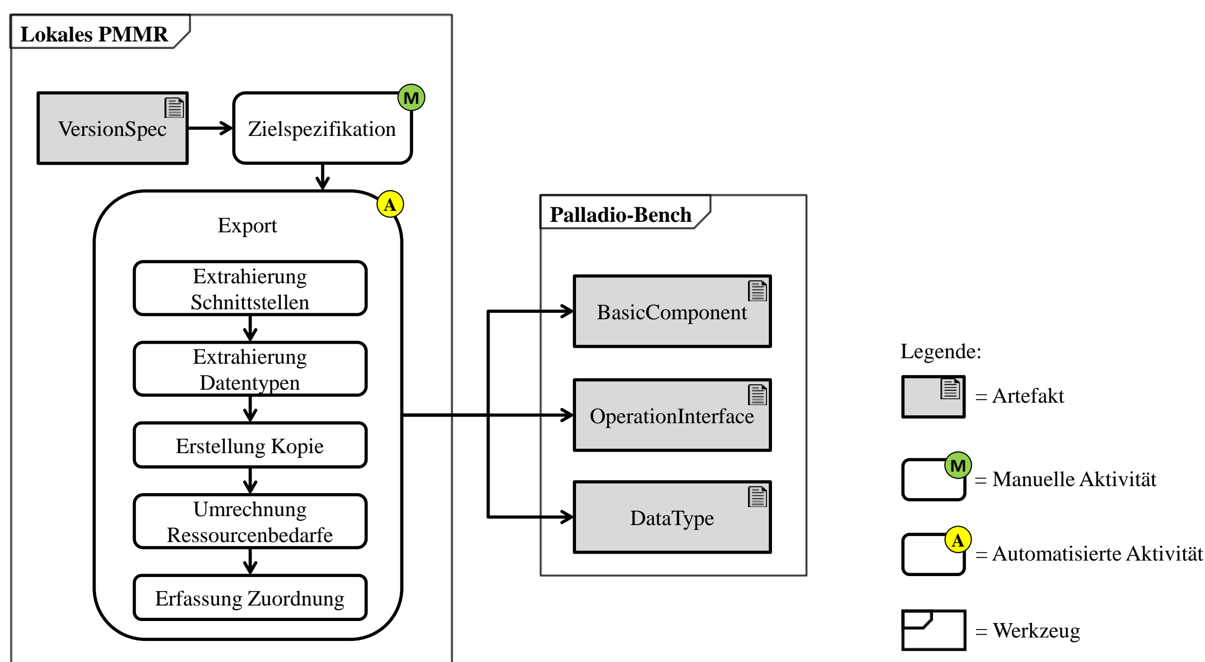


Abbildung 6-13: Export von Versionsspezifikationen nach existierenden PCM-Instanzen
Quelle: Eigene Darstellung

Von der resultierenden Menge der Versionsspezifikationen und Datentypen wird eine Kopie erstellt. Dabei werden Objekte vom Typ ComponentVersion als BasicComponent und Objekte vom Typ InterfaceVersion als OperationInterface instanziiert. Alle Versionsinformationen gehen dabei verloren. Das PMMR iteriert über alle BasicComponent-Instanzen und führt eine Umrechnung der Ressourcenbedarfe um. Für jeden Ressourcentyp wird anhand der Zielumgebung ein Umrechnungsfaktor berechnet und die Ressourcenbedarfe dementsprechend aktualisiert. Abschließend werden die kopierten Objekte im Repository-Modell eingefügt. Für jede

exportierte Versionsspezifikation wird ein PMMRMappingTuple-Eintrag angelegt (vgl. auch Abschnitt 6.3.2.3).

6.3.4 Integration mit dem Ansatz für Performancebewusstsein

Die Integration des PMMR mit dem Ansatz für Performancebewusstsein aus Kapitel 4 ist in Abbildung 6-14 dargestellt. Zur Durchführung einer Antwortzeitvorhersage für die fokussierte Komponente wird der gesamte Quelltext des Java-Projekts eingelesen und als KDM-Instanz dargestellt. Basierend auf der KDM-Instanz wird eine PCM-Instanz erstellt und simuliert. Die PCM-Instanz besteht aus unterschiedlichen Modellen, die sequenziell aufgebaut werden. Bei der Erstellung des Repository-Modells werden Komponenten sowie angebotene und benötigte Schnittstellen identifiziert und angelegt. Nach der Erstellung des System-, Resource-Environment-, Allocation- und Usage-Modells werden Komponentenspezifikationen um RDSEFF erweitert. Eine Integration mit dem PMMR wird durch die Erweiterung der Schritte zur Erstellung des Repository-Modells und der RDSEFF erreicht. Anstatt bei der wiederholten Durchführung dieselbe Version einer Java-Klasse zu verarbeiten, soll eine zuvor erstellte Komponenten-/Schnittstellenspezifikation wiederverwendet werden.

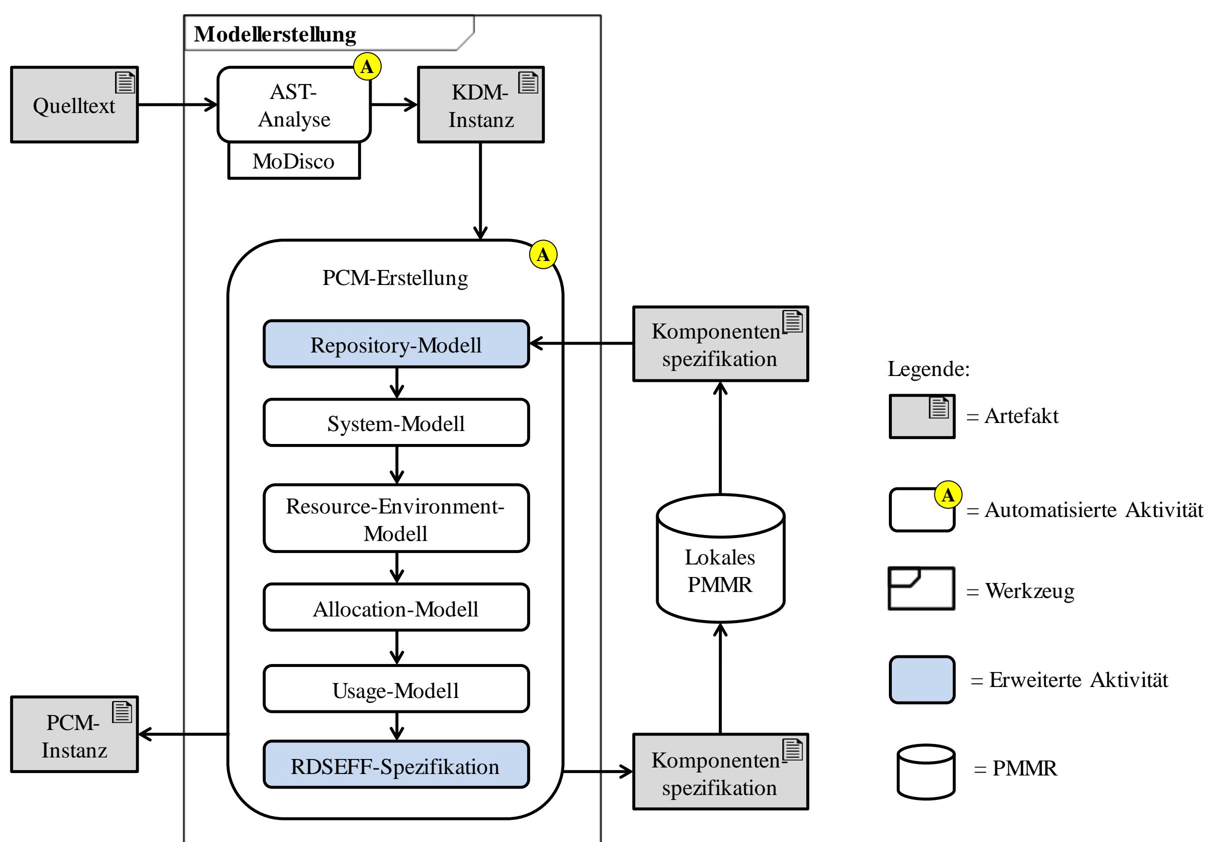


Abbildung 6-14: Integration des PMMR mit dem Ansatz für Performancebewusstsein
Quelle: Eigene Darstellung

6.3.4.1 Umfang der Wiederverwendung von Spezifikationen

Für eine möglichst effiziente Erstellung von Repository-Modellen sollen im PMMR sowohl BasicComponent- als auch OperationInterface-Instanzen zwischengespeichert werden. Referenzen von RDSEFF auf OperationInterface-Instanzen wären bei einer Vernachlässigung von Schnittstellenspezifikationen in einem inkonsistenten Zustand und müssten nach dem Auslesen aus dem PMMR wiederhergestellt werden. Der Umfang der wiederverwendbaren Spezifikationen aus dem PMMR hängt von den Quelltextänderungen zwischen zwei Durchläufen ab. Wie in Abbildung 6-15 dargestellt, können Quelltextänderungen an den angebotenen und benutzten Schnittstellen sowie an der Implementierung einer Komponente vorgenommen werden. Innerhalb einer Komponente oder Schnittstelle können eine oder mehrere Operationen geändert, hinzugefügt oder gelöscht werden. Auswirkungen der verschiedenen Arten von Änderungen werden im Folgenden beschrieben.

Änderungen von Schnittstellen

Dieselbe Schnittstelle kann gleichzeitig angeboten und benutzt werden. Die Art der Änderung hängt dann von der betrachteten Komponente ab. Schnittstellen werden geändert, indem beispielsweise Operationen gelöscht oder hinzugefügt werden. Eine existierende Operation kann hinsichtlich des Namens, der Rückgabewerts oder der Parametrisierung geändert werden. Nach solch einer Änderung kann die Spezifikation der Komponenten, welche diese Schnittstellen anbieten oder aufrufen, nicht ohne Anpassung wiederverwendet werden. Der Ansatz für Performancebewusstsein muss in diesem Fall alle betroffenen Schnittstellen und Komponenten identifizieren und hierfür neue Spezifikationen erzeugen. In der Java-Programmiersprache führt jede Änderung einer implementierten Schnittstelle zu einem fehlerhaften Zustand der implementierenden Klasse. Die Klasse muss dementsprechend auch angepasst werden. Aus diesem Grund kann davon ausgegangen werden, dass Änderungen einer implementierten Schnittstelle gleichzeitig mit der Änderung der implementierenden Klasse einhergehen. Bei der Wiederholung der Antwortzeitvorhersage identifiziert der Ansatz für Performancebewusstsein alle geänderten Schnittstellen und alle Komponenten, welche diese anbieten, und schließt diese von der Wiederverwendung aus.

Die Änderung einer Schnittstelle hat in der Java-Programmiersprache nicht zwangsläufig einen Einfluss auf die aufrufenden Klassen. Nur das Löschen und Ändern von aufgerufenen Operationen bedarf einer Anpassung an der Klasse. Da nach einer Änderung der Schnittstelle die vorherige Spezifikation nicht wiederverwendet wird, erstellt der Ansatz für Performancebewusstsein für die nutzenden Komponenten ebenso neue Spezifikationen.

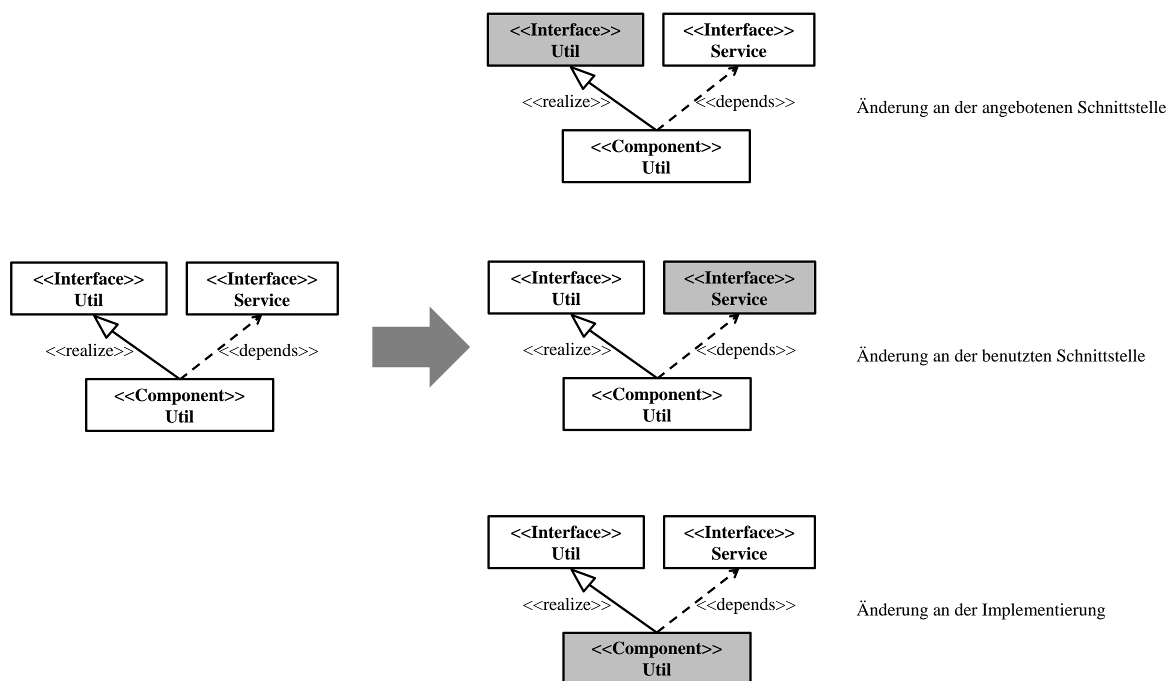


Abbildung 6-15: Beispiel für Änderungen zwischen zwei Durchläufen des Ansatzes für Performancebewusstsein
Quelle: Eigene Darstellung

Änderung einer Implementierung

Änderungen von Komponenten beeinflussen nicht die Spezifikation von Schnittstellen. Lediglich die Beziehungen zu benutzten und angebotenen Schnittstellen können von Änderungen betroffen sein. Der Ansatz für Performancebewusstsein identifiziert alle geänderten Komponenten und erstellt hierfür neue Spezifikationen.

6.3.4.2 Versionierung von Spezifikationen

Bei der Zwischenspeicherung von Schnittstellen- und Komponentenspezifikationen im PMMR werden diese anhand der Meta-Daten der zugrundeliegenden Java-Dateien versioniert. Bei der Wiederholung der Antwortzeitvorhersage werden wiederverwendbare Spezifikationen durch den Abgleich dieser Versionsinformationen identifiziert. Java-Dateien können ohne Version im Java-Projekt vorliegen, oder mit Hilfe eines Quelltextrepositories verwaltet werden.

Nicht versionierte Java-Dateien werden von einem einzigen Softwareentwickler bearbeitet und verfügen über einen Zeitstempel mit dem Zeitpunkt der letzten Änderung. Aus nicht versionierten Dateien abgeleitete Schnittstellen- oder Komponentenspezifikationen sollen nicht mit anderen Entwicklungsumgebungen ausgetauscht werden. Resultierende Spezifikationen werden im lokalen PMMR in einem eigenen Projekt, der nicht auf dem zentralen PMMR festgeschrieben wird, abgelegt. Als Version erhalten diese Spezifikationen den Zeitstempel der letzten Änderung der zugrundeliegenden Java-Datei.

Repository-Dateien werden von mehreren Softwareentwicklern gleichzeitig bearbeitet. Aus diesen Dateien abgeleitete Schnittstellen- oder Komponentenspezifikationen sollen mit anderen Entwicklungsumgebungen ausgetauscht werden. Die anderen Umgebungen können zur Durchführung von Antwortzeitvorhersagen mit Hilfe des Ansatzes für Performancebewusstsein die

ausgetauschten Spezifikationen wiederverwenden. Repository-Dateien verfügen über eine Revision. Das Quelltextrepository Subversion vergibt für Änderungen an versionierten Dateien eine fortlaufende Nummer als Revision (Collins-Sussman et al. 2009, S. 83). Folgende Revisionsnummern können dabei unterschieden werden:

- Revision des Repository: Dateien werden nicht individuell versioniert, das Festschreiben von Änderungen an einer oder mehreren Dateien führt zu einer neuen Revision des Repository, in dem diese Dateien verwaltet werden (Collins-Sussman et al. 2009, S. 44).
- Revision der letzten Änderung einer Datei: Zu jeder Datei wird auch die Revisionsnummer der letzten Änderung, von der die Datei betroffen war, verwaltet.

Aus versionierten Java-Dateien resultierende Spezifikationen werden auf einem zentralen PMMR-Projekt festgeschrieben. Als Version erhalten diese Spezifikationen die Revision der letzten Änderung der zugrundeliegenden Java-Datei. Die aktuelle Revision des Repositories wäre für diesen Zweck ungeeignet, da auch zwei identische Dateien unterschiedliche Revisionsnummer aufweisen können. Repository-Dateien können lokale Änderungen aufweisen, die noch nicht im Quelltextrepository festgeschrieben sind. Diese werden wie nicht versionierte Dateien behandelt und resultierende Spezifikationen werden nicht auf dem zentralen PMMR festgeschrieben.

Um festzustellen, ob eine Java-Datei versioniert ist, wird für das übergeordnete Projekt der *RepositoryProvider* abgerufen. Abhängig von dem eingesetzten Quelltextrepository wird eine entsprechende Implementierung des RepositoryProvider zurückgeliefert. Mit Hilfe dieses Objekts kann der Änderungszustand und die Revision einzelner Dateien abgefragt werden.

6.3.4.3 Abruf von existierenden Spezifikationen

Während der Erstellung des Repository-Modells (siehe Abbildung 6-14) identifiziert der erweiterte Ansatz für Performancebewusstsein zuerst wiederverwendbare Schnittstellen- und Komponentenspezifikationen. Anhand des Zeitstempels der letzten Änderung von nicht versionierten Java-Dateien und der Revision von versionierten Dateien werden im PMMR entsprechende Spezifikationen abgerufen.

Die bisherige Implementierung des Ansatzes sieht vor, dass zuerst über alle Klassendeklarationen iteriert wird und für jede dieser Klassen eine BasicComponent-Instanz angelegt und einem PCM-Repository hinzugefügt wird. Alle implementierten und aufgerufenen Java-Schnittstellen werden ermittelt und als OperationInterface-Instanz angelegt. Der *InterfaceBuilder* verwaltet dabei eine Liste von erzeugten Schnittstellenspezifikationen und stellt dadurch sicher, dass keine redundanten Instanzen angelegt werden. Der erweiterte Ansatz iteriert zuerst über alle Java-Schnittstellen und gleicht diese mit dem lokalen PMMR ab. Falls eine entsprechende Version verfügbar ist, wird diese für den Export aus dem PMMR vorgemerkt. Anschließend wird über alle Klassen iteriert. Diese werden für die Erzeugung einer neuen Komponentenspezifikation vorgemerkt, falls eine der folgenden Bedingungen zutrifft:

- Die Klasse implementiert eine Schnittstelle, deren Spezifikation aus dem PMMR nicht wiederverwendet werden kann.

- Die Klasse ruft eine Schnittstelle, deren Spezifikation aus dem PMMR nicht wiederverwendet werden kann, auf.
- Eine Komponentenspezifikation für die Version dieser Klasse konnte im PMMR nicht gefunden werden.

Die vorgemerkten Schnittstellen- und Komponentenspezifikationen werden aus dem PMMR exportiert. Im Gegensatz zum bisher beschriebenen Export von Versionsspezifikationen (vgl. Abschnitt 6.3.3.2) werden nicht alle angebotenen und verwendeten Schnittstellen der BasicComponent-Instanzen übernommen. Die OperationInterface-Instanzen werden in die Liste verfügbarer Spezifikationen im InterfaceBuilder hinzugefügt. Die BasicComponent-Instanzen werden im PCM-Repository hinzugefügt. Abschließend werden fehlende Spezifikationen neu erstellt.

6.3.4.4 Speicherung von Spezifikationen

RDSEFF müssen für Komponentenspezifikationen, die aus dem PMMR wiederverwendet wurden, nicht neu erstellt werden. Dadurch kann, abhängig vom Grad der Wiederverwendung, die Berechnung von Antwortzeitvorhersagen verkürzt werden. Zusätzlich werden Ressourcen des Entwicklerrechners weniger ausgelastet.

Schnittstellen- und Komponentenspezifikationen, die nicht wiederverwendet werden konnten, werden nach der Erstellung der RDSEFF ins PMMR importiert. Neu erstellte Spezifikationen können Beziehungen zu Objekten, die aus dem PMMR wiederverwendet wurden, aufweisen. Nach dem Import müssen diese Beziehungen zu den entsprechenden PMMR-Objekten wiederhergestellt werden. Das Ablegen von Spezifikationen ins PMMR kann asynchron ausgeführt werden, sodass die Antwortzeitvorhersage nicht davon betroffen wird.

6.4 Evaluation

Der Nutzen des Artefakts wird am Szenario der Integration des PMMR mit dem Ansatz für Performancebewusstsein nach Hevner et al. (2004) (vgl. Abschnitt 1.5) evaluiert. Die Dauer einer Antwortzeitvorhersage ohne die Wiederverwendung von bereits existierenden Komponentenspezifikationen wird hierfür gemessen und im Detail analysiert. Die Messung erfolgt auf einem handelsüblichen Geschäftslaptop mit einem Prozessor vom Typ Intel i5, einem Hauptspeicher im Umfang von 8 GB und einer SSD-Festplatte. In diesem Szenario verfügt ein Entwickler über eine mit dem Plugin für Performancebewusstsein ausgestattete IDE. Der Entwickler ruft für die Klasse BatchProcessingBean der Anwendung Cargo Tracker (vgl. Abschnitt 5.2.4.1) eine Antwortzeitvorhersage ab.

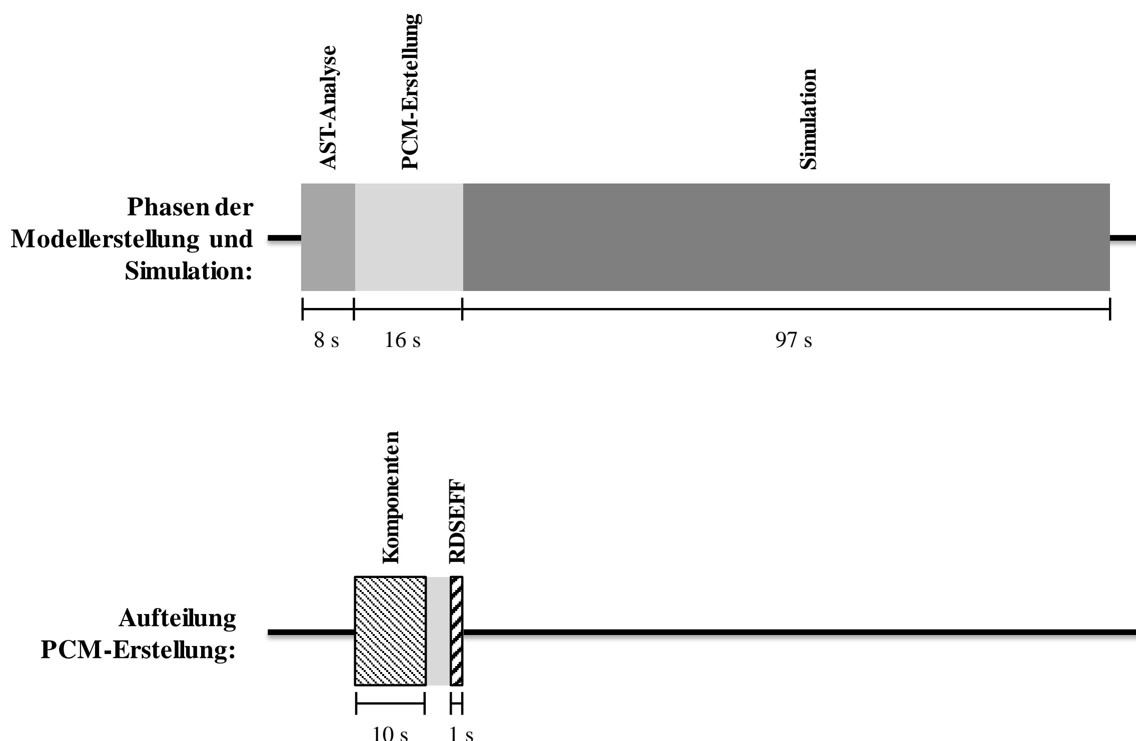


Abbildung 6-16: Optimierungspotenzial durch die Wiederverwendung von zwischengespeicherten Komponentenspezifikationen
Quelle: Eigene Darstellung

Die Ergebnisse der Messung über die Dauer der Vorhersage sind in Abbildung 6-16 dargestellt. Der Schritt der AST-Analyse benötigt ca. 8 Sekunden für die Erstellung einer KDM-Instanz. Für die Erstellung der PCM-Instanz benötigt das Plugin ca. 16 Sekunden. Im Anschluss erfolgt die Simulation des Modells über ca. 97 Sekunden. Die Dauer der Erstellung der PCM-Instanz gliedert sich in folgende Bestandteile auf: Komponentenspezifikationen werden innerhalb von ca. 10 Sekunden erstellt, RDSEFF innerhalb von ca. einer Sekunde. Die restlichen Modellbestandteile werden innerhalb von ca. 5 Sekunden erstellt. Abhängig von der Art um vom Umfang der Änderungen am Quellcode können durch die Wiederverwendung von Komponentenspezifikationen daher bis zu 11 Sekunden eingespart werden. Im PMMR zwischengespeicherte Komponentenspezifikationen beinhalten dabei schon RDSEFF. Das Optimierungspotenzial durch den Einsatz des PMMR beträgt ca. 69% von der Dauer der PCM-Erstellung und ca. 9% von der Gesamtdauer.

Unabhängig vom Einsatz des PMMR verarbeitet MoDisco immer den gesamten Quellcode während der AST-Analyse. Die Dauer der Simulation hingegen, kann vom Nutzer des Plugins über entsprechende Konfigurationsparameter verkürzt werden. Für die kontinuierliche Durchführung von Vorhersagen muss das vollständige Parsen von Java-Projekten auf Basis von MoDisco mit einem Verfahren zur Verarbeitung einzelner Quelltextänderungen ersetzt werden. Nach dem initialen Einlesen des gesamten Quelltextes könnten einzelne Änderungsoperationen registriert und in das Modell integriert werden. Eine weitere Reduzierung der Gesamtlaufzeit kann durch die Berechnung der Vorhersage auf Basis eines analytischen Verfahrens anstatt der Simulation erzielt werden.

6.5 Zusammenfassung und Ausblick

In diesem Kapitel wurde das Konzept und die Implementierung eines Performance Model Management Repository beschrieben. Dieses ermöglicht die Verwaltung und Versionierung von Spezifikation für Komponenten und Schnittstellen. Als Meta-Modell dient das PCM, welches für die Unterstützung der PMMR-Funktionen erweitert wurde. BasicComponent- und OperationInterface-Instanzen können ausgetauscht und durch PMMR-Nutzer parallel bearbeitet werden. Für die Wiederverwendung der Spezifikationen im Rahmen von Performanceevaluationen können diese zwischen dem PMMR und beliebigen PCM-Instanzen aus der Modellierungsumgebung ausgetauscht werden. Nachträgliche Änderungen innerhalb des PMMR werden auf die exportierten BasicComponent- und OperationInterface-Instanzen übertragen. Andererseits können Spezifikationen aus existierenden Performancemodellen in das PMMR importiert werden. Hardwarespezifische Ressourcenbedarfe werden dabei relativ zu einem Hardware-Benchmark-Ergebnis abgespeichert. Bei der Wiederverwendung von Komponentenspezifikationen können diese für eine bestimmte Zielumgebung umgerechnet werden. Das PMMR wurde auf der Grundlage der Palladio-Bench als Eclipse-Erweiterung implementiert. Als Modelldatenbank wird EMFStore verwendet. Durch die Integration mit dem Ansatz für Performancebewusstsein aus Kapitel 4 können bei der Durchführung von Antwortzeitvorhersagen Komponentenspezifikationen zwischengespeichert werden. Entsprechend dem Umfang der Quelltextänderungen zwischen zwei Antwortzeitvorhersagen kann die Berechnung beschleunigt und die Ressourcenauslastung des Entwicklerrechners verringert werden.

Für die Adressierung der im Abschnitt 6.1.2 formulierten Forschungsziele liefert der Ansatz des PMMR den wissenschaftlichen Beitrag der Erweiterung des PCM-Meta-Modells für die Abbildung und Verwaltung der Versionen von Komponenten- und Schnittstellenspezifikationen. Zu den Beiträgen des PMMR gehören noch:

- Der PMMR-Ansatz unterstützt die Umwandlung und Verwaltung hardwarespezifischer Ressourcenbedarfe.
- Der PMMR-Ansatz unterstützt den Austausch von Performancemodellen zwischen Benutzern.
- Die PMMR-Implementierung ist mit der Palladio-Bench integriert und unterstützt den Austausch von Modellartefakten mit PCM-Instanzen.

Die aktuelle Implementierung auf Basis von EMFStore sieht die vollständige Duplizierung eines Projekts in das lokale PMMR vor. Abhängig vom Umfang des PMMR-Projekts kann beispielsweise auf dem Entwicklerrechner dadurch ein großer Bedarf an Speicherkapazität resultieren. Zukünftige Forschung sollte den Einsatz alternativer Technologien für die Speicherung von Spezifikationen adressieren. Das Modellrepository CDO[36] könnte durch die Verwendung von Enterprise-Datenbanken die Verwaltung von Spezifikationen beschleunigen. Die Funktionen für das partielle Laden von Repository-Inhalten könnte die Verarbeitung großer Modelle besser unterstützen. Graphdatenbanken wie beispielsweise Neo4j[37] weisen für stark miteinander verflechtete Daten eine viel bessere Performance auf, als relationale Datenbanken

[36] http://www.eclipse.org/cdo/
[37] https://neo4j.com/

(Holzschuher/Peinl 2013). Mit EMF4J existiert auch eine Integration von EMF und Neo4j (Benelallam et al. 2014).

Die Einführung von Erweiterungen wird durch PCM auch mit Hilfe von Profile und Stereotypen unterstützt (Kramer et al. 2012). Hierdurch kann der Einfluss von Erweiterungen auf existierende PCM-Artefakte minimiert werden. Ein zusätzlicher Vorteil dieses Konzepts könnte eine einfachere Integration von Erweiterungen in bestehende Modellierungsumgebungen darstellen. PMMR-Erweiterungen könnten so als Plugin ausgeliefert werden (Kramer et al. 2012) und würden nicht mehr eine Anpassung der Palladio-Bench erfordern. Die Weiterentwicklung des PMMR sollte die Nutzung dieser Konzepte adressieren.

7 Zusammenfassung und Ausblick

Das Ziel der vorliegenden Arbeit war die Untersuchung, wie Softwareentwickler komponentenbasierter Unternehmensanwendungen mit Einsichten über die Performance ihrer Implementierung unterstützt werden können. Im Abschnitt 7.1 werden die Ergebnisse der adressierten Forschungsfragen zusammengefasst. Die Annahmen und Limitationen dieser Arbeit werden im Abschnitt 7.2 beschrieben. Die Vision einer Vernetzung dezentraler Entwicklungsumgebungen für die Erfassung und den Austausch von Optimierungsmaßnahmen wird abschließend im Abschnitt 7.4 vorgestellt.

7.1 Zusammenfassung der Ergebnisse

Die vorliegende Arbeit wurde von drei Forschungsfragen geleitet. Die Ergebnisse jeder einzelnen Forschungsfrage werden im Folgenden zusammengefasst.

7.1.1 Erste Forschungsfrage

Der Schwerpunkt der ersten Forschungsfrage lag auf die automatisierte Vorhersage der Antwortzeit für Komponenten in frühen Entwicklungsphasen und die Rückmeldung der Ergebnisse an dem Entwickler innerhalb der IDE. Die Forschungsfrage lautete:

Forschungsfrage 1 ***Wie können Antwortzeitvorhersagen für Softwarekomponenten aufgrund von wiederverwendeten Diensten durchgeführt und für die Schaffung von Performancebewusstsein bei Softwareentwicklern bereitgestellt werden?***

Zur Beantwortung der Forschungsfrage wurde ein Konzept für die Unterstützung von Performancebewusstsein bei Entwickler komponentenbasierter Software erstellt. Antwortzeitvorhersagen werden für eine fokussierte Komponente auf Basis der wiederverwendeten Dienste berechnet. Wiederverwendung im Sinne von Aufrufen externer Dienste wird durch die Analyse des Quelltexts einer Anwendung identifiziert. Die Abfolge der Aufrufe und die Struktur der Komponenten werden als Performancemodell abgebildet. Mit Hilfe von Informationen über das Verhalten der wiederverwendeten Komponenten wird das Modell parametrisiert und anschließend analysiert. Das Ergebnis der Antwortzeitvorhersage wird dem Entwickler innerhalb des Quelltexteditors dargestellt. Das entwickelte Konzept berücksichtigt Puffermechanismen und unterstützt die Spezifikation von Annahmen über die Parametrisierung von Aufrufen durch den Entwickler.

Das Konzept wurde für die Unterstützung von Performancebewusstsein in Java-EE-Entwicklungsumgebungen umgesetzt. Dazu wurde die Eclipse IDE um drei neue Werkzeuge erweitert. Eine Entwicklerschnittstelle adressiert den Kontext des Entwicklers. Hierüber können Antwortzeitvorhersagen für eine bestimmte Komponente angefordert werden. Die Entwicklerschnittstelle liefert Informationen über den Quelltext der Komponente und zeigt das Ergebnis der Antwortzeitvorhersage an. Während der Modellerstellung wird ausgehend von dem Quelltext zuerst mit Hilfe von MoDisco ein AST und anschließend mit Hilfe von SoMoX eine PCM-Instanz erstellt. Das Antwortzeitverhalten von wiederverwendeten Diensten wird durch die Instrumentierung des Bytecodes von laufenden Unternehmensanwendungen mit Hilfe des

Kieker Framework gemessen. Messungen werden anschließend in eine Performancedatenbank geladen und aggregiert. Während der Modellerstellung wird das Antwortzeitverhalten von wiederverwendeten Diensten von der Datenbank abgefragt und im Performancemodell integriert. Für die Vorhersage der Antwortzeit wird die PCM-Instanz simuliert. Übersteigt der vorhergesagte Wert einen bestimmten Schwellenwert, wird entweder ein Fehler oder eine Warnung im Quelltexteditor angezeigt.

Zur Untersuchung der Präzision der Vorhersagen wurde ein kontrolliertes Experiment durchgeführt. Antwortzeiten wurden für Dienste von Komponenten des SPECjEnterprise2010-Benchmarks zuerst vorhergesagt. Der Benchmark wurde anschließend ausgeführt und die tatsächlichen Antwortzeiten gemessen. Die durchschnittliche Abweichung der Antwortzeitvorhersagen lag bei dem Einsatz von 95-Perzentilen und Annotationen bei 38%.

7.1.2 Zweite Forschungsfrage

Der Schwerpunkt der dritten Forschungsfrage lag auf die Untersuchung und Bewertung des Einflusses von Performancebewusstsein auf die Antwortzeit von Softwarekomponenten. Die Forschungsfrage lautete:

Forschungsfrage 2 ***Welche Verbesserungen in der Antwortzeit der Dienste einer Komponente können durch die Unterstützung von Performancebewusstsein bei Softwareentwickler erzielt werden?***

Zur Beantwortung der zweiten Forschungsfrage wurde ein Experiment durchgeführt. Im Rahmen einer Aufgabe sollten Versuchspersonen das Antwortzeitverhalten existierender Komponenten optimieren. 13 Performancebugs wurden hierfür in die Cargo-Tracker-Anwendung eingebaut. Die Effektivität der Optimierungsmaßnahmen wurde anhand der Anzahl der beseitigten Performancebugs gemessen. Getestet wurde die Hypothese, dass die Nutzung des Ansatzes für Performancebewusstsein keinen Einfluss auf die Anzahl der beseitigten Performancebugs hat. Die unabhängige Variable des Experiments war die Verfügbarkeit des Ansatzes in der Entwicklungsumgebung der Versuchsperson. Der Ansatz für Performancebewusstsein wurde nur den Versuchspersonen der Testgruppe zur Verfügung gestellt. Am Experiment nahmen insgesamt 26 Versuchspersonen (21 Studenten, vier Wissenschaftler und ein Praktiker) teil. Versuchspersonen der Testgruppe haben durchschnittlich über drei Mal mehr Performancebugs behoben als die Versuchspersonen der Kontrollgruppe. Hypothesentests deuten darauf hin, dass sich die Verteilung der Anzahl der behobenen Performancebugs der Testgruppe signifikant von der Verteilung der Kontrollgruppe unterscheidet.

Ausgehend von den erhobenen Daten wurden noch weitere Hypothesen aufgestellt und getestet. Das Verhalten der Versuchspersonen wurde hinsichtlich der Zeit, welche diese innerhalb der Entwicklungsumgebung verbracht haben, untersucht. Getestet wurde hierfür die Hypothese, dass die Nutzung des Ansatzes für Performancebewusstsein keinen Einfluss auf den Anteil der Zeit, die Versuchspersonen während der Optimierung von Komponenten innerhalb ihrer IDE verbringen, hat. Versuchspersonen der Testgruppe hatten durchschnittlich 10,6% mehr Zeit innerhalb der IDE verbracht als Versuchspersonen der Kontrollgruppe. Hypothesentests deuten darauf hin, dass sich die Verteilung des Zeitanteils der Testgruppe signifikant von der Verteilung der Kontrollgruppe unterscheidet. Der Test auf Korrelation zwischen der Anzahl der

behobenen Fehler und der innerhalb der IDE verbrachten Zeit zeigte jedoch keine signifikante Korrelation an.

Der Einfluss eines methodischen Vorgehens bei der Untersuchung des Quelltextes auf die Ergebnisse der Teilnehmer wurde anhand der besuchten Klassen untersucht. Probanden der Testgruppe besuchten signifikant mehr relevante Klassen als Probanden der Kontrollgruppe. Der Test für die Korrelation zwischen der methodischen Untersuchung der relevanten Klassen und der Effektivität der Probanden der Testgruppe wies auf eine signifikante lineare Korrelation hin. Ausgehend von diesen Ergebnissen wurde abgeleitet, dass das Awareness Plugin eine methodische Untersuchung von Quellcode im Rahmen von Wartungsarbeiten unterstützt.

Hinsichtlich der Benutzerfreundlichkeit wurde untersucht, ob die Nutzung des Ansatzes für Performancebewusstsein einen Einfluss auf die Zufriedenheit der Versuchspersonen während der Optimierung von Komponenten hat. Im Vergleich zur Standard-IDE wurde die Benutzerfreundlichkeit der Erweiterung für Performancebewusstsein durchschnittlich besser bewertet. Auch hier konnte ein statistisch signifikanter Unterschied festgestellt werden.

7.1.3 Dritte Forschungsfrage

Der Schwerpunkt der dritten Forschungsfrage lag auf die Versionierung sowie Verwaltung von Komponenten- und Schnittstellenspezifikationen. Die Forschungsfrage lautete:

Forschungsfrage 3 ***Wie können Bausteine von Performancemodellen innerhalb einer Unternehmensumgebung versioniert, verwaltet und ausgetauscht werden?***

Zur Beantwortung der dritten Forschungsfrage wurde das Konzept eines Performance Model Management Repository für die Verwaltung und Versionierung von Spezifikation für Komponenten und Schnittstellen entwickelt. Spezifikationen werden auf einer zentralen Ablage abgespeichert und können durch unterschiedliche Teams bzw. Nutzer erstellt, bearbeitet und ausgelesen werden. Zur Abbildung des Lebenszyklus von Komponenten und Schnittstellen können die entsprechenden Spezifikationen mit Versionsinformationen versehen werden. Inhalte werden durch Nutzer lokal zwischengespeichert und Änderungen auf die zentrale Ablage festgeschrieben. Nutzer des PMMR können Spezifikationen in lokale Performancemodelle exportieren. Nachträgliche Änderungen innerhalb des PMMR werden auf die entsprechenden Performancemodelle übertragen. Andererseits können Spezifikationen aus Performancemodellen auch in das PMMR importiert werden. Hardwarespezifische Ressourcenbedarfe werden relativ zu einem Hardware-Benchmark-Ergebnis abgespeichert. Bei dem Export von Komponentenspezifikationen können diese für eine bestimmte Zielumgebung umgerechnet werden.

Das Konzept wurde auf der Grundlage der Palladio-Bench als Eclipse-Erweiterung implementiert und verwendet EMFStore als Modelldatenbank. Durch die Integration mit dem Ansatz für Performancebewusstsein können Komponentenspezifikationen während der Durchführung von Antwortzeitvorhersagen zwischengespeichert werden. Entsprechend dem Umfang der Quelltextänderungen zwischen zwei Antwortzeitvorhersagen kann die Berechnung beschleunigt und die Ressourcenauslastung des Entwicklerrechners verringert werden.

7.2 Beiträge für Wissenschaft und Praxis

Die vorliegende Arbeit liefert Beiträge für unterschiedliche Forschungsbereiche. Diese werden im Folgenden beschrieben.

Unterstützung von Performancebewusstsein bei Softwareentwickler

In dem Forschungsbereich Performancebewusstsein erweitert die vorliegende Arbeit die Wissensbasis um einen neuen Ansatz für die Unterstützung von Softwareentwickler mit Einsichten über das zukünftige Antwortzeitverhalten von Softwarekomponenten. Vorhersagen werden auf der Grandlage des Quelltextes von Komponenten und von Antwortzeitmessungen von wiederverwendeten Diensten durchgeführt. Der Ansatz kann während der Entwicklung, noch bevor der Quellcode ausführbar ist, angewendet werden. Bei der Modellierung des Antwortzeitverhaltens von Komponenten werden Puffermechanismen und Eingaben zur Parametrisierung durch den Entwickler berücksichtigt.

Die Evaluation mit Versuchspersonen erweitert die existierende Wissensbasis um die Erkenntnis über den Grad der Wirksamkeit des vorgestellten Ansatzes für Performancebewusstsein. Die Auswertung der Ergebnisse des Experiments liefern Aufschlüsse über den Faktor, um den Softwareentwickler mehr Performancebugs beheben, wenn diese über Performancebewusstsein verfügen.

Durch den Einsatz des Ansatzes in der Praxis können Entwicklungsabteilungen und Softwaredienstleister die Qualität der implementierten Softwarelösungen erhöhen, ohne dabei die Produktivität der Entwickler zu gefährden.

Im Kontext des Continuous Software Enginnerring unterstützt der Ansatz eine frühe, häufige und zeitnahe Bereitstellung von Feedback über die Performance der Software. Eine Antwortzeitvorhersage kann schon bevor eine Anwendung ausführbar ist durchgeführt werden. Diese kann nach einer Änderung auch direkt wiederholt werden. Ergebnisse werden dem Entwickler nach wenigen Minuten bereitgestellt.

Repositories für Performancemodelle

Im Forschungsbereich des SPE erweitert die vorliegende Arbeit die Wissensbasis um einen Ansatz für die Verwaltung und Versionierung von Performancemodellen in komplexen Unternehmensumgebungen. Im Rahmen der technischen Umsetzung liefert die Arbeit eine Erweiterung des Palladio Component Model für die:

- Versionierung von Komponenten- und Schnittstellenspezifikationen,
- Verwaltung von Komponenten- und Schnittstellenspezifikationen in einem zentralen Repository,
- Übertragung von Änderungen an Spezifikationen auf Performancemodelle und die
- Verwaltung von hardwarespezifischen Ressourcenbedarfen.

Durch den Einsatz des PMMR in der Praxis können konkrete Herausforderungen im Einsatz von Performancemodellen bewältigt werden. Entwicklungsabteilungen und Softwaredienstleister können mit Hilfe des PMMR Performancemodelle effizienter verwalten, austauschen und zeitgleich bearbeiten.

7.3 Annahmen und Limitationen

Die grundlegende Annahme des Ansatzes für Performancebewusstsein ist, dass neue Komponenten existierende Dienste aufrufen und die Antwortzeiten dieser Dienste gemessen werden können. Falls eine Komponente keine Dienste wiederverwendet, oder keine Antwortzeitmessungen für wiederverwendete Dienste vorliegen, kann keine Vorhersage getroffen werden. Der Ressourcenverbrauch und mögliche Ressourcenkonflikte werden bei der Berechnung der Antwortzeit vernachlässigt.

Plattform

Eine weitere Limitation der vorliegenden Arbeit ergibt sich aus der Entwicklung des Ansatzes für Performancebewusstsein am Beispiel von Java-EE-Komponenten. Daraus resultiert eine Einschränkung auf die Java-Programmiersprache und das EE-Framework. Von dieser Limitation ist jedoch nur die Abbildung des Quelltextes als Performancemodell betroffen. Hierbei werden Java-spezifische Konstrukte und Rahmenbedingungen adressiert. Das Einlesen und Abbilden des Quelltextes als AST mit Hilfe von MoDisco und die darauffolgende Erstellung des Performancemodells lassen sich wegen der syntaktischen Inkompatibilität nicht auf andere Programmiersprachen anwenden. Java ist jedoch eine der am weitesten verbreiteten Programmiersprachen und weist große Ähnlichkeiten zu anderen relevanten Sprachen, wie C++ und .Net, auf. Bei der Verwendung eines entsprechenden Parsers könnten auch diese Programmiersprachen als Grundlage für die Erstellung einer PCM-Instanz dienen. Die Simulation der resultierenden Performancemodelle und die Erhebung von Antwortzeitmessungen sind weiterhin uneingeschränkt anwendbar. Andere Frameworks für Enterprise Java, wie z.B. Spring, nutzen ähnliche Verfahren für Dependency Injection auf Basis von Annotationen im Quelltext und können daher auch verarbeitet werden.

Wiederverwendung

Die Berechnung von Antwortzeitvorhersagen berücksichtigt vor allem die Wiederverwendung von Diensten und Kontrollstrukturen. Diese werden in Form von Anweisungen aus Klassen und Methodendeklarationen extrahiert und in einem Performancemodell abgebildet. Die Java-EE-Spezifikation unterstützt jedoch auch die Steuerung des Kontrollflusses über Annotationen. Über die Methodenannotation @PostConstruct kann beispielsweise der Aufruf einer Methode nach der Initialisierung einer Komponente spezifiziert werden. Die entsprechende Methode muss dafür nicht explizit aufgerufen werden. Dieser Kontrollfluss wird durch den Ansatz weder erkannt noch abgebildet. Wie im Abschnitt 2.4 beschrieben, können neben Annotationen auch andere Arten der Konfiguration von DI eingesetzt werden. Diese werden durch den Ansatz auch nicht berücksichtigt. Die Wiederverwendung von Diensten wird aktuell nur über DI im Quelltext auf der Ebene von Methodenaufrufen identifiziert. Web Services und REST-basierte Dienste können jedoch auch über HTTP-Aufrufe anhand einer URL adressiert werden. Die Verarbeitung dieser Art der Wiederverwendung wird aktuell nicht unterstützt.

Performancemetriken

Die Anwendung des Ansatzes zielt auf die Verbesserung des Antwortzeitverhaltens von Komponenten ab. Bei der Optimierung der Antwortzeit können jedoch andere Performanceeigenschaften, wie z.B. der Ressourcenverbrauch, verschlechtert werden. Durch die Zwischenspeicherung von Rückgabewerten entfallen zusätzliche Aufrufe, dafür steigt aber die Auslastung des Hauptspeichers. Im Rahmen dieser Arbeit wurde der Aspekt des Ressourcenverbrauchs nicht berücksichtigt. Durch die Annahme einer höheren Priorität der Antwortzeit gegenüber dem Ressourcenverbrauch aus Benutzersicht, wurde diese Einschränkung hingenommen. Für die Adressierung dieser Limitation sollte zukünftige Forschung die parallele Abschätzung des Hauptspeicherverbrauchs während der Berechnung von Antwortzeitvorhersagen adressieren. Im Rahmen der Abbildung des Kontrollflusses kann auch die Instanziierung und die Rückgabe von Objekten berücksichtigt werden. Der Speicherbedarf eines Objektes kann ausgehend von der KDM-Instanz anhand des Typs der Attribute abgeschätzt werden. Das Verhalten des Garbage Collector könnte im Kontext der Analyse einer einzelnen Methodenausführung vernachlässigt werden.

In dem Experiment wurden Versuchspersonen dahingehend beeinflusst, die Abfrage einer Sammlung von größtenteils nicht benötigten Entities mit der Abfrage eines spezifischen Objekts zu ersetzen. Im Kontext von mobilen Anwendungen, könnte dies unter bestimmten Umständen das Ansprechverhalten einer Anwendung negativ beeinflussen. Die Auswirkungen von exzessiven Datenabfragen und von häufigen Einzelabfragen hängen vom Inhalt der Datenbank und vom Workload ab. Dieser Aspekt wurde im Rahmen des Experiments zugunsten einer einfacheren Auswertung ausgeblendet. Neben Aspekten der Performance können zusätzlich auch andere Qualitätsmerkmale, wie z.B. die Wartbarkeit von Quelltext, betroffen sein.

Eigenschaften, wie die Laufzeit des Garbage Collector, Computation Offloading oder Zeiten für die Inbetriebnahme sind für die in Rahmen dieser Arbeit adressierten Java-EE-Anwendungen nicht relevant. Anwendungsinstanzen aus den adressierten Domänen, wie beispielsweise dem elektronischen Handel, laufen über längere Zeiträume. Wichtiger als das Auftreten von Ausreißern sind die durchschnittlichen Antwortzeiten.

Evaluation

Die Präzision der Antwortzeitvorhersagen wird nur anhand des SPECjEnterprise2010-Benchmarks evaluiert. Die Effektivität des Ansatzes wird nur anhand der Anwendung Cargo-Tracker untersucht. Beide Anwendungen weisen eine große Abdeckung der Java-EE-Spezifikation auf, implementieren jedoch einfache Geschäftsprozesse.

Die Aufgabe der Versuchspersonen während dem Experiment bestand in der Optimierung von existierenden Komponenten durch die Beseitigung von Performancebugs. Ähnliche Wartungsarbeiten können auch in der Praxis vorkommen. Die Ergebnisse des Experiments sind jedoch nicht unbedingt auf Entwicklungsarbeiten übertragbar.

7.4 Ausblick

Ausgehend von den zugrundeligenden Annahmen und den resultierenden Limitationen dieser Arbeit werden im Folgenden Anknüpfungspunkte für zukünftige Forschung diskutiert.

Serviceorientierte Architekturen

Eine Anpassung des Ansatzes für Performancebewusstsein für den Einsatz im Kontext von SOA und Microservice-Architekturen ist vielversprechend und könnte verschiedene Aspekte adressieren. In seinem jetzigen Stadium identifiziert der Ansatz die Wiederverwendung von Diensten in Form von direkten Aufrufen oder DI mit Hilfe von Annotationen im Quelltext. Da bei serviceorientierten Architekturen wiederverwendete Komponenten auf entfernte Knoten liegen, werden benötigte Dienste nicht über direkte Aufrufe und DI, sondern viel eher über HTTP-Aufrufe integriert.

Beim Aufruf von Simple Object Access Protocol (SOAP) Services innerhalb einer Java-Umgebung implementiert der Aufrufer typischerweise spezielle Klassen, welche die Instanz einer Web Services Description Language (WSDL) des Web Service abbilden und Client Stubs genannt werden. Aufgrund der relativ einfachen Zuordnung der Namen der entsprechenden Stub-Klassen und -Operationen zum aufgerufenen entfernten Dienst, könnte der Ansatz für Performancebewusstsein für die Unterstützung dieser Technologie mit relativ wenig Aufwand erweitert werden. Um die Generierung von Stubs zur Design-Zeit zu vermeiden, können zur Laufzeit auch dynamische Proxies basierend auf der WSDL-Spezifikation generiert werden. Dynamische Proxies bilden den Namen der aufgerufenen Komponenten ab und werden somit vom Ansatz für Performancebewusstsein unterstützt. Eine dritte Art der Wiederverwendung von entfernten SOAP Services im Java-EE-Kontext stellt das Dynamic Invocation Interface dar. Ausgehend von der WSDL-Spezifikation und der Angabe der aufzurufenden Service-Namen werden hierbei zur Laufzeit entfernte Aufrufe generiert. Weil die Parametrisierung dieser Aufrufe evtl. nur zur Laufzeit ermittelt wird, kann der Ansatz für Performancebewusstsein für die Unterstützung dieser Aufrufe nicht eingesetzt werden. Die Anwendbarkeit des Ansatzes unter Verwendung anderer Frameworks als Java EE für den Aufruf von SOAP Services muss fallabhängig beurteilt werden. Trotz der weiten Verbreitung des SOAP-Standards in der Praxis, wird diese Technik aufgrund des erheblichen Overheads vor allem auf eingeschränkte Geräte eine immer geringere Anwendung finden.

Die Auflösung von REST-Aufrufen kann im Gegensatz zu SOAP erheblich komplexer sein. Diese leitgewichtige und aktuell weit verbreitete Technik sieht weder eine WSDL noch dedizierte Stub-Implementierungen vor. Stattdessen werden REST-Aufrufe im Java-Kontext über generische HTTP-Aufrufe realisiert. Der Ansatz für Performancebewusstsein hat bis auf eine aufgerufene URL, welche evtl. auch außerhalb des Quelltextes vorliegt oder erst zur Laufzeit ermittelt wird, keine Anhaltspunkte über die aufgerufene Komponente. Bei der Interpretierung der URL existieren schließlich auch verschiedene Herausforderungen. Dienste können mehrfach auf vielen Servernoten instanziiert sein. Serverknoten können sowohl direkt als auch über Load Balancer adressiert werden. Zielknoten können sowohl über IP-Adresse als auch über Domänennamen adressiert werden. URL-Pfade können von Netzwerk- oder Serverknoten transformiert bzw. uminterpretiert werden. Eine eindeutige Zuordnung von Performancemessungen zu den REST-basierten HTTP-Aufrufen ist daher nur in sehr einfachen Szenarien möglich.

Für die Realisierung komplexer Szenarien unter Einsatz von horizontaler Skalierung und Lastverteilung findet jedoch meistens auch eine Integration von entfernten Diensten mittels Messaging Middleware bzw. ESB, API Layer bzw. API Gateway oder Service Mesh statt. In diesen Fällen könnte die Erhebung von Performancemessungen für wiederverwendete Dienste mit Hilfe der Messaging Middleware oder dem API Layer anstatt von Instrumentierung im Quelltext erfolgen. Für den Ansatz für Performancebewusstsein würden sich dadurch einige Vorteile ergeben. Einerseits würden der Aufwand und Overhead der Instrumentierung entfallen. Entsprechende ESB oder API Gateways speichern und aggregieren Antwortzeitmessungen für aufgerufene Dienste standardmäßig. Darüber wäre die Identifizierung von Wiederverwendung auf Basis von voll qualifizierten Domänennamen anstatt Klassennamen möglich. Middleware- und Gateway-Implementierungen speichern meistens zu jedem eingehenden auch den entsprechenden ausgehenden Aufruf zusammen mit der beobachteten Antwortzeit. Der Ansatz für Performancebewusstsein könnte somit auch die Wiederwendendung auf Basis von HTTP-Aufrufen unterstützen. Hierfür müssten die Performancemessungen über API abrufbar sein.

Fog und Edge Computing

Auch im Kontext von Fog Computing erfolgt die Integration von wiederverwendeten Diensten über Netzwerkaufrufe. Meistens handelt es sich dabei um asynchrone Protokolle, wie Message Queuing Telemetry Transport (MQTT)[38]. Eine Anwendung des Ansatzes für Performancebewusstsein bringt bei einer rein asynchronen Kommunikation zwischen der aufrufenden und der wiederverwendeten Komponente keinen Mehrwert. In diesem Fall hat die Antwortzeit des aufgerufenen Dienstes keinen Einfluss auf die Performance der aufrufenden Komponente. Nichtdestotrotz könnte die Schaffung eines Bewusstseins über die Zeitverzögerung bei der Übertragung der asynchronen Nachricht einen Mehrwert für den Entwickler bieten. Unter diesen Bedingungen würde dem Entwickler eine Kontextinformation über die Verzögerung anstatt einer Antwortzeitvorhersage zurückgemeldet werden. Für die Bereitstellung solcher Einsichten müssten Messungen im MQTT Broker durchgeführt und ausgelesen werden. Doch auch mobile Endgeräte können entfernte Dienste über HTTP-Aufrufe wiederverwenden. In diesem Fall würden Aufrufe von einem API Gateway entgegengenommen werden und die weiter oben beschriebenen Erweiterungen des Ansatzes könnten angewendet werden.

Im Bereich des Edge Computing wäre die Schaffung von Bewusstsein über den Energieverbrauch von Anwendungen eine sinnvolle Erweiterung des Ansatzes. Ausgehend von dem ermittelten Kontrollfluss und dem resultierenden Performancemodell könnten Laufzeiten und der Hauptspeicherverbrauch abgeschätzt werden.

Data Federation

Eine der grundlegenden Annahmen dieser Arbeit ist die Verfügbarkeit einer zentralen Datenbank zur Abspeicherung und Bereitstellung von Antwortzeitmessungen. Bei einer Erweiterung des Ansatzes für das Abrufen von Performancemessungen bei Integrationsplattformen im Kontext von SOA und Microservice-Architekturen könnte diese Annahme nicht mehr zutreffen. Einerseits können mehrere Middleware-Plattformen von ein und derselben Anwendung aufgerufen werden. In diesem Fall müsste der Ansatz einen umfangreichen Mechanismus für die Anbindung mehrerer heterogenen Performancedatenbanken unterstützen. Anhand der

[38] http://mqtt.org/

aufgerufenen Endpunkte müsste dann ableitet werden, um welche Plattform es sich handelt. Andererseits ist es auch wahrscheinlich, dass Entwickler API Gateways fremder Organisationen integrieren. In diesem Fall müsste die Entwicklung gemeinsamer Standards für den Austausch von Performancekennzahlen adressiert werden.

Austausch von Optimierungsmaßnahmen

Eine Gemeinsamkeit der existierenden Ansätze für Performancebewusstsein ist, dass diese isoliert durch Entwickler eingesetzt werden und nicht untereinander vernetzt sind. Einsichten über die Performance von Systemen werden lokal abgerufen, bereitgestellt und verarbeitet. Die Reaktion des Entwicklers auf diese Beobachtungen wird weder verwertet noch gespeichert. Das Potenzial, aus dieser Reaktion allgemeine Optimierungsmaßnahmen für bestimmte Performanceprobleme abzuleiten, bleibt ungenutzt. Zukünftige Forschung sollte untersuchen, wie Optimierungsmaßnahmen identifiziert, klassifiziert, gespeichert und für den Kontext anderer Entwickler abgerufen werden können. Ein Ausblick auf die Erweiterung des Ansatzes für Performancebewusstsein aus der vorliegenden Arbeit zur Unterstützung des Austauschs von Optimierungsmaßnahmen ist in Abbildung 7-1 dargestellt.

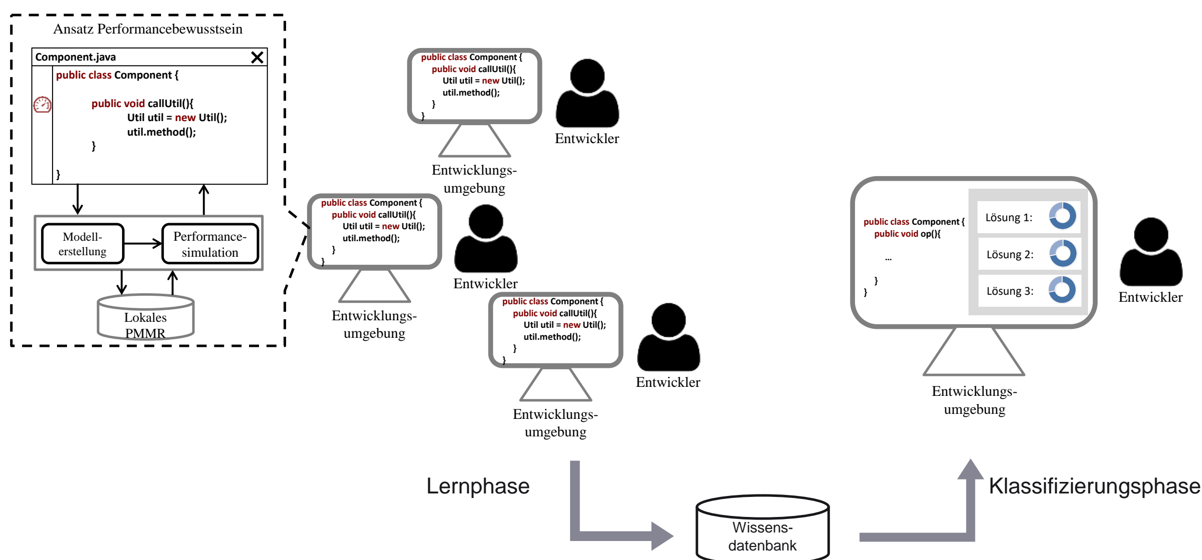


Abbildung 7-1: ***Erweiterung des Ansatzes für Performancebewusstsein für den Austausch von Optimierungsmaßnahmen***
Quelle: Eigene Darstellung

Der Ansatz für Performancebewusstsein soll in den Entwicklungsumgebungen einer Gruppe von Entwickler integriert werden und Vorhersagen über die Antwortzeit des aktuell bearbeiteten Quelltextes bereitstellen. Änderungsmaßnahmen, welche aufgrund der angezeigten Hinweise erfolgen, sollen erfasst und bewertet werden. Falls eine Änderung zu einer Verbesserung der Performance führt, soll diese als Optimierungsmaßnahme eingestuft und auf einer zentralen Wissensbasis abgespeichert werden. Neben der eingeführten Änderung soll auch die Implementierung in der Version vor und nach der Optimierung sowie der Kontext des Entwicklers mitgespeichert werden.

Der Kontext kann unter anderem folgende Aspekte umfassen:

- eingesetzte Programmiersprache,
- eingesetzte Frameworks und Technologien inkl. deren aktuelle Konfiguration und
- wiederverwendete Dienste und Bibliotheken.

Von der Speicherung einer Optimierungsmaßnahme bzw. einer Implementierung in Form von Quelltext wäre implizit auch geistiges Eigentum betroffen. Um dies zu verhindern soll eine Darstellung des Quelltextes als Performancemodell in Form einer PCM-Instanz in der Wissensbasis erfasst werden. Andere Entwickler, die auch den erweiterten Ansatz für Performancebewusstsein nutzen, sollen Vorschläge über mögliche Optimierungsmaßnahmen erhalten. Der aktuell bearbeitete Quelltext soll zuerst als Performancemodell abgebildet werden. Anschließend soll die Wissensbasis nach einer ähnlichen Struktur durchsucht werden. Anhand der Ausprägung des Kontexts des Entwicklers sollen die identifizierten Optimierungsmaßnahmen klassifiziert und nach dem Grad der Übereinstimmung priorisiert werden. Als Ergebnis sollen in der Entwicklungsumgebung eine Untermenge der relevanten Maßnahmen angezeigt werden. Der Entwickler soll die Möglichkeit haben die angezeigten Ergebnisse hinsichtlich deren Verwertbarkeit zu bewerten. Aufgrund dieser Bewertungen soll der Klassifizierungsmechanismus verbessert werden können.

Neben automatisch erfassten Optimierungsmaßnahmen sollen Entwickler die Möglichkeit haben, selbständig Hinweise in Form von Text ablegen zu können. Auf Performance bezogene Anmerkungen, wie z.B. kritische Konfigurationsparameter, sollen dadurch ausgetauscht werden können. Entwickler in einem ähnlichen Kontext sollen diese Hinweise eingeblendet bekommen.

Die Einbettung des erweiterten Ansatzes in einem Ökosystem für den Austausch von Optimierungsmaßnahmen ist in Abbildung 7-2 dargestellt. Die Wissensdatenbank könnte als Cloud-Dienst bereitgestellt werden. Verschiedene Communities von Entwickler, wie beispielsweise Projektteams oder ganze Unternehmen, können über diesen Dienst miteinander vernetzt werden. Communities können hinsichtlich der Bereitschaft Wissen auszutauschen unterschiedliche Ansätze verfolgen. Der Austausch von Optimierungsmaßnahmen und Hinweisen soll daher auf einzelne Communities oder auf Gruppen von Communities eingegrenzt werden können.

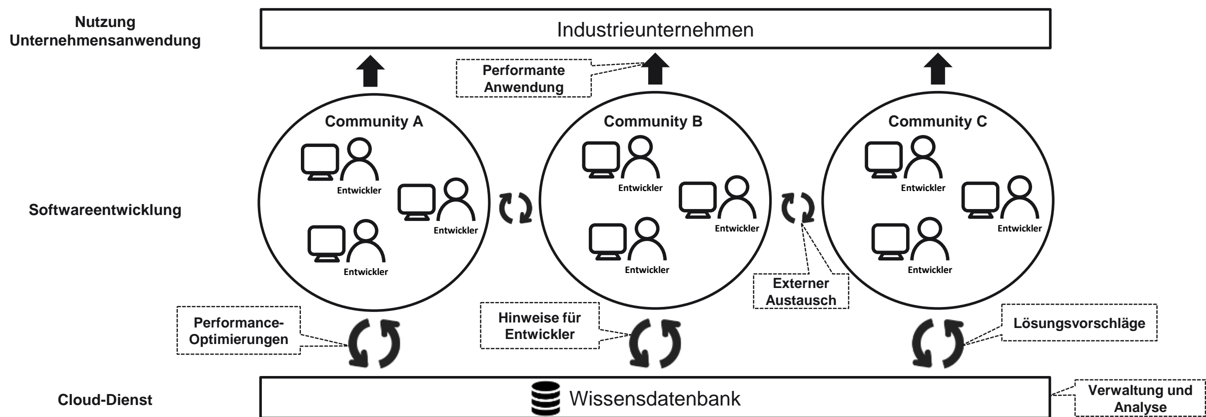


Abbildung 7-2: ***Ökosystem für den Austausch von Optimierungsmaßnahmen zwischen Communities***
Quelle: Eigene Darstellung

Der Ansatz für Performancebewusstsein aus der vorliegenden Arbeit sieht die Erfassung von Performancemessungen aus dem produktiven Betrieb vor. Diese werden für die Durchführung von Antwortzeitvorhersagen für die Unterstützung von Entwickler eingesetzt. Zukünftige Forschung könnte untersuchen, wie auch umgekehrt Informationen aus Entwicklungsumgebungen für die Unterstützung des Betriebs verwertet werden können. Performancevorhersagen aus den Entwicklungsumgebungen könnten beispielsweise für eine präventive Anpassung der Infrastruktur im Betrieb eingesetzt werden.

Für eine Erhöhung der Akzeptanz von Ansätzen für Performancebewusstsein bei Entwickler sollten umfangreiche Evaluationen durchgeführt werden. Die Verbreitung dieser Ansätze in der Praxis könnte dadurch vorangetrieben werden.

Literaturverzeichnis

Altmanninger, K.; Seidl, M.; Wimmer, M. (2009): A survey on model versioning approaches. In: International Journal of Web Information Systems, Band 5 (2009) Nr. 3, S. 271-304.

Andrikopoulos, V.; Benbernou, S.; Papazoglou, M.P. (2012): On the Evolution of Services. In: IEEE Transactions on Software Engineering, Band 38 (2012) Nr. 3, S. 609-628.

Anquetil, N.; Lethbridge, T.C. (1999): Recovering software architecture from the names of source files. In: Journal of Software Maintenance, Band 11 (1999) Nr. 3, S. 201-221.

Basili, V.R.; Caldiera, G. (1988): Reusing existing software. Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, 1988.

Basili, V.R.; Caldiera, G.; Rombach, H.D. (1994): The goal question metric approach. In: Encyclopedia of Software Engineering, Band 2 (1994) Nr. 1994, S. 528-532.

Bauer, C.; King, G. (2006): Java Persistence with Hibernate, Manning Publications Co., Greenwich, CT, USA 2006.

Becker, S. (2008): Coupled Model Transformations for QoS Enabled Component-Based Software Design. Dissertation, Universität Oldenburg 2008.

Becker, S.; Hasselbring, W.; Hoorn, A.v.; Reussner, R. (Hrsg.) (2013): Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days (Band 1083). CEUR-WS.org, Karlsruhe, Germany 2013.

Becker, S.; Hauck, M.; Trifu, M.; Krogmann, K.; Kofron, J. (2010): Reverse Engineering Component Models for Quality Predictions. Vorgestellt in: 14th European Conference on Software Maintenance and Reengineering, S. 194-197.

Becker, S.; Koziolek, H.; Reussner, R. (2009): The Palladio component model for model-driven performance prediction. In: Journal of Systems and Software, Band 82 (2009) Nr. 1, S. 3-22.

Benelallam, A.; Gómez, A.; Sunyé, G.; Tisi, M.; Launay, D. (2014): Neo4EMF, A Scalable Persistence Layer for EMF Models. In: Modelling Foundations and Applications. Cabot, J.; Rubin, J. (Hrsg.). Springer International Publishing, Cham 2014, S. 230-241.

Bennett, C.; Myers, D.; Storey, M.A.; German, D.M.; Ouellet, D.; Salois, M.; Charland, P. (2008): A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. In: Journal of Software Maintenance and Evolution: Research and Practice, Band 20 (2008) Nr. 4, S. 291-315.

Bertolino, A.; Mirandola, R. (2004): CB-SPE Tool: Putting component-based performance engineering into practice. In: Component-Based Software Engineering (Band 3054). Crnkovic, I.; Stafford, J.A.; Schmidt, H.W.; Wallnau, K. (Hrsg.). Springer, Berlin, Heidelberg 2004, S. 233-248.

Binder, W.; Hulaas, J.; Moret, P. (2007): Advanced Java bytecode instrumentation. Vorgestellt in: 5th International Symposium on Principles and Practice of Programming in Java, Lisboa, Portugal, S. 135-144.

Boehm, B.W. (1973): Characteristics of software quality. In: TRW Software Series: TRW-SS, Redondo Beach, Ca.: TRW, 1973, (1973).

Boehm, B.W.; Brown, J.R.; Lipow, M. (1976): Quantitative evaluation of software quality. Vorgestellt in: 2nd International Conference on Software engineering, S. 592-605.

Bolch, G.; Greiner, S.; de Meer, H.; Trivedi, K.S. (2006): Queueing networks and Markov chains: modeling and performance evaluation with computer science applications, John Wiley & Sons, Hoboken, NJ, USA 2006.

Bonomi, F.; Milito, R.; Natarajan, P.; Zhu, J. (2014): Fog computing: A platform for internet of things and analytics. In: Big data and internet of things: A roadmap for smart environments. Bessis, N.; Dobre, C. (Hrsg.). Springer, Cham 2014, S. 169-186.

Bortz, J.; Döring, N. (2007): Forschungsmethoden und Evaluation für Human-und Sozialwissenschaftler: Limitierte Sonderausgabe, Springer-Verlag, Heidelberg, Germany 2007.

Bosch, J. (2014): Continuous software engineering: An introduction. In: Continuous software engineering. Bosch, J. (Hrsg.). Springer 2014, S. 3-13.

Brogi, A.; Forti, S. (2017): QoS-Aware Deployment of IoT Applications Through the Fog. In: IEEE Internet of Things Journal, Band 4 (2017) Nr. 5, S. 1185-1192.

Brooke, J. (1996): SUS - A quick and dirty usability scale. In: Usability Evaluation in Industry, Band 189 (1996) Nr. 194, S. 4-7.

Brosig, F.; Huber, N.; Kounev, S. (2011): Automated extraction of architecture-level performance models of distributed component-based systems. Vorgestellt in: 26th International Conference on Automated Software Engineering, Lawrence, KS, USA, S. 183-192.

Brosig, F.; Kounev, S.; Krogmann, K. (2009): Automated extraction of palladio component models from running enterprise Java applications. Vorgestellt in: 4th International Conference on Performance Evaluation Methodologies and Tools, Pisa, Italy, S. 1-10.

Bruegge, B.; Creighton, O.; Helming, J.; Kögel, M. (2008): Unicase – an ecosystem for unified software engineering research tools. Vorgestellt in: 3rd International Conference on Global Software Engineering, S. 12-17.

Bruneliere, H.; Cabot, J.; Jouault, F.; Madiot, F. (2010): MoDisco: a generic and extensible framework for model driven reverse engineering. Vorgestellt in: International Conference on Automated Software Engineering, Antwerp, Belgium, S. 173-174.

Bruneo, D.; Fritz, T.; Keidar-Barner, S.; Leitner, P.; Longo, F.; Marquezan, C.; Metzger, A.; Pohl, K.; Puliafito, A.; Raz, D.; Roth, A.; Salant, E.; Segall, I.; Villari, M.; Wolfsthal, Y.; Woods, C. (2014): CloudWave: Where adaptive cloud management meets DevOps. Vorgestellt in: Symposium on Computers and Communications, S. 1-6.

Brunnert, A.; Danciu, A.; Krcmar, H. (2015a): Towards a Performance Model Management Repository for Component-based Enterprise Applications. Vorgestellt in: 6th International Conference on Performance Engineering, Austin, Texas, USA, S. 321-324.

Brunnert, A.; Krcmar, H. (2015): Continuous performance evaluation and capacity planning using resource profiles for enterprise applications. In: Journal of Systems and Software, Band 123 (2015).

Brunnert, A.; van Hoorn, A.; Willnecker, F.; Danciu, A.; Hasselbring, W.; Heger, C.; Herbst, N.; Jamshidi, P.; Jung, R.; von Kistowski, J.; Koziolek, A.; Kroß, J.;

Spinner, S.; Vögele, C.; Walter, J.; Wert, A. (2015b): Performance-oriented DevOps: A Research Agenda (Technical Report SPEC-RG-2015-01). SPEC Research Group - DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), 2015b.

Brunnert, A.; Vögele, C.; Danciu, A.; Pfaff, M.; Mayer, M.; Krcmar, H. (2014): Performance Management Work. In: Business & Information Systems Engineering, Band 6 (2014) Nr. 3, S. 177-179.

Brunnert, A.; Vögele, C.; Krcmar, H. (2013): Automatic Performance Model Generation for Java Enterprise Edition (EE) Applications. In: Computer Performance Engineering. Balsamo, M.S.; Knottenbelt, W.J.; Marin, A. (Hrsg.). Springer, Berlin, Heidelberg 2013, S. 74-88.

Bulej, L.; Bureš, T.; Gerostathopoulos, I.; Horký, V.; Keznikl, J.; Marek, L.; Tschaikowski, M.; Tribastone, M.; Tůma, P. (2015): Supporting Performance Awareness in Autonomous Ensembles. In: Software Engineering for Collective Autonomic Systems: The ASCENS Approach. Wirsing, M.; Hölzl, M.; Koch, N.; Mayer, P. (Hrsg.). Springer International Publishing, Cham 2015, S. 291-322.

Bulej, L.; Bureš, T.; Horký, V.; Keznikl, J.; Tůma, P. (2012a): Performance Awareness in Component Systems: Vision Paper. Vorgestellt in: 36th Computer Software and Applications Conference Workshops, S. 514-519.

Bulej, L.; Bureš, T.; Keznikl, J.; Koubková, A.; Podzimek, A.; Tůma, P. (2012b): Capturing performance assumptions using stochastic performance logic. Vorgestellt in: 3rd International Conference on Performance Engineering, Boston, Massachusetts, USA, S. 311-322.

Bureš, T.; Horký, V.; Kit, M.; Marek, L.; Tůma, P. (2014): Towards Performance-Aware Engineering of Autonomic Component Ensembles. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change. Margaria, T.; Steffen, B. (Hrsg.). Springer, Berlin, Heidelberg 2014, S. 131-146.

Canfora, G.; Penta, M.D.; Esposito, R.; Villani, M.L. (2005): An approach for QoS-aware service composition based on genetic algorithms. Vorgestellt in: 7th Conference on Genetic and Evolutionary Computation, Washington DC, USA, S. 1069-1075.

Chappell, D.A. (2004): Enterprise Service Bus, O'Reilly Media 2004.

Chen, T.-H.; Shang, W.; Jiang, Z.M.; Hassan, A.E.; Nasser, M.; Flora, P. (2014): Detecting performance anti-patterns for applications developed using object-relational mapping. Vorgestellt in: 36th International Conference on Software Engineering, Hyderabad, India, S. 1001-1012.

Chikofsky, E.J.; Cross, J.H. (1990): Reverse engineering and design recovery: a taxonomy. In: IEEE Software, Band 7 (1990) Nr. 1, S. 13-17.

Chouambe, L.; Klatt, B.; Krogmann, K. (2008): Reverse Engineering Software-Models of Component-Based Systems. Vorgestellt in: 12th European Conference on Software Maintenance and Reengineering, S. 93-102.

Ciolkowski, M.; Differding, C.; Laitenberger, O.; Münch, J. (1997): Empirical investigation of perspective-based reading: A replicated experiment (Technical Report No. 13/97). International Software Engineering Research Network (ISERN), 1997.

Cito, J.; Leitner, P.; Gall, H.C.; Dadashi, A.; Keller, A.; Roth, A. (2015): Runtime metric meets developer: building better cloud applications using feedback. Vorgestellt in:

International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), Pittsburgh, PA, USA, S. 14-27.

Collins-Sussman, B.; Fitzpatrick, B.W.; Pilato, C.M. (2009): Subversion 1.6 Official Guide: Version Control with Subversion, Fultus Corporation 2009.

Conradi, R.; Westfechtel, B. (1997): Towards a uniform version model for software configuration management. In: Software Configuration Management. Conradi, R. (Hrsg.). Springer, Berlin, Heidelberg 1997, S. 1-17.

Conradi, R.; Westfechtel, B. (1998): Version models for software configuration management. In: ACM Computing Surveys, Band 30 (1998) Nr. 2, S. 232-282.

Cornelissen, B.; Zaidman, A.; Deursen, A.v.; Moonen, L.; Koschke, R. (2009a): A Systematic Survey of Program Comprehension through Dynamic Analysis. In: IEEE Transactions on Software Engineering, Band 35 (2009a) Nr. 5, S. 684-702.

Cornelissen, B.; Zaidman, A.; Deursen, A.v.; Rompaey, B.v. (2009b): Trace visualization for program comprehension: A controlled experiment. Vorgestellt in: 17th International Conference on Program Comprehension, S. 100-109.

Cortellessa, V.; Frittella, L. (2007): A framework for automated generation of architectural feedback from software performance analysis. Vorgestellt in: European Performance Engineering Workshop, S. 171-185.

Costa, L.B.; Al-Kiswany, S.; Yang, H.; Ripeanu, M. (2014a): Supporting storage configuration for I/O intensive workflows. Vorgestellt in: 28th International Conference on Supercomputing, Munich, Germany, S. 191-200.

Costa, L.B.; Brunet, J.; Hattori, L.; Ripeanu, M. (2014b): Experience with Using a Performance Predictor during Development: A Distributed Storage System Tale. Vorgestellt in: 2nd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering, S. 13-19.

Courtois, M.; Woodside, M. (2000): Using regression splines for software performance analysis. Vorgestellt in: 2nd International Workshop on Software and Performance, Ottawa, Ontario, Canada, S. 105-114.

Danciu, A.; Brunnert, A.; Krcmar, H. (2014): Towards Performance Awareness in Java EE Development Environments. Vorgestellt in: Symposium on Software Performance, Stuttgart, Germany, S. 152-159.

Danciu, A.; Brunnert, A.; Krcmar, H. (2015a): A Performance Model Management Repository Based on the Palladio Component Model. In: Softwaretechnik-Trends, Band 35 (2015a) Nr. 3.

Danciu, A.; Chrusciel, A.; Brunnert, A.; Krcmar, H. (2015b): Performance Awareness in Java EE Development Environments. In: Computer Performance Engineering (Band 9272). Beltrán, M.; Knottenbelt, W.; Bradley, J. (Hrsg.). Springer International Publishing 2015b, S. 146-160.

Danciu, A.; Krcmar, H. (2018): To What Extent Does Performance Awareness Support Developers in Fixing Performance Bugs? In: Computer Performance Engineering (Band 11178). Bakhshi, R.; Ballarini, P.; Barbot, B.; Castel-Taleb, H.; Remke, A. (Hrsg.). Springer International Publishing, Cham 2018, S. 14-29.

Danciu, A.; Kroß, J.; Brunnert, A.; Willnecker, F.; Vögele, C.; Kapadia, A.; Krcmar, H. (2015c): Landscaping Performance Research at the ICPE and its Predecessors: A

Systematic Literature Review. Vorgestellt in: 6th International Conference on Performance Engineering, Austin, Texas, USA, S. 91-96.

Deissenboeck, F.; Juergens, E.; Lochmann, K.; Wagner, S. (2009): Software quality models: Purposes, usage scenarios and requirements. Vorgestellt in: ICSE Workshop on Software Quality, Vancouver, BC, Canada, S. 9-14.

DeMichiel, L. (2013): JSR 338: Java™ Persistence API, Version 2.1 (Specification JSR-000338). Oracle, 2013.

DeMichiel, L.; Shannon, B. (2013): Java™ Platform, Enterprise Edition (Java EE) Specification, v7 (Specification JSR-342). Oracle, 2013.

Dourish, P.; Bellotti, V. (1992): Awareness and coordination in shared workspaces. Vorgestellt in: Conference on Computer-supported Cooperative Work, Toronto, Ontario, Canada, S. 107-114.

Dourish, P.; Bly, S. (1992): Portholes: supporting awareness in a distributed work group. Vorgestellt in: Conference on Human Factors in Computing Systems, Monterey, California, USA, S. 541-547.

Dromey, R.G. (1995): A model for software product quality. In: IEEE Transactions on Software Engineering, Band 21 (1995) Nr. 2, S. 146-162.

Dufour, B.; Driesen, K.; Hendren, L.; Verbrugge, C. (2003): Dynamic metrics for java. Vorgestellt in: 18th Conference on Object-oriented Programing, Systems, Languages, and Applications, Anaheim, California, USA, S. 149-168.

Dugan, R.F.; Glinert, E.P.; Shokoufandeh, A. (2002): The Sisyphus database retrieval software performance antipattern. Vorgestellt in: 3rd International Workshop on Software and Performance, Rome, Italy, S. 10-16.

Elliott, A.C.; Woodward, W.A. (2007): Statistical analysis quick reference guidebook: With SPSS examples, Sage, Thousand Oaks, CA, USA 2007.

Espinosa, A.; Margalef, T.; Luque, E. (1998): Automatic performance evaluation of parallel programs. Vorgestellt in: 6th Workshop on Parallel and Distributed Processing, S. 43-49.

Favre, J.M.; Duclos, F.; Estublier, J.; Sanlaville, R.; Auffret, J.J. (2001): Reverse engineering a large component-based software product. Vorgestellt in: 5th European Conference on Software Maintenance and Reengineering, S. 95-104.

Fowler, M. (2005): ServiceOrientedAmbiguity. https://martinfowler.com/bliki/ServiceOrientedAmbiguity.html, zugegriffen am 13.10.2019.

Gallardo, D.; Burnette, E.; McGovern, R. (2003): Eclipse in action: a guide for java developers, Manning Publications Co. 2003.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995): Design patterns: elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA 1995.

Garvin, D.A. (1984): What Does "Product Quality" Really Mean? In: Sloan management review, Band 25 (1984).

Göb, A. (2013): SOA und Softwarequalität. Dissertation, Technische Universität München 2013.

Gosling, J.; Joy, B.; Steele, G.; Bracha, G.; Buckley, A. (2013): The Java® Language Specification (Specification JSR-000901). Oracle, 2013.

Grady, R.B. (1992): Practical software metrics for project management and process improvement, Prentice-Hall, Inc. 1992.

Gubbi, J.; Buyya, R.; Marusic, S.; Palaniswami, M. (2013): Internet of Things (IoT): A vision, architectural elements, and future directions. In: Future Generation Computer Systems, Band 29 (2013) Nr. 7, S. 1645-1660.

Heger, C. (2015): An Approach for Guiding Developers to Performance and Scalability Solutions. Dissertation, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany 2015.

Heger, C.; Happe, J.; Farahbod, R. (2013): Automated root cause isolation of performance regressions during software development. Vorgestellt in: 4th International Conference on Performance Engineering, Prague, Czech Republic, S. 27-38.

Hevner, A.R.; March, S.T.; Park, J.; Ram, S. (2004): Design science in information systems research. In: Management Information Systems Quarterly, Band 28 (2004) Nr. 1, S. 75-105.

Holzschuher, F.; Peinl, R. (2013): Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. Vorgestellt in: Joint EDBT/ICDT Workshops, Genoa, Italy, S. 195-204.

Horký, V.; Haas, F.; Kotrč, J.; Lacina, M.; Tůma, P. (2013): Performance Regression Unit Testing: A Case Study. In: Computer Performance Engineering. Balsamo, M.S.; Knottenbelt, W.J.; Marin, A. (Hrsg.). Springer, Berlin, Heidelberg 2013, S. 149-163.

Horký, V.; Libič, P.; Marek, L.; Steinhauser, A.; Tůma, P. (2015): Utilizing Performance Unit Tests To Increase Performance Awareness. Vorgestellt in: 6th International Conference on Performance Engineering, Austin, Texas, USA, S. 289-300.

Höst, M.; Regnell, B.; Wohlin, C. (2000): Using students as subjects - a comparative study of students and professionals in lead-time impact assessment. In: Empirical Software Engineering, Band 5 (2000) Nr. 3, S. 201-214.

Hrischuk, C.E.; Woodside, C.M.; Rolia, J.A. (1999): Trace-based load characterization for generating performance software models. In: IEEE Transactions on Software Engineering, Band 25 (1999) Nr. 1, S. 122-135.

Humble, J.; Farley, D. (2010): Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader), Pearson Education 2010.

IEEE (1988): IEEE Guide to Software Configuration Management. *ANSI/IEEE Std 1042-1987*.

ISO (1998): Ergonomic requirements for office work with visual display terminals (VDTs) -- Part 11: Guidance on usability (ISO 9241-11:1998(E)). Genv: ISO.

ISO/IEC (2011): Systems and software engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and software quality models - ISO/IEC 25010:2011. ISO.

Israr, T.; Woodside, M.; Franks, G. (2007): Interaction tree algorithms to extract effective architecture and layered performance models from traces. In: Journal of Systems and Software, Band 80 (2007) Nr. 4, S. 474-492.

Jain, R. (1991): The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling, Wiley, New York, NY, USA 1991.

Jedlitschka, A.; Pfahl, D. (2005): Reporting guidelines for controlled experiments in software engineering. Vorgestellt in: International Symposium on Empirical Software Engineering, S. 95–104.

Jin, G.; Song, L.; Shi, X.; Scherpelz, J.; Lu, S. (2012): Understanding and detecting real-world performance bugs. In: ACM SIGPLAN Notices, Band 47 (2012) Nr. 6, S. 77-88.

Jonkers, H. (1994): Queueing models of parallel applications: The glamis methodology. Vorgestellt in: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, S. 123-138.

Kan, S.H. (2002): Metrics and Models in Software Quality Engineering, Addison-Wesley Longman Publishing Co., Inc. 2002.

Kappler, T.; Koziolek, H.; Krogmann, K.; Reussner, R.H. (2008): Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering. In: Software Engineering, Band 121 (2008), S. 140-154.

Keller, R.K.; Schauer, R.; Robitaille, S.; Pagé, P. (1999): Pattern-based reverse-engineering of design components. Vorgestellt in: 21st International Conference on Software Engineering, Los Angeles, California, USA, S. 226-235.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwin, J. (1997): Aspect-oriented programming. In: Object-Oriented Programming. Akşit, M.; Matsuoka, S. (Hrsg.). Springer, Berlin, Heidelberg 1997, S. 220-242.

Kieker Project (2015): Kieker User Guide (Research Report), 2015.

Killian, C.; Nagaraj, K.; Pervez, S.; Braud, R.; Anderson, J.W.; Jhala, R. (2010): Finding latent performance bugs in systems implementations. Vorgestellt in: 18th International Symposium on Foundations of Software Engineering, Santa Fe, New Mexico, USA, S. 17-26.

Kitchenham, B.; Fry, J.; Linkman, S. (2003): The case against cross-over designs in software engineering. Vorgestellt in: 11th International Workshop on Software Technology and Engineering Practice, S. 65-67.

Kitchenham, B.; Pfleeger, S.L. (1996): Software quality: the elusive target [special issues section]. In: IEEE Software, Band 13 (1996) Nr. 1, S. 12-21.

Kitchenham, B.A.; Pfleeger, S.L.; Pickard, L.M.; Jones, P.W.; Hoaglin, D.C.; Emam, K.E.; Rosenberg, J. (2002): Preliminary guidelines for empirical research in software engineering. In: IEEE Transactions on Software Engineering, Band 28 (2002) Nr. 8, S. 721-734.

Klatt, B.; Küster, M.; Krogmann, K.; Burkhardt, O. (2013): A change impact analysis case study: Replacing the input data model of SoMoX. Vorgestellt in: 15th Workshop Software Reengineering, Bad Honnef, Germany.

Kögel, M. (2011): Operation-based Model Evolution. Dissertation, Technischen Universität München, München, Deutschland 2011.

Kohavi, R.; Longbotham, R. (2007): Online Experiments: Lessons Learned. In: Computer, Band 40 (2007) Nr. 9, S. 103-105.

Koschke, R. (2005): Rekonstruktion von Software-Architekturen. In: Informatik - Forschung und Entwicklung, Band 19 (2005) Nr. 3, S. 127-140.

Kounev, S. (2006): Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. In: IEEE Transactions on Software Engineering, Band 32 (2006) Nr. 7, S. 486-502.

Koziolek, H. (2008): Parameter dependencies for reusable performance specifications of software components. Dissertation, Universität Oldenburg 2008.

Koziolek, H. (2010): Performance evaluation of component-based software systems: A survey. In: Performance Evaluation, Band 67 (2010) Nr. 8, S. 634-658.

Koziolek, H.; Schlich, B.; Becker, S.; Hauck, M. (2013): Performance and reliability prediction for evolving service-oriented software systems. In: Empirical Software Engineering, Band 18 (2013) Nr. 4, S. 746-790.

Kramer, M.E.; Durdik, Z.; Hauck, M.; Henss, J.; Küster, M.; Merkle, P.; Rentschler, A. (2012): Extending the Palladio component model using profiles and stereotypes. Vorgestellt in: Palladio Days, S. 7-15.

Krogmann, K. (2010): Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis. Dissertation, Karlsruhe Institute of Technology (KIT), Karlsruhe, Deutschland 2010.

Krogmann, K.; Kuperberg, M.; Reussner, R. (2010): Using Genetic Search for Reverse Engineering of Parametric Behavior Models for Performance Prediction. In: IEEE Transactions on Software Engineering, Band 36 (2010) Nr. 6, S. 865-877.

Krusche, S.; Bruegge, B. (2014): Model-based real-time synchronization. Vorgestellt in: International Workshop on Comparison and Versioning of Software Models.

Krusche, S.; Bruegge, B. (2017): CSEPM - A Continuous Software Engineering Process Metamodel. Vorgestellt in: 3rd International Workshop on Rapid Continuous Software Engineering, S. 2-8.

Kuryazov, D.; Winter, A. (2014): Representing Model Differences by Delta Operations. Vorgestellt in: EDOC Workshops, S. 211-220.

Kuryazov, D.; Winter, A. (2015): Collaborative Modeling Empowered By Modeling Deltas. Vorgestellt in: 3rd International Workshop on Document Changes: Modeling, Detection, Storage and Visualization, Lausanne, Switzerland, S. 1-6.

Laddad, R. (2009): AspectJ in Action: Enterprise AOP with Spring Applications, Manning Publications Co., Greenwich, CT, USA 2009.

Lau, K.-K. (2006): Software component models. Vorgestellt in: 28th International Conference on Software Engineering, Shanghai, China, S. 1081-1082.

Lee, D.; Cha, S.K.; Lee, A.H. (2012): A Performance Anomaly Detection and Analysis Framework for DBMS Development. In: IEEE Transactions on Knowledge and Data Engineering, Band 24 (2012) Nr. 8, S. 1345-1360.

Leonhardt, S.; Hettwer, B.; Hoor, J.; Langhammer, M. (2015): Integration of Existing Software Artifacts into a View- and Change-Driven Development Approach. Vorgestellt in: Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering, L'Aquila, Italy, S. 17-24.

Lewis, J.; Fowler, M. (2014): Microservices. https://martinfowler.com/articles/microservices.html#CharacteristicsOfAMicroserviceArchitecture, zugegriffen am 13.10.2019.

Lilja, D.J. (2000): Measuring Computer Performance, Cambridge University Press, Cambridge, UK 2000.

Linthicum, D.S. (2017): Connecting fog and cloud computing. In: IEEE Cloud Computing, Band 4 (2017) Nr. 2, S. 18-20.

Lochmann, K. (2014): Defining and Assessing Software Quality by Quality Models. Dissertation, Technische Universität München 2014.

McCall, J.A.; Richards, P.K.; Walters, G.F. (1977): Factors in software quality. volume i. concepts and definitions of software quality (RADC-TR-77-369, Volume 1). GENERAL ELECTRIC CO SUNNYVALE CA, 1977.

McIlroy, M.D.; Buxton, J.; Naur, P.; Randell, B. (1968): Mass-produced software components. Vorgestellt in: NATO Conference Sofware Engineering, S. 138–155.

Menascé, D.A.; Almeida, V. (2002): Capacity Planning for Web Services: metrics, models, and methods, Prentice Hall, Upper Saddle River, NJ, USA 2002.

Menascé, D.A.; Dowdy, L.W.; Almeida, V.A.F. (2004): Performance by Design: Computer Capacity Planning By Example, Prentice Hall, Upper Saddle River, NJ, USA 2004.

Merkle, P.; Henss, J. (2011): EventSim – an event-driven Palladio software architecture simulator. Vorgestellt in: Palladio Days, S. 15-22.

Miller, B.P.; Callaghan, M.D.; Cargille, J.M.; Hollingsworth, J.K.; Irvin, R.B.; Karavanic, K.L.; Kunchithapadam, K.; Newhall, T. (1995): The Paradyn parallel performance measurement tool. In: Computer, Band 28 (1995) Nr. 11, S. 37-46.

Minelli, R.; Mocci, A.; Lanza, M. (2015): I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. Vorgestellt in: 23rd International Conference on Program Comprehension, S. 25-35.

Mitchell, B.S.; Mancoridis, S. (2001): Comparing the decompositions produced by software clustering algorithms using similarity measurements. Vorgestellt in: International Conference on Software Maintenance, S. 744-753.

Montesi, F.; Weber, J. (2016): Circuit breakers, discovery, and API gateways in microservices. In: arXiv:1609.05830v2, (2016).

Moore, B.; Dean, D.; Gerber, A.; Wagenknecht, G.; Vanderheyden, P. (2004): Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework, IBM 2004.

Mueller, H.; Gogouvitis, S.V.; Seitz, A.; Bruegge, B. (2017): Seamless Computing for Industrial Systems Spanning Cloud and Edge. Vorgestellt in: International Conference on High Performance Computing & Simulation, Genoa, Italy, S. 209-216.

Müller-Hofmann, F.; Hiller, M.; Wanner, G. (2015): Programmierung von verteilten Systemen und Webanwendungen mit Java EE, Springer Vieweg 2015.

Müller, H.A.; Orgun, M.A.; Tilley, S.R.; Uhl, J.S. (1993): A reverse-engineering approach to subsystem structure identification. In: Journal of Software Maintenance: Research and Practice, Band 5 (1993) Nr. 4, S. 181-204.

Murta, L.; Oliveira, H.; Dantas, C.; Lopes, L.G.; Werner, C. (2007): Odyssey-SCM: An integrated software configuration management infrastructure for UML models. In: Science of Computer Programming, Band 65 (2007) Nr. 3, S. 249-274.

Nadareishvili, I.; Mitra, R.; McLarty, M.; Amundsen, M. (2016): Microservice architecture: aligning principles, practices, and culture, O'Reilly Media 2016.

Newman, S. (2015): Building microservices: designing fine-grained systems, O'Reilly Media 2015.

Nistor, A.; Jiang, T.; Tan, L. (2013a): Discovering, reporting, and fixing performance bugs. Vorgestellt in: 10th Working Conference on Mining Software Repositories, S. 237-246.

Nistor, A.; Song, L.; Marinov, D.; Lu, S. (2013b): Toddler: detecting performance problems via similar memory-access patterns. Vorgestellt in: International Conference on Software Engineering, San Francisco, CA, USA, S. 562-571.

O'Brien, L.; Stoermer, C.; Verhoef, C. (2002): Software architecture reconstruction: Practice needs and current approaches (Technical Report No. CMU/SEI-2002-TR-024). Carnegie Mellon University Software Engineering Institute, 2002.

OMG (2015): OMG Unified Modeling Language™ (OMG UML). OMG.

Panda, D.; Rahman, R.; Cuprak, R.; Remijan, M. (2014): EJB 3 in Action, Manning Publications Co., Shelter Island, NY, USA 2014.

Parsons, T. (2007): Automatic detection of performance design and deployment antipatterns in component based enterprise systems. Dissertation, University College Dublin 2007.

Parsons, T.; Mos, A.; Trofin, M.; Gschwind, T.; Murphy, J. (2008): Extracting Interactions in Component-Based Systems. In: IEEE Transactions on Software Engineering, Band 34 (2008) Nr. 6, S. 783-799.

Peffers, K.; Tuunanen, T.; Gengler, C.E.; Rossi, M.; Hui, W.; Virtanen, V.; Bragge, J. (2006): The design science research process: a model for producing and presenting information systems research. Vorgestellt in: 1st International Conference on Design Science Research in Information Systems and Technology, S. 83-106.

Ponzanelli, L.; Bavota, G.; Penta, M.D.; Oliveto, R.; Lanza, M. (2014): Mining StackOverflow to turn the IDE into a self-confident programming prompter. Vorgestellt in: 11th Working Conference on Mining Software Repositories, Hyderabad, India, S. 102-111.

Pradel, M.; Huggler, M.; Gross, T.R. (2014): Performance regression testing of concurrent classes. Vorgestellt in: International Symposium on Software Testing and Analysis, San Jose, CA, USA, S. 13-25.

Prasanna, D.R. (2009): Dependency injection, Manning Publications Co., Greenwich, CT, USA 2009.

Quante, J. (2008): Do Dynamic Object Process Graphs Support Program Understanding? - A Controlled Experiment. Vorgestellt in: 16th International Conference on Program Comprehension, S. 73-82.

Richards, M. (2015): Microservices vs. service-oriented architecture, O'Reilly Media, Sebastopol, CA 2015.

Robillard, M.P.; Coelho, W.; Murphy, G.C. (2004): How effective developers investigate source code: an exploratory study. In: IEEE Transactions on Software Engineering, Band 30 (2004) Nr. 12, S. 889-903.

Rohr, M.; Hoorn, A.v.; Hasselbring, W.; Lübcke, M.; Alekseev, S. (2010): Workload-intensity-sensitive timing behavior analysis for distributed multi-user software systems. Vorgestellt in: 1st International Conference on Performance Engineering, San Jose, California, USA, S. 87-92.

Ross, K.D. (2006): Towards an automatic complexity analysis for generic programs. Vorgestellt in: Workshop on Generic Programming, Portland, Oregon, USA, S. 87-95.

Rothlisberger, D.; Harry, M.; Binder, W.; Moret, P.; Ansaloni, D.; Villazon, A.; Nierstrasz, O. (2012): Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks. In: IEEE Transactions on Software Engineering, Band 38 (2012) Nr. 3, S. 579-591.

Runeson, P. (2003): Using students as experiment subjects–an analysis on graduate and freshmen student data. Vorgestellt in: 7th International Conference on Empirical Assessment in Software Engineering, S. 95-102.

Saks, K. (2009): JSR 318: Enterprise JavaBeans™, Version 3.1 - EJB Core Contracts and Requirements (Specification JSR-000318). Sun Microsystems, 2009.

Sanchez, H.; Robbes, R.; Gonzalez, V.M. (2015): An empirical study of work fragmentation in software evolution tasks. Vorgestellt in: 22nd International Conference on Software Analysis, Evolution, and Reengineering, S. 251-260.

SAP (2018): CIO Guide: Process and Data Integration in Hybrid Landscapes. https://www.sap.com/documents/2018/06/de3238a0-077d-0010-87a3-c30de2ffd8ff.html, zugegriffen am 13.10.2019.2019.

Sartipi, K. (2003): Software architecture recovery based on pattern matching. Vorgestellt in: International Conference on Software Maintenance, S. 293-296.

Satyanarayanan, M. (2017): The Emergence of Edge Computing. In: Computer, Band 50 (2017) Nr. 1, S. 30-39.

Satyanarayanan, M.; Bahl, V.; Caceres, R.; Davies, N. (2009): The case for vm-based cloudlets in mobile computing. In: IEEE Pervasive Computing, (2009).

Sauro, J.; Lewis, J.R. (2012): Quantifying the User Experience: Practical Statistics for User Research, Elsevier/Morgan Kaufmann, Waltham, MA, USA 2012.

Seitz, A.; Buchinger, D.; Bruegge, B. (2018): The Conjunction of Fog Computing and the Industrial Internet of Things - An Applied Approach. Vorgestellt in: International Conference on Pervasive Computing and Communications Workshops, Athens, Greece, S. 812-817.

Seitz, A.H.N. (2019): An Architectural Style for Fog Computing: Formalization and Application. Dissertation, Technische Universität München 2019.

Sharp, J.H.; Ryan, S.D. (2010): A theoretical framework of component-based software development phases. In: ACM SIGMIS Database, Band 41 (2010) Nr. 1, S. 56-75.

Smith, C.U. (1993): Software performance engineering. In: Performance Evaluation of Computer and Communication Systems: Joint Tutorial Papers of Performance '93 and Sigmetrics '93. Donatiello, L.; Nelson, R. (Hrsg.). Springer, Berlin, Heidelberg 1993, S. 509-536.

Smith, C.U.; Williams, L.G. (2000): Software performance antipatterns. Vorgestellt in: 2nd International Workshop on Software and Performance, Ottawa, Ontario, Canada, S. 127-136.

Smith, C.U.; Williams, L.G. (2002): New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot. Vorgestellt in: Computer Measurement Group Conference, S. 667-674.

Smith, C.U.; Williams, L.G. (2003): More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot. Vorgestellt in: Computer Measurement Group Conference, S. 717-725.

Snipes, W.; Murphy-Hill, E.; Fritz, T.; Vakilian, M.; Damevski, K.; Nair, A.R.; Shepherd, D. (2015): Chapter 5 - A Practical Guide to Analyzing IDE Usage Data A2 - Bird, Christian. In: The Art and Science of Analyzing Software Data. Menzies, T.; Zimmermann, T. (Hrsg.). Morgan Kaufmann, Boston 2015, S. 85-138.

SPEC (2014): SPECjEnterprise2010. http://www.spec.org/jEnterprise2010, zugegriffen am 09.05.2016.

Stahl, T.; Voelter, M.; Bettin, J.; Haase, A.; Helsen, S. (2006): Model-Driven Software Development: Technology, Engineering, Management, John Wiley & Sons, West Sussex, England 2006.

Stecklein, J.M.; Dabney, J.; Dick, B.; Haskins, B.; Lovell, R.; Moroney, G. (2004): Error cost escalation through the project life cycle. NASA Johnson Space Center, 2004.

Stevens, S.S. (1946): On the Theory of Scales of Measurement. In: Science, Band 103 (1946) Nr. 2684, S. 677-680.

Storey, M.-A.D.; Čubranić, D.; German, D.M. (2005): On the use of visualization to support awareness of human activities in software development: a survey and a framework. Vorgestellt in: Symposium on Software Visualization, St. Louis, MO, USA, S. 193-202.

Strein, D.; Lincke, R.; Lundberg, J.; Löwe, W. (2007): An Extensible Meta-Model for Program Analysis. In: IEEE Transactions on Software Engineering, Band 33 (2007) Nr. 9, S. 592-607.

Stuckenholz, A. (2005): Component evolution and versioning state of the art. In: ACM SIGSOFT Software Engineering Notes, Band 30 (2005) Nr. 1, S. 7.

Svahnberg, M.; Aurum, A.; Wohlin, C. (2008): Using students as subjects - an empirical evaluation. Vorgestellt in: 2nd International Symposium on Empirical Software Engineering and Measurement, S. 288-290.

Svorobej, S.; Byrne, J.; Liston, P.; Byrne, P.; Stier, C.; Groenda, H.; Papazachos, Z.; Nikolopoulos, D.S. (2015): Towards automated data-driven model creation for cloud computing simulation. Vorgestellt in: 8th International Conference on Simulation Tools and Techniques, S. 248-255.

Szyperski, C. (2002): Component Software: Beyond Object-Oriented Programming, Addison-Wesley Longman Publishing Co., Inc., London, UK 2002.

Tam, J.; Greenberg, S. (2006): A framework for asynchronous change awareness in collaborative documents and workspaces. In: International Journal of Human-Computer Studies, Band 64 (2006) Nr. 7, S. 583-598.

Tate, B.; Clark, M.; Lee, B.; Linskey, P. (2003): Bitter EJB, Manning Publications Co., Greenwich, CT, USA 2003.

Tate, B.A. (2002): Bitter Java, Manning Publications Co., Greenwich, CT, USA 2002.

Tichy, W.F. (1984): RCS - a system for version control (Report 84-474). Department of Computer Sciences, Purdue University, 1984.

Tichy, W.F. (1988): Tools for Software Configuration Management. Vorgestellt in: International Workshop on Software Version and Configuration Control, S. 1-20.

Tůma, P. (2014): Performance awareness: keynote abstract. Vorgestellt in: 5th International Conference on Performance Engineering, Dublin, Ireland, S. 135-136.

van Hoorn, A.; Rohr, M.; Hasselbring, W.; Waller, J.; Ehlers, J.; Frey, S.; Kieselhorst, D. (2009): Continuous Monitoring of Software Services: Design and Application of the Kieker Framework (Research Report). Kiel University, 2009.

van Hoorn, A.; Waller, J.; Hasselbring, W. (2012): Kieker: a framework for application performance monitoring and dynamic software analysis. Vorgestellt in: 3rd International Conference on Performance Engineering, Boston, Massachusetts, USA, S. 247-248.

van Steen, M.; Tanenbaum, A.S. (2017): Distributed Systems, CreateSpace Independent Publishing Platform 2017.

Varshney, P.; Simmhan, Y. (2017): Demystifying Fog Computing: Characterizing Architectures, Applications and Abstractions. Vorgestellt in: 1st International Conference on Fog and Edge Computing, Madrid, Spain, S. 115-124.

Vetter, J. (2000): Performance analysis of distributed applications using automatic classification of communication inefficiencies. Vorgestellt in: 14th International Conference on Supercomputing, Santa Fe, New Mexico, USA, S. 245-254.

Vögele, C.; Brunnert, A.; Danciu, A.; Tertilt, D.; Krcmar, H. (2014): Using performance models to support load testing in a large SOA environment. Vorgestellt in: 3rd International Workshop on Large Scale Testing, Dublin, Ireland, S. 5-6.

W3C (2004): Web Services Glossary - W3C Working Group Note. http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/, zugegriffen am 13.10.2019.

Waller, J.; Ehmke, N.C.; Hasselbring, W. (2015): Including Performance Benchmarks into Continuous Integration to Enable DevOps. In: ACM SIGSOFT Software Engineering Notes, Band 40 (2015) Nr. 2, S. 1-4.

Waller, J.; Hasselbring, W. (2013): A Benchmark Engineering Methodology to Measure the Overhead of Application-Level Monitoring. Vorgestellt in: Symposium on Software Performance: Joint Kieker/Palladio Days, Karlsruhe, Germany, S. 59-68.

Weiss, C.; Westermann, D.; Heger, C.; Moser, M. (2013): Systematic performance evaluation based on tailored benchmark applications. Vorgestellt in: 4th International Conference on Performance Engineering, Prague, Czech Republic, S. 411-420.

Wert, A. (2015): Performance Problem Diagnostics by Systematic Experimentation. Dissertation, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany 2015.

Wert, A.; Happe, J.; Westermann, D. (2012): Integrating software performance curves with the palladio component model. Vorgestellt in: 3rd International Conference on Performance Engineering, Boston, Massachusetts, USA, S. 283-286.

Westermann, D. (2012): A generic methodology to derive domain-specific performance feedback for developers. Vorgestellt in: 34th International Conference on Software Engineering, Zurich, Switzerland, S. 1527-1530.

Westermann, D. (2014): Deriving Goal-oriented Performance Models by Systematic Experimentation. Dissertation, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany 2014.

Westermann, D.; Krebs, R.; Happe, J. (2011): Efficient Experiment Selection in Automated Software Performance Evaluations. In: Computer Performance Engineering. Thomas, N. (Hrsg.). Springer, Berlin, Heidelberg 2011, S. 325-339.

Wettel, R.; Lanza, M.; Robbes, R. (2011): Software systems as cities: a controlled experiment. Vorgestellt in: 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA, S. 551-560.

Weyuker, E.J.; Vokolos, F.I. (2000): Experience with performance testing of software systems: issues, an approach, and case study. In: IEEE Transactions on Software Engineering, Band 26 (2000) Nr. 12, S. 1147-1156.

Whalen, M.W.; Godefroid, P.; Mariani, L.; Polini, A.; Tillmann, N.; Visser, W. (2010): FITE: future integrated testing environment. Vorgestellt in: Workshop on Future of Software Engineering Research, Santa Fe, New Mexico, USA, S. 401-406.

Wöbker, C.; Seitz, A.; Mueller, H.; Bruegge, B. (2018): Fogernetes: Deployment and management of fog computing applications. Vorgestellt in: Network Operations and Management Symposium, Taipei, Taiwan, S. 1-7.

Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.; Regnell, B.; Wesslén, A. (2012): Experimentation in Software Engineering, Springer, Berlin, Heidelberg 2012.

Woodside, M. (2013): Tutorial introduction to layered modeling of software performance. Carleton University, Ottawa, Canada, 2013.

Woodside, M.; Franks, G.; Petriu, D.C. (2007): The Future of Software Performance Engineering. Vorgestellt in: Future of Software Engineering, S. 171-187.

Woodside, M.; Hrischuk, C.; Selic, B.; Bayarov, S. (2001): Automated performance modeling of software generated by a design environment. In: Performance Evaluation, Band 45 (2001) Nr. 2–3, S. 107-123.

Woodside, M.; Neilson, J.E.; Petriu, D.C.; Majumdar, S. (1995): The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. In: IEEE Transactions on Computers, Band 44 (1995) Nr. 1, S. 20-34.

Yan, D.; Xu, G.; Rountev, A. (2012): Uncovering performance problems in Java applications with reference propagation profiling. Vorgestellt in: 34th International Conference on Software Engineering, Zurich, Switzerland, S. 134-144.

Yi, S.; Li, C.; Li, Q. (2015a): A Survey of Fog Computing: Concepts, Applications and Issues. *Workshop on Mobile Big Data* (S. 37-42). Hangzhou, China: ACM.

Yi, S.; Qin, Z.; Li, Q. (2015b): Security and Privacy Issues of Fog Computing: A Survey. In: Wireless Algorithms, Systems, and Applications. Xu, K.; Zhu, H. (Hrsg.). Springer International Publishing, Cham 2015b, S. 685-695.

Zheng, T.; Woodside, C.M.; Litoiu, M. (2008): Performance Model Estimation and Tracking Using Optimal Filters. In: IEEE Transactions on Software Engineering, Band 34 (2008) Nr. 3, S. 391-406.

Anhang A Anleitung zur Durchführung des Experiments

Anhang A-1 Anleitung für Probanden der Testgruppe

Experiment procedure:

- Read these instructions carefully (10 Minutes)
- Perform the task described below using eclipse (40 Minutes)
- Close eclipse, you have completed the experiment!

Scenario

You are part of a team developing a web application called *cargo-tracker*. You just finished implementing the first version of the *BatchProcessingBean* (*net.java.cargotracker.batch. BatchProcessingBean.java*). This component is called to reschedule overdue shipments and assign itineraries to new shipments.

A non-functional requirement for the application is that the response times of user actions should be as low as possible, ideally below one second.

Your Eclipse IDE provides a plugin which supports evaluating the response time of method implementations and identifying the most expensive calls in your code.

About the application

The cargo-tracker web application is an end-to-end system for keeping track of shipping cargo. It has several interfaces described in the following sections.

The tracking interface lets you track the status of cargo and is intended for the general public. Try entering a tracking ID like ABC123 (the application is pre-populated with some sample data).

The administrative interface is intended for the shipping company that manages cargo. The landing page of the interface is a dashboard providing an overall view of registered cargo. The dashboard will update automatically when cargo is handled (described below). You can book cargo using the booking interface. One cargo is booked, you can route it. When you initiate a routing request, the system will determine routes that might work for the cargo. Once you select a route, the cargo will be ready to process handling events at the port. You can also change the destination for cargo if needed or track cargo.

The Incident Logging interface is intended for port personnel registering what happened to cargo. The interface is primarily intended for mobile devices, but you can use it via a desktop browser. The interface is accessible at: http://localhost:8080/cargo-tracker/incident-logger/. Generally speaking cargo goes though these events:

- It's received at the origin port
- It's loaded and unloaded onto voyages on its itinerary

- It's claimed at its destination port
- It may go through customs at arbitrary points

You can access the application by calling http://localhost:8080/cargo-tracker within your browser:

To execute the *BatchProcessingBean* go to *Administration Interface:*

File Edit View History Bookmarks Tools Help

Cargo Dashboard

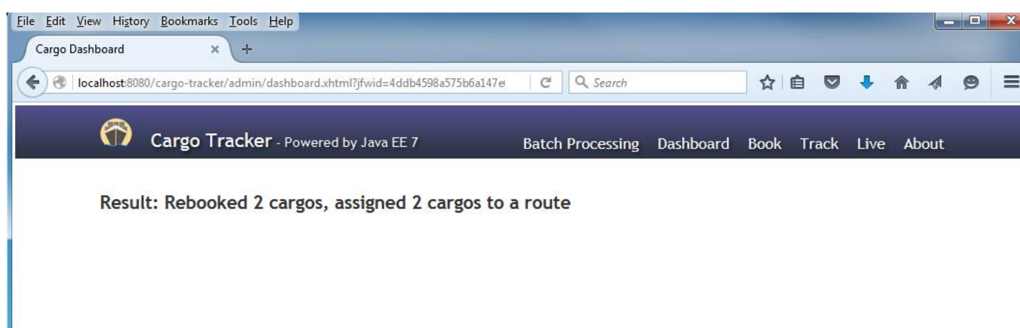
localhost:8080/cargo-tracker/admin/dashboard.xhtml

Cargo Tracker - Powered by Java EE 7 Batch Processing Dashboard Book Track Live About

Routed

Id	Origin	Destination	Last Known Location	Status	Deadline
ABC123	Hong Kong CNHKG	Helsinki FIHEL	New York USNYC	IN_PORT	03/15/2015 12:00 AM MEZ
JKL567	Hangzhou CNHGH	Stockholm SESTO	New York USNYC	ONBOARD_CARRIER	03/18/2015 12:00 AM MEZ

Then go to *Batch Processing:*



How to run the application

1. Start: Run C:\Experiment\Run_cargo-tracker.bat

2. Stop: Within the console, press CTRL-C

File structure overview:

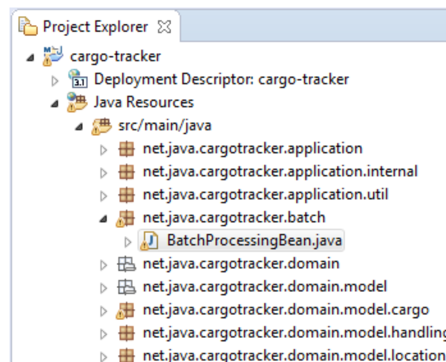
- C:\Experiment\
    - eclipse\
        - eclipse.exe -> Run this executable to start your eclipse IDE
    - Run_cargo-tracker.bat -> Run this executable to start the cargo-tracker application

How to use the performance evaluation tool

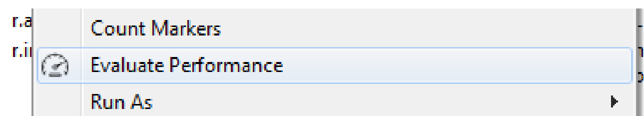
The Performance Evaluation plugin parses the source code of the selected java class (and all subsequent classes which are called by its methods) and estimates its response time. Methods and calls exceeding a predefined threshold are highlighted in the code editor. A tool tip description displays the expected response time in milliseconds.

Execute the following steps to obtain an estimation of the response time:

1. Select the corresponding java class in the *Project Explorer*



2. Right click the java class and select *Evaluate Performance*:

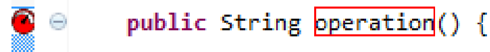


3. The evaluation progress is displayed in the lower right corner (can take up to 2 minutes):

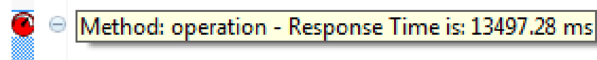
4. Eclipse can be used in the meantime for other activities. However, the Java code should not be edited while it is analyzed. Otherwise, the evaluation results are obsolete.

5. Results of the performance evaluation are displayed within the source code of the analyzed class and all called functions:

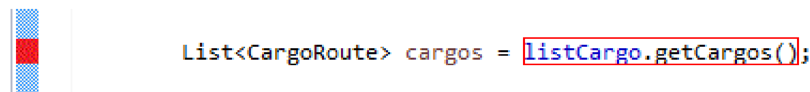
Marker: Estimation for a method of the selected class exceeds the error threshold:



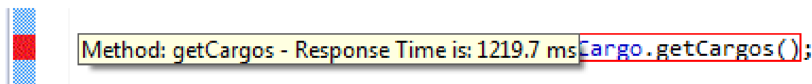
Tool tip: Estimation for a method of the selected class exceeds the error threshold:



Marker: Response time of a method call exceeds the error threshold:



Tool tip: Response time of a method call exceeds the error threshold:



6. After improving the source code start a new evaluation as described in step 1.

Task

Review the implementations of the *BatchProcessingBean* and its associated components to identify opportunities to improve their response time. Apply the corresponding changes to the current implementation. Run the application on your local machine to ensure the functionality of the application is not altered.

Use the Performance Evaluation tool to obtain a response time estimation for the *BatchProcessingBean* and to identify potential improvements. Start this task with executing the tool, as described below.

Comments

- You are only allowed to edit .java files
- Within your development environment files and folders might be created by background processes. Please do not interfere in these operations by deleting files.
- Please avoid searching the internet for clues during the experiment.

Anhang A-2 Anleitung für Probanden der Kontrollgruppe

Experiment procedure:

- Read these instructions carefully (10 Minutes)
- Perform the task described below using eclipse (40 Minutes)
- Close eclipse, you have completed the experiment!

Scenario

You are part of a team developing a web application called *cargo-tracker*. You just finished implementing the first version of the *BatchProcessingBean* (*net.java.cargotracker.batch. BatchProcessingBean.java*). This component is called to reschedule overdue shipments and assign itineraries to new shipments.

A non-functional requirement for the application is that the response times of user actions should be as low as possible, ideally below one second.

About the application

The cargo-tracker web application is an end-to-end system for keeping track of shipping cargo. It has several interfaces described in the following sections.

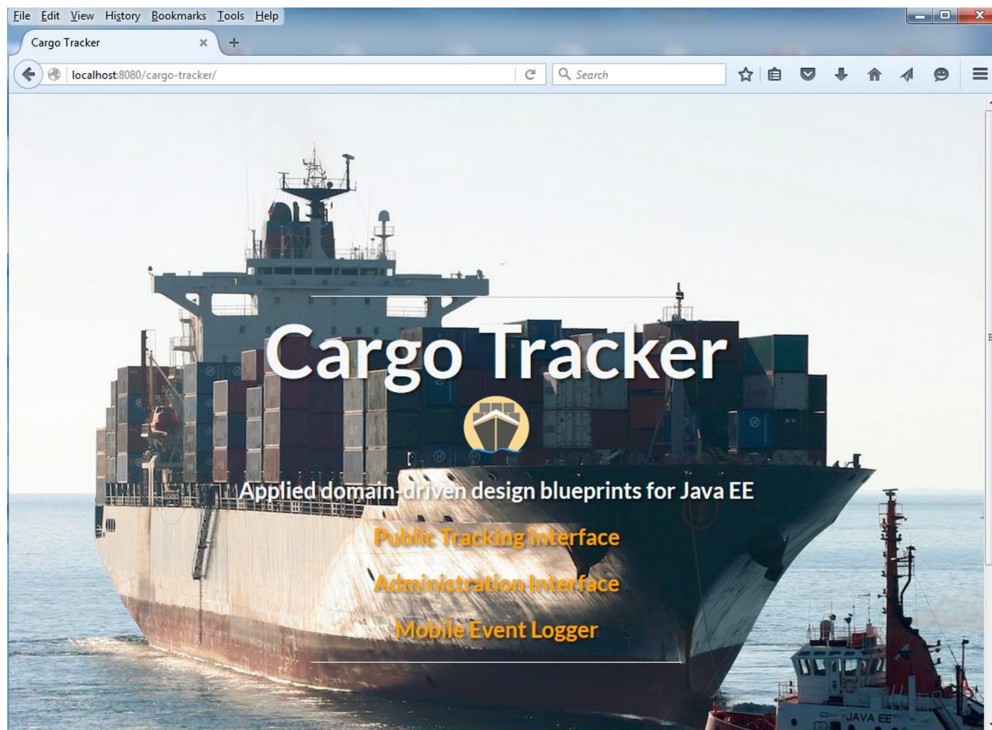
The tracking interface lets you track the status of cargo and is intended for the general public. Try entering a tracking ID like ABC123 (the application is pre-populated with some sample data).

The administrative interface is intended for the shipping company that manages cargo. The landing page of the interface is a dashboard providing an overall view of registered cargo. The dashboard will update automatically when cargo is handled (described below). You can book cargo using the booking interface. One cargo is booked, you can route it. When you initiate a routing request, the system will determine routes that might work for the cargo. Once you select a route, the cargo will be ready to process handling events at the port. You can also change the destination for cargo if needed or track cargo.

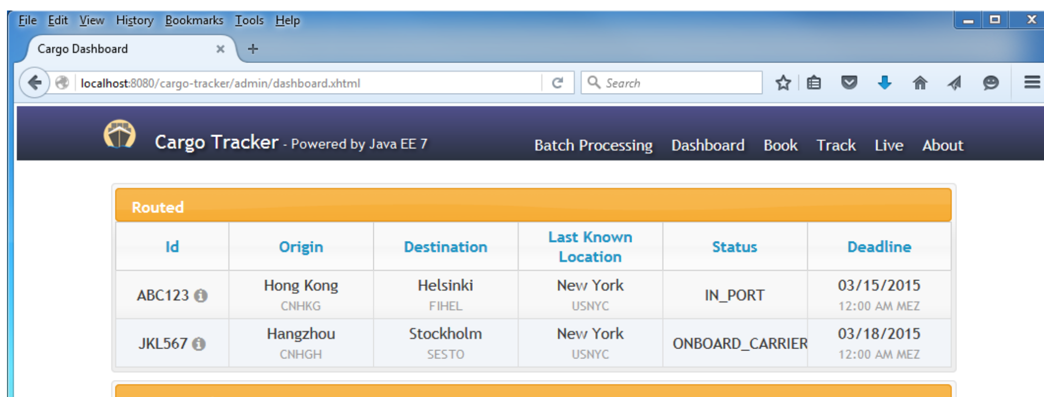
The Incident Logging interface is intended for port personnel registering what happened to cargo. The interface is primarily intended for mobile devices, but you can use it via a desktop browser. The interface is accessible at: http://localhost:8080/cargo-tracker/incident-logger/. Generally speaking cargo goes though these events:

- It's received at the origin port
- It's loaded and unloaded onto voyages on its itinerary
- It's claimed at its destination port
- It may go through customs at arbitrary points

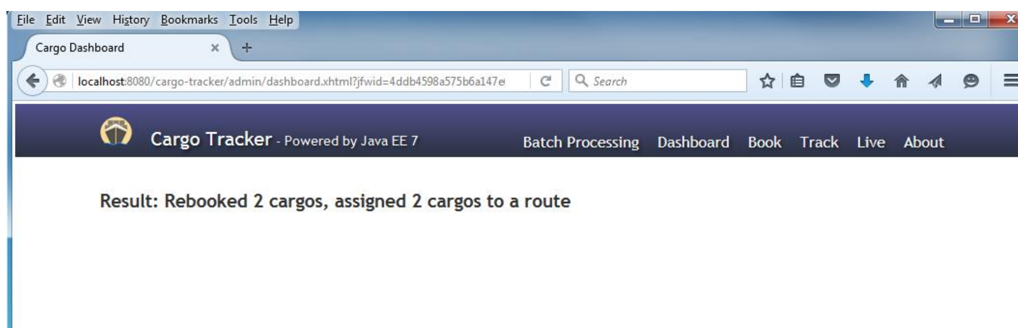
You can access the application by calling http://localhost:8080/cargo-tracker within your browser:



To execute the *BatchProcessingBean* go to *Administration Interface:*



Then go to *Batch Processing:*



How to run the application

1. Start: Run C:\Experiment\Run_cargo-tracker.bat

2. Stop: Within the console, press CTRL-C

File structure overview:

- C:\Experiment\
    - eclipse\
        - eclipse.exe -> Run this executable to start your eclipse IDE
    - Run_cargo-tracker.bat -> Run this executable to start the cargo-tracker application

Task

Review the implementations of the *BatchProcessingBean* and its associated components to identify opportunities to improve their response time. Apply the corresponding changes to the current implementation. Run the application on your local machine to ensure the functionality of the application is not altered.

Comments

- You are only allowed to edit .java files
- Within your development environment files and folders might be created by background processes. Please do not interfere in these operations by deleting files.
- Please avoid searching the internet for clues during the experiment.

Anhang B Fragebogen für Versuchspersonen

Anhang B-1 Fragebogen für Versuchspersonen der Testgruppe

Participant ID:

General Information

Highest degree or level of school:
- High school graduate
- Bachelor’s degree
- Master’s degree
- Doctorate degree

Employment status:
- Student
- Employed researcher
- Employed practitioner
- Self-employed

Programming skills

General knowledge in programming:
- Only 1-2 courses
- 3 ore more courses, no industrial experience
- A few courses and some industrial experience
- More than three courses and more than 1 year industrial experience

Knowledge in Java:
- No prior knowledge
- Read a book or followed a course
- Some industrial experience (less than 6 months)
- Industrial experience

Knowledge in Java Enterprise Edition:

- No prior knowledge
- Read a book or followed a course
- Some industrial experience (less than 6 months)
- Industrial experience

Knowledge in software performance management:

- No prior knowledge
- Read a book or followed a course
- Some industrial experience (less than 6 months)
- Industrial experience

Tool/API Skills

Knowledge of the eclipse IDE:

- No prior knowledge
- Read a book or followed a course
- Some industrial experience (less than 6 months)
- Industrial experience

Knowledge of the Java Persistence API (JPA):

- No prior knowledge
- Read a book or followed a course
- Some industrial experience (less than 6 months)
- Industrial experience

Knowledge of the cargo-tracker application:

- No prior knowledge
- Informed about the application
- Used the application
- Developed the application

Tool Usability

Rate the performance evaluation tool (not eclipse).

Strongly disagree 1 2 3 4 5 6 7 Strongly agree

		1	2	3	4	5	6	7
1	I think that I would like to use this tool.	○	○	○	○	○	○	○
2	I found the tool unnecessarily complex.	○	○	○	○	○	○	○
3	I thought the tool was easy to use.	○	○	○	○	○	○	○
4	I think that I would need the support of a technical person to be able to use this tool.	○	○	○	○	○	○	○
5	I found the various functions in the tool were well integrated.	○	○	○	○	○	○	○
6	I thought there was too much inconsistency in this tool.	○	○	○	○	○	○	○
7	I would imagine that most people would learn to use this tool very quickly.	○	○	○	○	○	○	○
8	I found the tool very cumbersome to use.	○	○	○	○	○	○	○
9	I felt very confident using the tool.	○	○	○	○	○	○	○
10	I needed to learn a lot of things before I could get going with this tool.	○	○	○	○	○	○	○

Comments on the usability and/or functionality of the performance evaluation tool (not eclipse):

Anhang B-2 Fragebogen für Versuchspersonen der Kontrollgruppe

Participant ID:

General Information

Highest degree or level of school:

- High school graduate
- Bachelor’s degree
- Master’s degree
- Doctorate degree

Employment status:

- Student
- Employed researcher
- Employed practitioner
- Self-employed

Programming skills

General knowledge in programming:

- Only 1-2 courses
- 3 ore more courses, no industrial experience
- A few courses and some industrial experience
- More than three courses and more than 1 year industrial experience

Knowledge in Java:

- No prior knowledge
- Read a book or followed a course
- Some industrial experience (less than 6 months)
- Industrial experience

Knowledge in Java Enterprise Edition:

- No prior knowledge
- Read a book or followed a course
- Some industrial experience (less than 6 months)
- Industrial experience

Knowledge in software performance management:

- No prior knowledge
- Read a book or followed a course
- Some industrial experience (less than 6 months)
- Industrial experience

Tool/API Skills

Knowledge of the eclipse IDE:

- No prior knowledge
- Read a book or followed a course
- Some industrial experience (less than 6 months)
- Industrial experience

Knowledge of the Java Persistence API (JPA):

- No prior knowledge
- Read a book or followed a course
- Some industrial experience (less than 6 months)
- Industrial experience

Knowledge of the cargo-tracker application:

- No prior knowledge
- Informed about the application
- Used the application
- Developed the application

Tool Usability

Rate the eclipse IDE.

Strongly disagree 1 2 3 4 5 6 7 Strongly agree

		1	2	3	4	5	6	7
1	I think that I would like to use this tool.	○	○	○	○	○	○	○
2	I found the tool unnecessarily complex.	○	○	○	○	○	○	○
3	I thought the tool was easy to use.	○	○	○	○	○	○	○
4	I think that I would need the support of a technical person to be able to use this tool.	○	○	○	○	○	○	○
5	I found the various functions in the tool were well integrated.	○	○	○	○	○	○	○
6	I thought there was too much inconsistency in this tool.	○	○	○	○	○	○	○
7	I would imagine that most people would learn to use this tool very quickly.	○	○	○	○	○	○	○
8	I found the tool very cumbersome to use.	○	○	○	○	○	○	○
9	I felt very confident using the tool.	○	○	○	○	○	○	○
10	I needed to learn a lot of things before I could get going with this tool.	○	○	○	○	○	○	○

Comments on the usability and/or functionality of the eclipse IDE: