UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Universität Paderborn
Zukunftsmeile 1
D-33102 Paderborn

PBlaman: kontraktbasierte Performance-Blame-Analysis

Schriftliche Arbeit
zur Erlangung des Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

betreut von
Prof. Dr. Gregor Engels
Prof. Dr.-Ing. Steffen Becker

vorgelegt von
Frank Brüseke

Paderborn, November 2014

Frank Brüseke
PBlaman: kontraktbasierte Performance-Blame-Analysis, Universität Paderborn, November 2014

Promotionskommission

Prof. Dr. Gregor Engels, Universität Paderborn (Vorsitzender)

Prof. Dr. Steffen Becker, Technische Universität Chemnitz

Prof. Dr. Uwe Kastens, Universität Paderborn

Prof. Dr. Christian Plessl, Universität Paderborn

Dr. Stefan Sauer, Universität Paderborn

Gutachter

Prof. Dr. Gregor Engels, Universität Paderborn

Prof. Dr. Steffen Becker, Technische Universität Chemnitz

Tag der Verteidigung: 12.12.2014

Für Andrea

Danke

Danke an meine Doktorväter Gregor Engels und Steffen Becker. Ihr habt mich zu jeder Zeit unterstützt. Euer Feedback hat mir immer sehr geholfen und Ihr habt mir so einiges Wichtige beigebracht. Ihr habt zudem auch den Rahmen geschaffen, der es mir, und wohl auch meinen Kollegen, ermöglicht überhaupt so gut zu arbeiten.

Danke an meine Familie. Andrea Du hast mich zu jeder Zeit unterstützt und mir den Rücken frei gehalten, wenn es sein musste. Aber Du hast mir auch zur rechten Zeit einen Tritt in den Hintern verpasst. Jan Du hast mich motiviert und hast akzeptiert, dass ich nicht häufiger mit dir spielen konnte.

Danke an meine Eltern für Eure immerwährende Unterstützung.

Danke an meine Kollegen. Euer Feedback war stets sehr wertvoll. Diskussionen mit Euch haben mir viel Freude bereitet und ich habe dabei einiges gelernt. Ich danke Euch auch für so manche vergnügliche Kaffee- bzw. Mittagspause.

Danke an meinen Arbeitgeber die Alfons Venjakob GmbH & Co KG. Sie haben mir in der Zeit, in der ich bei Ihnen bin, die nötige Flexibilität und den nötigen Freiraum eingeräumt.

Zusammenfassung

Beim performance-getriebenen Software-Engineering wird die Performance eines Systems vor seiner Implementierung mithilfe von Modellen überprüft. Nach der Implementierung wird die Performance des Systems durch Performance-Tests validiert. Wenn beim Test eines komponentenbasierten Systems auffällt, dass eine Performance-Anforderung nicht eingehalten wird, muss der Systemarchitekt herausfinden, ob die Komponenten fehlerhaft sind oder die Komponentenverwendung zu Fehlern führt. Diese Aufgabe wird *Performance-Blame-Analysis* genannt.

Vorhandene Performance-Analyse-Ansätze sind für die Performance-Blame-Analysis wenig geeignet, weil entweder erwartete Performancewerte genutzt werden, die Performance-Fehler nicht zuverlässig finden, oder weil ihre Analyse komponentenbasierte Systeme nicht ausreichend unterstützt. Dagegen wird in dieser Arbeit der Performance-Blame-Analysis-Ansatz *PBlaman* beschrieben, der auf komponentenbasierte Systeme spezialisiert ist und die in verschiedenen Kontexten einsetzbaren Performance-Kontrakte des Palladio Component Model (PCM) einsetzt. PBlaman entscheidet, welche Komponenten zu beschuldigen sind, indem es die Antwortzeitdatenreihe jeder Komponentenoperation aus dem fehlgeschlagenen Testfall mit der erwarteten Antwortzeit, die aus den Performance-Kontrakten abgeleitet wird, vergleicht. Der Systemarchitekt erhält zwei Entscheidungsunterstützungsartefakte aus der PBlaman-Analyse. Einerseits bietet der Blame-Graph eine Übersicht über den Aufrufbaum. Anderseits vergleicht der Performance-Report die Verteilung beider Antwortzeitdatenreihen und die Lageparameter für jede Komponentenoperation. Der Nutzen von PBlaman wird in zwei Fallstudien dargestellt, von denen jede eine Anwendung mit einem bestimmten Architekturstil untersucht. Die erste Fallstudie befasst sich mit dem Handelssystem aus dem „Common Component Modeling Example“ (CoCoME), das eine Schichtarchitektur implementiert. Die zweite Fallstudie analysiert ein System, das mithilfe von Algorithmen strukturierte Informationen aus unstrukturierten Texten extrahiert und das eine „Pipes-and-Filters“-Architektur realisiert. Die Fallstudien zeigen, dass PBlaman auf die Beispiele angewendet werden kann und auch dass die Entscheidungsunterstützungsartefakte Blame-Graph und Performance-Report konsistente Ergebnisse enthalten.

Abstract

In performance-driven software engineering, the performance of a system is evaluated through models before the system is assembled. After assembly, the performance is validated using performance tests. When a component-based system fails certain performance requirements during the tests, it is important to find out whether individual components yield performance errors or whether the composition is faulty. This task is called *performance blame analysis*.

Existing performance analysis approaches are not easily applicable to tackle performance blame analysis. Either they use expected performance values that do not reliably find performance errors or their analysis does not properly support component-based systems. In contrast, this thesis describes the performance blame analysis approach *PBlaman* that specializes in analyzing component-based systems and that employs the context-portable performance contracts of the Palladio Component Model (PCM). PBlaman decides what components to blame by comparing the observed response time data series of each single component operation in a failed test case to the operation's expected response time data series derived from the contracts. PBlaman's analysis generates two decision support artifacts that assist the system architects. First, the blame graph gives an overview by depicting a call tree diagram. Second, the performance report compares both response time data series' distributions and the measures of location for each component operation. The benefits of PBlaman are exemplified in two case studies, each of which representing applications that follow a particular architectural style. The first case study deals with the trading system from the Common Component Modeling Example (CoCoME), which implements a layered architecture. The second case study investigates a system that algorithmically extracts structured information from unstructured texts and that realizes a pipes-and-filters architecture. The case studies show that PBlaman is applicable to the example systems and that the decision support artifacts blame graph and performance report contain consistent results.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Seit Meyer 1992 mit der Idee des „Design by Contract“ [49] die Grundlage für die moderne komponentenbasierte Softwareentwicklung gelegt hat, hat sich diese fest im Bereich des Softwareengineering etabliert. Bei der komponentenbasierten Softwareentwicklung werden Softwaresysteme aus fertigen Bausteinen, den Komponenten, zusammengesetzt [71]. Um solche Systeme leicht zusammensetzen zu können, kommunizieren Komponenten ausschließlich über Schnittstellen. Auf jede Komponente darf nur über ihre angebotenen Schnittstellen zugegriffen werden. Entsprechend darf jede Komponente nur über ihre benötigten Schnittstellen andere Komponenten in Anspruch nehmen. Wenn eine Komponente dieselbe Schnittstelle benötigt, die eine andere Komponente anbietet, können diese Komponenten kombiniert werden. In einer solchen Softwarelandschaft können Komponenten überall dort zum Einsatz kommen, wo ihre angebotenen Schnittstellen benötigt werden. Diese explizite Bindung zwischen Komponenten erleichtert es, Komponenten wiederzuverwenden oder aber sie gegen andere Komponenten auszutauschen. Beides sind Ziele der komponentenbasierten Softwareentwicklung, die unterschiedliche Folgen nach sich ziehen [15]. Die verstärkte Wiederverwendung soll für eine erhöhte Komponentenqualität sorgen, weil sich Wartungskosten durch häufigen Einsatz schneller amortisieren. Eine hohe Komponentenqualität soll dabei auch eine erhöhte Systemqualität zur Folge haben. Die Wiederverwendung fertiger Komponenten spart die Implementierung und soll so die Produktivität bei der Systementwicklung steigern, um komponentenbasierte Systeme schneller als herkömmliche Systeme liefern zu können. Die Austauschbarkeit von Komponenten soll dazu beitragen, sich schnell auf ändernde Anforderungen einstellen zu können.

In der komponentenbasierten Softwareentwicklung sind die Entwicklungsprozesse der Komponenten und des komponentenbasierten Softwaresystems voneinander entkoppelt [16]. Auf der einen Seite durchlaufen Komponenten einen herkömmlichen Softwareentwicklungsprozess, wie man ihn auch bei der Erstellung anderer Softwarezwischenprodukte, wie Softwarebibliotheken

oder Frameworks, verwendet. Auf der anderen Seite wird ein komponentenbasiertes System nicht selbst implementiert, sondern aus am Markt verfügbaren Komponenten zusammengesetzt. Diese Komponenten werden zunächst ausgewählt und dann zu einer sogenannten Komposition zusammengesetzt. Dabei kann eine Instanz einer Komponente auch mehrfach zum Einsatz kommen. Schließlich werden diese Komponentenobjekte [15] auf bestimmte Hardwareknoten verteilt. Dies wird Komponentenverteilung genannt. Das Ausführen der verteilten Komponentenobjekte ergibt schließlich das funktionsfähige Gesamtsystem.

Die Qualität eines komponentenbasierten Systems beruht zwar entscheidend auf der Qualität der eingesetzten Komponenten, aber ebenso großen Einfluss hat beispielsweise das Zusammenspiel der Komponenten untereinander. Dementsprechend findet eine Qualitätssicherung sowohl bei der Entwicklung der Komponenten als auch bei der Entwicklung des komponentenbasierten Gesamtsystems statt. Eine der etabliertesten Qualitätssicherungsmaßnahmen ist der Test. Beim Test werden Anwendungsbeispiele auf Fehler überprüft und somit Vertrauen in die Komponente bzw. das System aufgebaut [69]. Dabei wird zunächst eine bestimmte Situation in einem Testfall nachgestellt und anschließend überprüft, ob ein sogenanntes Testkriterium eingehalten wurde. Die Auswertung des Testkriteriums ergibt das Testergebnis, also entweder bestanden, nicht bestanden oder ungültig. Ist ein Testfall nicht bestanden, so hat der Tester eine sogenannte Fehlerwirkung beobachtet, die auf einen Fehler im System hindeutet. Die Fehlerwirkung meldet der Tester in einer Fehlermeldung an die Entwickler. Die Fehlerwirkung wird durch das Testkriterium festgestellt, dass überprüft, ob bestimmte Verhaltensweisen des getesteten Systems dem erwarteten Verhalten entsprechen. Das jeweils erwartete Verhalten wird von einem Testorakel abgeleitet. Das Testorakel kann beispielsweise eine Spezifikation, ein bereits existierendes System oder das Wissen einer Person sein.

Mit Tests können neben der Funktionalität auch andere Softwarequalitäten, wie die Performance, in der Literatur auch Effizienz genannt, überprüft werden [51]. Der Begriff Performance wird im Standard ISO 9126 definiert und mit Performance-Metriken unterfüttert [35]. Gemäß den dort angeführten Performance-Kriterien „Zeitverbrauch“ und „Ressourcenverbrauch“ gibt es die Performance-Metriken Antwortzeit, Durchsatz und Ressourcenverbrauch. Diese Performance-Metriken müssen in möglichst realistischen Testfällen überprüft werden, um die Performance eines Systems zu testen. In der Realität verarbeiten Systeme häufig gleichzeitig eintreffende Anfragen. Wenn man diese Situation testen will, muss ein Testfall nebenläufig Anfragen an das System stellen und deren Performance auswerten. Da beispielsweise die Antwortzeit

einzelner Anfragen, z. B. aufgrund von Hintergrundlast, stark schwanken kann, empfiehlt es sich die Anfragen häufiger zu wiederholen und dann z. B. den Mittelwert als Testkriterium zu verwenden. Besonders bei betrieblichen Informationssystemen ist es üblich, solche stochastischen Testkriterien zu verwenden und einen gewissen Fehler zu tolerieren. Beispielsweise kann ein solches Testkriterium folgendermaßen lauten: „Die mittlere Antwortzeit der Operation XY ist kleiner als 200 ms“.

Um die Performance eines komponentenbasierten Softwaresystems zu testen, wird eine vollständige Implementierung dieses Systems zwingend benötigt. Wenn der Systemarchitekt[1] schon vorher Einblick in die Performance des Systems erhalten will, kann er mithilfe einer modellbasierten Analyse die Performance-Messungen aus dem Test näherungsweise vorhersagen. Anstatt der Implementierung werden hierfür Performance-Modelle, die die Performance-Charakteristika der Komponenten sowie die Komposition in einer geeigneten Modellierungssprache (z. B. Layered Queuing Networks (LQN) [61] bzw. das Palladio-Komponentenmodell (PCM) [8]) spezifizieren, benötigt. Diese Modelle werden entweder nach mathematischen Regeln ausgerechnet oder simuliert, um die zum Test ähnlichen Performance-Werte zu erhalten.

Bei LQN-Modellen handelt es sich um ein Modell, das sich gut mathematisch lösen lässt, da alle Werte, wie Berechnungszeiten, direkt numerisch angegeben sind. Dies hat allerdings den Nachteil, dass es bei der komponentenbasierten Entwicklung schwerer anzuwenden ist. Dagegen etabliert das PCM ein Rollenmodell bei dem verschiedene Modellteile durch Parameter von einander entkoppelt werden. Dieses Prinzip wird nun am Beispiel eines Performance-Kontrakts verdeutlicht. Performance-Kontrakte werden durch den Komponentenentwickler erstellt und vom Systemarchitekten zu einem Gesamtsystem zusammengesetzt. Ein solcher Komponentenkontrakt gibt an, welche Performance jede Komponentenoperation bietet. Im Folgenden wird beispielhaft ein parametrisierter Performance-Kontrakt für die Operation `setStockForProducts` hergeleitet. Die Operation setzt dabei den Lagerbestand verschiedener Produkte in einer SQL-Datenbank jeweils auf einen neuen Wert (s. Abbildung 1.1). In LQN wäre diese Angabe als Performance-Wert enthalten, z. B. „Operation `setStockForProducts` hat höchstens eine Antwortzeit (AZ) von 50 ms“. In einer Formel (die Formeln sind nicht in PCM-Syntax) ließe sich das wie folgt darstellen:

$$AZ <= 50ms$$

[1] Im Laufe dieser Arbeit wird der Softwarearchitekt des komponentenbasierten Gesamtsystems als Systemarchitekt bezeichnet.

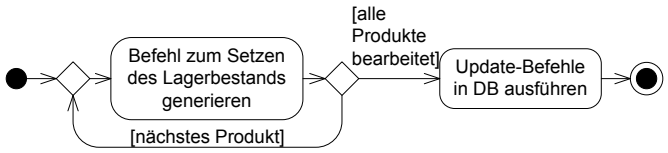

Abbildung 1.1: Ablauf der fiktiven Operation `setStockForProducts`

Bei einem PCM Performance-Kontrakt kann der Komponentenentwickler explizit durch Parameter Annahmen über die Umgebung der Komponente darstellen. Der Komponentenentwickler weiß beispielsweise nicht, auf welcher Hardware seine Komponente ausgeführt wird. Somit sollte der Performance-Kontrakt mit einem Parameter auf die Hardware des Endanwenders Rücksicht nehmen. Der Berechnungsaufwand kann beispielsweise durch Gleitkommaoperationen (FLOP) und die CPU-Geschwindigkeit dementsprechend durch Gleitkommaoperationen pro Sekunde (FLOPS) abgeschätzt werden:

$$AZ <= \frac{500FLOP}{FLOPS(CPU1)}$$

Der Komponentenentwickler weiß nicht, welche benötigten Komponenten zum Einsatz kommen. Da auch die Performance dieser Komponenten von Parametern abhängt, soll ein weiterer Parameter für jeden Aufruf einer Operation aus einer benötigten Komponente eingeführt werden. Im Beispiel-Performance-Kontrakt wird die Antwortzeit (AZ) der Operation `DB.update(...)` als Parameter hinzugefügt:

$$AZ <= \frac{200FLOP}{FLOPS(CPU1)} + AZ(DB.update(...))$$

Zuletzt kann der Komponentenentwickler auch nicht wissen, wie seine Komponente verwendet wird. Daher sollte er auch Parameter für Eingabewerte vorsehen. Es kann z. B. statt eines konstanten Berechnungsaufwandes ein Berechnungsaufwand abhängig von der Anzahl der Eingabeelemente (hier Produkte) verwendet werden:

$$AZ <= \frac{ANZAHL(Produkte) * 70FLOP}{FLOPS(CPU1)} + AZ(DB.update(...))$$

Die drei Aspekte Komponentengebrauch, benötigte andere Komponenten und ausführende Hardware werden auch als der Kontext einer Komponente bezeichnet [6]. Dadurch, dass der Komponentenentwickler den Kontext durch Parameter berücksichtigt, kann der Performance-Kontrakt unverändert in verschiedenen Gesamtsystemen verwendet werden.

1.1 Problemstellung

Angenommen es wurde während eines Tests des Gesamtsystems ein Performance-Fehler entdeckt. Der Systemarchitekt beauftragt nun die Entwickler damit, den Fehler zu analysieren und zu beheben. Allerdings sind an der Entwicklung komponentenbasierter Systeme nicht nur die Systementwickler sondern auch verschiedene Gruppen von Komponentenentwicklern beteiligt. Daher muss der Systemarchitekt zunächst entscheiden, welche dieser Gruppen den Fehler vermutlich beheben kann. Er muss also einen Teil der Fehleranalyse vorwegnehmen, um den Fehler an die passendste Entwicklergruppe weiterzugeben. Insbesondere muss er untersuchen, ob der Fehler durch eine oder mehrere Komponenten verursacht wurde, oder ob der Fehler durch die Komponentenverwendung im Kontext des Gesamtsystems entsteht. Diese Analyse, die zur Entscheidung führt an welche Entwicklergruppe der Fehler weitergegeben wird, nennt sich „Performance-Blame-Analysis“ (vgl. Abschnitt 3.1).

Zum Beispiel ist ein Performance-Fehler entdeckt worden, weil eine Systemoperation die im Testkriterium festgelegte mittlere Antwortzeit von 200 ms überschritten hat. Nun muss also der Systemarchitekt im Rahmen der Performance-Blame-Analysis klären, ob der Fehler aufgetreten ist, weil bestimmte Komponenten eine zu hohe Antwortzeit aufweisen oder nicht. Wenn eine oder mehrere Komponenten eine zu hohe Antwortzeit aufweisen, sollen die jeweiligen Komponentenentwickler den Fehler weiter analysieren und beseitigen. Liegt der Fehler andernorts, soll er von den Systementwicklern weiter analysiert und behoben werden.

Performance-Blame-Analysis ist besonders bei komponentenbasierter Softwareentwicklung wichtig. Wie schon erwähnt, ist die Entwicklung des Gesamtsystems von der Implementierung der Komponenten getrennt und die Gesamtsystem- bzw. Komponentenentwickler interagieren über einen Komponentenmarkt. In dieser Situation ist der Quelltext der Komponenten häufig nicht verfügbar. In einer solchen Blackbox-Komponente muss die Beseitigung eines Fehlers zwangsläufig beim Komponentenhersteller erfolgen. Je nachdem, welche rechtliche Vereinbarung dem Komponenteneinsatz im Gesamtsystem zugrunde liegt, stellen die Komponentenhersteller den Systementwicklern diese Fehlerbehebung in Rechnung. Auf jeden Fall geht mit der zusätzlichen Erläuterung des Fehlers gegenüber dem Komponentenhersteller Zeit verloren, die bei einer Eigenentwicklung nicht anfällt. Daher erfordert es die Interaktion über den Markt, besonderes Augenmerk darauf zu legen auf Anhieb die richtige Entwicklergruppe mit der Fehlerbeseitigung zu beauftragen.

1.2 Beispielanwendung CoCoME

Im Folgenden wird ein Beispiel beschrieben, auf das eine Performance-Blame-Analysis angewendet werden kann. Dieses Beispiel wird im Verlauf dieser Arbeit verwendet, um verschiedene Aspekte der Performance-Blame-Analysis und des Lösungsansatzes zu erläutern. Eine detaillierte Fallstudie für diese Beispielanwendung findet sich in Abschnitt 5.1. Das Beispiel basiert auf dem „Common Component Modeling Example“ (CoCoME) [33]. CoCoME ist ein Benchmark-System für verschiedene Analysemethoden bezüglich komponentenbasierter Systeme. Das CoCoME spezifiziert ein komponentenbasiertes Handelssystem einer Supermarktkette, das einem realen System nachempfunden ist. Das Handelssystem enthält die komplette Infrastruktur von den Kassen (engl. „cash desk“), an denen Waren verkauft werden, über die Filial-Server (engl. „store server“), die die Verkäufe registrieren, bis zum Zentral-Server (engl. „enterprise server“), der alle Information speichert und aggregiert. Das Handelssystem wird im Rahmen dieser Arbeit auch *CoCoME-System* genannt.

Die Struktur des CoCoME-Systems ist in Abbildung 1.2 zu sehen. In diesem Systemstrukturdiagramm lässt sich die Verbindung der Hardware-Knoten im CoCoME-System erkennen. Beispielsweise ist ein Filial-Server mit mehreren Kassen verbunden. Auf jedem dieser Hardware-Knoten sind eine oder mehrere Komponenten installiert (vgl. auch Abbildung 1.3). Auf dem Zentral-Server ist zum Beispiel die Datenbank, die Komponente `ProductDispatcher` und die Komponente `EnterpriseQuery` installiert. Dagegen finden sich auf den Filial-Servern die Komponenten `Store` und `StoreQuery`.

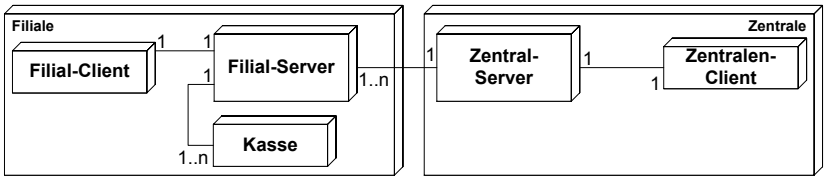

Abbildung 1.2: Die Struktur des CoCoME-Systems; CoCoME dient als Beispiel im Rahmen dieser Arbeit

Die Art und Weise wie die Komponenten des CoCoME-Systems komponiert sind, bildet eine Schichtarchitektur. Damit steht das CoCoME-System stellvertretend für diesen Architekturstil. Dabei bilden die auf den Kassensystemen und den Clients installierten Komponenten die GUI-Schicht. Die auf dem Filial-Server und dem Zentral-Server ausgeführten Komponenten, mit Ausnahme

der Datenbank, bilden die Anwendungsschicht. Die auf dem Zentral-Server befindliche Datenbank-Komponente bildet die Datenbankschicht. Das Beispiel befasst sich ausschließlich mit der Anwendungsschicht, also mit den Filial-Servern und dem Zentral-Server.

Der Beispieltestfall umfasst den komplexesten Anwendungsfall „Filialaustausch“, bei dem Waren zwischen den Filialen einer Supermarktkette ausgetauscht werden. Dieser Anwendungsfall wird durch den Verkauf einer Ware eingeleitet. Dabei überprüft der Filial-Server zunächst, ob der restliche Lagerbestand dieser Ware einen bestimmten Schwellwert unterschreitet. Wenn das der Fall ist, fordert der Filial-Server beim Zentral-Server fehlende Waren an. Der Zentral-Server wird Filialen in der Nähe anweisen, überzählige Waren an die anfragende Filiale zu senden. Dabei optimiert der Zentral-Server die Warenbewegungen, so dass die gesamte Transportdistanz minimal ist und keine Filiale mit zu geringem Lagerbestand verbleibt.

Nun wird die genaue Abfolge von Komponentenoperationen für den Anwendungsfall Filialaustausch erläutert. Abbildung 1.3 zeigt die Komponentenoperationsaufrufe beim Filialaustausch. Die beteiligten Komponentenobjekte sind auf verschiedene Knoten verteilt: auf den Zentral-Server und auf die Filial-Server. Die durchgezogenen Pfeile entlang der Schnittstellen-Kopplung (in Kopf/Fassung-Notation) stehen für Operationsaufrufe, wogegen die gestrichelten Pfeile für die Operationsrückgabe stehen. Die Operationsaufrufe sind entsprechend ihrer Abfolge nummeriert. Bei den Operationsrückgaben wird auf eine Nummerierung verzichtet, wenn die Rückgabe direkt nach dem zugehörigen Aufruf der Operation erfolgt.

Im oberen Teil von Abbildung 1.3 wird dargestellt, dass der Beispieltestfall damit beginnt, dass eine Testtreiber-Komponente einen Warenverkauf registriert (Nachricht 1). Der Verkauf sorgt dafür, dass nicht mehr genügend Waren dieser Art auf Lager sind. Daraufhin ermittelt der Filial-Server alle fehlenden Waren (Nachricht 2) und fordert die fehlenden Waren vom Zentral-Server an (Nachricht 3). Als Nächstes ermittelt die `ProductDispatcher`-Komponente die nötigen Informationen über die Supermarktkette. Dabei handelt es sich im Wesentlichen um die Information, welche Filialen existieren und welche Produkte dort verkauft werden. Dann fragt die Komponente die Lagerbestände für die fehlenden Waren bei allen anderen Filialen nach (Nachrichten 5), dabei wird eine Nachricht an jeden Filial-Server versandt.[2] Der Zentral-Server errechnet nun einen optimalen Transportplan

[2] Der Multi-Node „andere Filial-Server“ in Abbildung 1.3 steht für mehrere Filial-Server. Alle Nachrichten dorthin symbolisieren mehrere Nachrichten, eine an jeden der anderen Filial-Server.

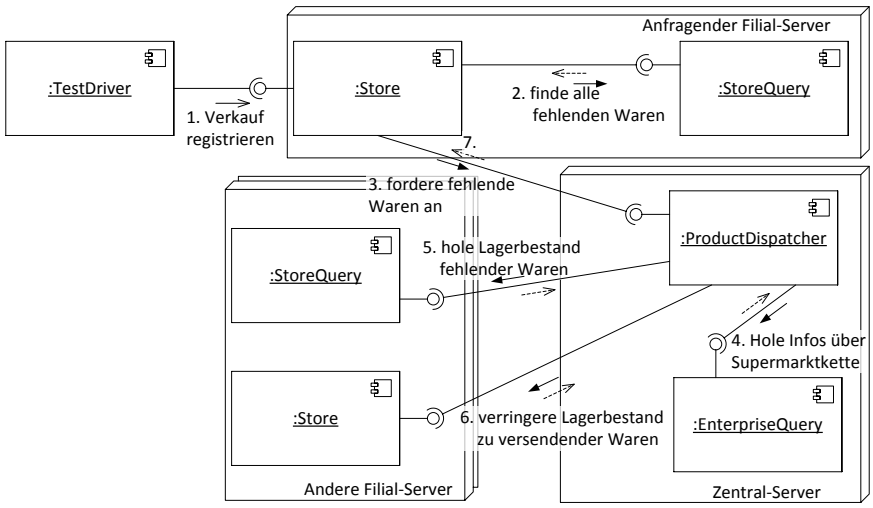

Abbildung 1.3: Das Beispielszenario Filialaustausch mit allen beteiligten Komponenten und Operationsaufrufen.

und verringert die Lagerbestände für die Filialen, die Waren versenden sollen (Nachrichten 6). Danach unterrichtet der Zentralserver den anfragenden Filial-Server davon, welche Waren an ihn versendet werden (Nachricht 7). Zuletzt vermerkt der Filial-Server die eingehenden Waren in seinem Lagerbestand.

In dieser Arbeit wird angenommen, dass das beschriebene Szenario als Testfall verwendet wird und dass das Testkriterium, eine Antwortzeitschwelle, verletzt wird. Dann wird eine Performance-Blame-Analysis notwendig. In dieser Analyse wird insbesondere geklärt, ob eine oder mehrere der beteiligten Komponentenoperationen, die in Abbildung 1.3 als Nachrichten enthalten sind, den Fehler durch zu hohe Antwortzeiten verursachen.

1.3 Lösungsansatz

In dieser Arbeit wird der Ansatz „kontraktbasierte Performance-Blame-Analysis“ (PBlaman) vorgestellt. PBlaman hilft dem Systemarchitekten bei der Performance-Blame-Analysis. Wenn also ein Performance-Testfall fehlschlägt, kann PBlaman beurteilen, welche Komponentenoperationen wahrscheinlich für den Fehler verantwortlich sind. Wenn PBlaman keine Komponentenoperationen als mögliche Ursachen für den Fehler findet, ist der Fehler aufseiten der Systementwickler zu suchen. PBlaman ist derzeit auf die Analyse von Antwortzeitfehlern beschränkt.

PBlaman arbeitet auf Grundlage des fehlgeschlagenen Testfalls. Um zu analysieren, welche Komponentenoperationen als Fehlerursache in Frage kommen, benötigt PBlaman einen Performance-Kontrakt für jede am Testfall teilnehmende Komponente. PBlaman geht davon aus, dass diese Performance-Kontrakte als Zusicherung der Komponentenentwickler mit den Komponenten ausgeliefert werden. Im Testfall unseres Anwendungsbeispiels (s. Abbildung 1.3) müssten also Performance-Kontrakte für die Komponenten `Store`, `StoreQuery`, `ProductDispatcher` und `EnterpriseQuery` vorliegen.

Im Folgenden wird der Performance-Kontrakt für die Komponentenoperation „verringere Lagerbestand zu versendender Waren“ in der `Store`-Komponente dargestellt (vgl. Nachricht 6 in Abbildung 1.3). Dabei wird für jedes zu versendende Produkt zunächst über die Operation `queryProductById` der Komponente `StoreQuery` eine Datenbankrepräsentation des Produkts geholt. Es fließt also die Antwortzeit (AZ) dieser Operation hier ein. An der Datenbankrepräsentation der Produkte wird nun jeweils der Lagerbestand durch eine Zuweisung verringert. Der Ressourcenverbrauch der Zuweisung wird hier mit 5 FLOP angegeben. Als Formel (die Formel ist nicht in PCM-Notation) ließe sich der Kontrakt wie folgt formulieren:

$$AZ <= ANZAHL(Produkte) * \left(AZ(StoreQuery.queryProductById(...)) + \frac{5 * FLOP}{FLOPS(CPU1)} \right)$$

PBlaman nutzt derartige Performance-Kontrakte als Testorakel, aus denen die erwarteten Operationsantwortzeiten für den Testfall abgeleitet werden. Im PBlaman-Ansatz werden Performance-Kontrakte in Palladio-Komponentenmodell-Notation verwendet, weil sich diese Kontrakte besonders gut für den Austausch zwischen Komponenten- und Systementwickler eignen, da sie den Komponentenkontext mithilfe von Parametern berücksichtigen können.

Das Ergebnis des PBlaman-Ansatzes ist eine Menge von Komponentenoperationen, die als Fehlerursache für den Testfall identifiziert wurden. Im Anwendungsbeispiel würde PBlaman also eine Menge der in Abbildung 1.3 als Nachrichten dargestellten Komponentenoperationen als Ergebnis liefern, z. B. {„fordere fehlende Waren an“, „verringere Lagerbestand zu versendender Waren“}. Der Systemarchitekt kommt mithilfe einer Entscheidungsunterstützung zu diesem Ergebnis. Die Ergebnisse der Entscheidungsunterstützung bieten ihm dabei eine aufrufbaumartige Ansicht, in der mit Farben gekennzeichnet ist, ob die jeweilige Komponentenoperation als Fehlerursache in Frage kommt. Hinzu kommt die Verteilung der Antwortzeiten und weitere statistische Werte für die jeweilige Komponentenoperation.

PBlaman verarbeitet die genannten Eingaben, wie in Abbildung 1.4 dargestellt, in drei Schritten. Zunächst werden die Antwortzeiten für jede Komponentenoperation im Test gemessen und aus den Performance-Kontrakten abgeleitet (Schritt 1). Dann werden die Antwortzeiten aus den beiden Quellen im Rahmen der Entscheidungsunterstützung miteinander verglichen (Schritt 2). Wenn die Test-Antwortzeit einer Komponentenoperation höher ausfällt als die aus den Performance-Kontrakten abgeleitete Antwortzeit, wird diese Komponentenoperation für den Fehler verantwortlich gemacht. Dieser Vergleich ist nicht trivial, da es sich nicht um je zwei einzelne Antwortzeitwerte handelt sondern um zwei Antwortzeitdatenreihen je Komponentenoperation. Die Entscheidungsunterstützung resultiert in zwei Artefakten, eines umfasst die angesprochene Aufrufbaumdarstellung, den sogenannten „Blame Graph“, und das andere, genannt „Peformance-Report“, die Antwortzeitverteilung sowie die weiteren statistischen Werte. Auf dieser Grundlage entscheidet der Systemarchitekt, welche Komponentenoperationen er für die Ursache des Fehlers hält (Schritt 3). Dieser Entscheidung folgend veranlasst er die Behebung des Fehlers (Schritt 4).

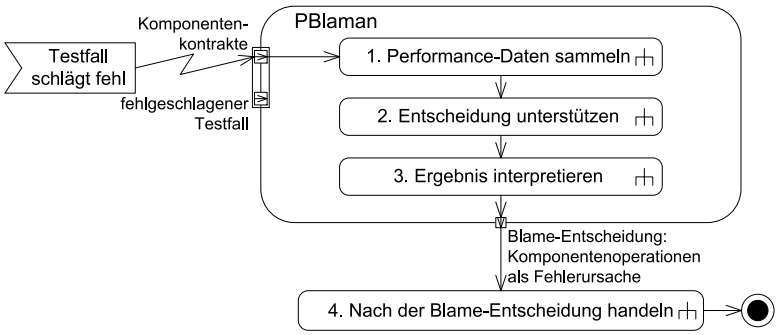

Abbildung 1.4: Das dargestellte UML-Aktivitätendiagramm zeigt die Prozessschritte des Ansatzes „kontraktbasierte Performance-Blame-Analysis“ (PBlaman)

Der Vergleich einer Testdatenreihe mit einer Datenreihe aus der Performance-Vorhersage, ist nur dann gültig, wenn in der Performance-Vorhersage, ein zum Testfall äquivalentes Szenario betrachtet wird. Dabei sollen die Eingaben des Systems vergleichbar sein und auch die Komposition und Komponentenverteilung des Systems müssen übereinstimmen. Um dies sicherzustellen sollen die Tester, die Testfallnotation „Palladio-basierte Testfälle“ [13, 12, 11] einsetzen. Dabei werden einzelne Aspekte des Testfalls mithilfe des PCM spezifiziert. Auf dieser Grundlage kann später leicht eine PCM-Instanz zur

Performance-Vorhersage erstellt werden, die äquivalent zu diesem Testfall ist.

1.4 Verwandte Arbeiten

So wie bei Tests auch das erwartete Verhalten spezifiziert wird, um den Ausgang eines Testfalls bewerten, ist dies auch bei der Performance-Analyse nötig. Es kann zwar hilfreich sein, wenn Ansätze [1, 62, 70] eine einzelne Performance-Datenreihe nach Auffälligkeiten untersuchen, aber für eine abschließende Bewertung, ob sich eine Komponente fehlerhaft verhält, ist dies nicht ausreichend. Somit können diese Ansätze nur vorgeben, welche Performance-Metriken erfasst werden, wie diese weiter aggregiert werden und wie Auffälligkeiten in der Performance-Datenreihe erkannt werden können.

Wenn Ansätze [38, 48, 63] zwei Performance-Datenreihen miteinander vergleichen, ist eine Bewertung der Performance aufgrund von Abweichungen möglich. Die Bewertung stellt Änderungen im Verhalten von Komponenten und Komponentenoperationen fest. Damit die Bewertungsergebnisse Bestand haben, ist es erforderlich, dass die Basisdatenreihe tatsächlich die gewünschten Eigenschaften aufweist. Wenn das nicht der Fall ist, wird ein Performance-Fehler keine Abweichung erzeugen. Die Basisdatenreihe und die Vergleichsdatenreihe sind spezifisch für eine bestimmte Lastsituation und müssen damit für jeden Testfall einzeln erfasst werden.

Zuletzt können Ansätze auch eine Performance-Datenreihe mit einer Spezifikation vergleichen. Dabei können aus der Spezifikation Performance-Datenreihen für den Testfall abgeleitet werden [31, 30], die als erwartete Performance verwendet werden können. Solche Ansätze können, abhängig von der Spezifikation, eine zutreffende Bewertung von Komponenten vornehmen. Alternativ kann die Spezifikation auch fest vorgegebene Werte enthalten. Da die Performance von Komponenten stark von ihrem Kontext abhängen, sind von Seiten der Hersteller fest vorgegebene Performance-Kriterien, wie bei eingebauten Tests [9] (engl. „built-in tests“), nicht in der Lage eine adäquate Bewertung der Performance vorzunehmen.

Einige der angesprochenen Ansätze [62, 70] verzichten auf eine Visualisierung. Andere Ansätze [1, 48] visualisieren die Ergebnisse auf einem sehr detaillierten Niveau, das einen übersichtlichen Einstieg, z. B. auf Komponentenebene, vermissen lässt. Wenn eine Aggregierung auf z. B. Hostebene vorgenommen wird [48], ist die Darstellung häufig mit sehr vielen Details

angereichert, was zwar hilfreich ist, aber die Übersicht erschwert. Dagegen bildet der Ansatz von Jiang et al. [38] zunächst nur die Abweichung auf Komponentenebene ab, auf Wunsch können dann weitere Details dazu eingeblendet werden. In PBlaman kommt eine Visualisierung bei den Ergebnissen der Entscheidungsunterstützung zum Einsatz. Dabei wird durch Farben in einen kompakten Aufrufbaum angezeigt, ob Komponentenoperationen für den zu untersuchenden Fehler verantwortlich sind. Dabei greift PBlaman auf eine bekannte Visualisierung [28, 29, 39] zurück, die auch in ähnlicher Form in Performance-Profiler-Werkzeugen [20, 19] zum Einsatz kommt.

1.5 Wissenschaftlicher Beitrag

Der Beitrag dieser Arbeit ist die ausführliche und vollständige Vorstellung des PBlaman-Ansatzes. Der Ansatz zeigt, wie Performance-Kontrakte als Testorakel genutzt werden können, um eine wirksame Performance-Blame-Analysis umzusetzen. In meiner bisherigen Arbeit habe ich schon einige Schritte des Ansatzes präsentiert [11, 12]. Eine Übersicht über den gesamten Ansatz wurde ebenfalls schon veröffentlicht [13]. In PBlaman wird eine Antwortzeitdatenreihe aus dem Test mit einer Antwortzeitdatenreihe, die mit modellbasierter Performance-Analyse aus den Performance-Kontrakten gewonnen wird, verglichen. Ein Vorwurf wird gegen eine Komponentenoperation und damit auch gegen die Komponente erhoben, wenn die Antwortzeitdatenreihe aus dem Test die insgesamt höheren Werte aufweist. Dieses Ergebnis wird für alle Komponentenoperationen auf einen Blick visualisiert, indem die Analyseergebnisse durch Farben in eine kompakte Aufrufbaumvisualisierung integriert werden.

PBlaman wurde in zwei verschiedenen Fallstudien evaluiert. Die erste Fallstudie [12] zeigt, dass der PBlaman-Ansatz in komponentenbasierten Systemen mit Schichtarchitektur eingesetzt werden kann. In der zweiten Fallstudie [13] wird untersucht, ob PBlaman auch in Anwendung mit anderem Architekturstil, nämlich dem „Pipes-and-Filters“-Stil, nützlich ist. Um die Nützlichkeit der Ergebnisse näher zu untersuchen, wurde auch untersucht, wie schnell PBlaman eine Komponentenoperation als mögliche Fehlerursache einstuft. Dazu wurde eine unauffällige Komponente in mehreren Testläufen durch wachsende Wartezeiten verlangsamt.

1.6 Aufbau der Arbeit

Im Folgenden wird der Aufbau dieser Arbeit kapitelweise zusammengefasst.

Kapitel 2

In Kapitel 2 werden die Grundlagen der Performance-Blame-Analysis und auch des PBlaman-Ansatzes dargestellt. Da diese Arbeit auf komponentenbasierte Systeme eingeht, werden zunächst die grundlegenden Begriffe in Abschnitt 2.1 erläutert. Dabei werden die Bereiche Spezifikation, Performance-Messung sowie Implementierung/Ausführung im Hinblick auf komponentenbasierte Systeme beleuchtet.

Im Anschluss an diese grundsätzliche Erläuterung werden die komponentenbasierten Systeme noch einmal in Abschnitt 2.2 aus Sicht der Literatur thematisiert. Dabei wird der Komponentenbegriff diskutiert und die für diese Arbeit wichtigen Aspekte werden hervorgehoben. Zudem wird auch ein Entwicklungsprozess für komponentenbasierte Systeme vorgestellt.

Daraufhin wird das Palladio Componentent Model (PCM) in Abschnitt 2.3 vorgestellt. Dabei werden die einzelnen Diagrammarten und auch die Simulation von PCM-Modellen, bei der die Performance-Metriken aus dem Modell gewonnen werden, erläutert. Außerdem wird das PCM noch dem in den Grundbegriffen dargestellten Modell von komponentenbasierter Software gegenübergestellt.

Schließlich wird in Abschnitt 2.4 noch auf den Bereich Performance-Test eingegangen. Hier werden Teststufen und Performance-Tests im Allgemeinen sowie Testfälle und ihre Notation im Besonderen erläutert. Zuletzt werden die Eigenarten des Tests komponentenbasierter Systeme diskutiert.

Kapitel 3

In Kapitel 3 wird die Performance-Blame-Analysis anhand von Literaturquellen näher erläutert und für diese Arbeit präzisiert (s. Abschnitt 3.1). Daraus werden in Abschnitt 3.2 Anforderungen an einen Performance-Blame-Analysis-Ansatz hergeleitet. Im Anschluss werden verwandte Arbeiten aus den Bereichen Performance-Analyse und -Visualisierung vorgestellt (s. Abschnitt 3.3). Die Performance-Analyse-Ansätze werden anhand der zuvor hergeleiteten Anforderungen bewertet. Für die Performance-Visualisierungen werden zunächst eigene Anforderungen hergeleitet und dann werden die

Visualisierungen analog zu den Analyseverfahren vorgestellt und bewertet. Schließlich werden die Ansätze aus den jeweiligen Bereichen in Abschnitt 3.4 noch vergleichend zueinander bewertet. Dabei werden auch die noch verbliebenen Anforderungen identifiziert, die in dieser Arbeit bearbeitet werden sollen.

Kapitel 4

Kapitel 4 stellt den PBlaman-Ansatz im Detail vor. Zunächst werden die Palladio-basierten Testfälle in Abschnitt 4.1 diskutiert, mit denen die Tester Performance-Testfälle spezifizieren sollen. Hierbei werden einzelne Aspekte des Testfalls mithilfe von PCM-Diagrammen beschrieben aus denen anschließend ein zu vervollständigendes JUnit-Testskript generiert wird. Anhand der Palladio-basierten Testfälle kann immer auch eine äquivalente PCM-Instanz erstellt werden, mit der der Testfall simuliert werden kann.

Im Anschluss werden die Hauptschritte des PBlaman-Prozesses, wie sie in Abbildung 1.4 dargestellt sind, durch Subaktivitäten konkretisiert. Für das Sammeln der Performance-Daten wird in Abschnitt 4.2 besprochen, wie dies im Einzelnen für den Performance-Test und für die Performance-Vorhersage geschieht. Der wichtigste Hauptschritt ist die Entscheidungsunterstützung (s. Abschnitt 4.3). Hierfür wird zunächst die Auswahl der Entscheidungskriterien und Visualisierungen, die in den Entscheidungsunterstützungsartefakten zum Einsatz kommen, begründet. Anschließend wird dargestellt, welche einzelnen Schritte notwendig sind, um die Entscheidungsunterstützungsartefakte zu erstellen. Für den letzten Hauptschritt, die Ergebnisinterpretation (s. Abschnitt 4.4), wird dem Systemarchitekten ein Vorgehen auf Grundlage der Entscheidungsunterstützungsartefakte vorgeschlagen. Schließlich wird in Abschnitt 4.5 noch einmal explizit auf die im Rahmen von PBlaman zum Einsatz kommenden Werkzeuge eingegangen.

Kapitel 5

Der PBlaman-Ansatz wurde mit zwei Fallstudien evaluiert. Jede der Fallstudien zeigt die Anwendbarkeit des Ansatzes auf eine Beispielanwendung mit einem bestimmten Architekturstil. Die erste Fallstudie (vgl. [12]) untersucht in Abschnitt 5.1 zwei Implementierungen des „Common Component Modeling Example“ (CoCoME) [33]. Das CoCoME ist als Benchmark für verschiedene komponentenbasierte Analysemethoden entworfen worden.

Das dort implementierte Handelssystem für Supermarktketten realisiert eine komponentenbasierte Schichtarchitektur.

Die zweite Fallstudie [13] betrachtet in Abschnitt 5.2 ein System für die algorithmische Analyse unstrukturierter Texte. Die Beispielanwendung basiert auf dem Framework „Unstructured Information Management Architecture“ (UIMA) [23] und implementiert den „Pipes-and-Filters“-Architekturstil. Die Beispielanwendung ermöglicht es, Vorhersagen bezüglich der finanziellen Entwicklung von Unternehmen über die Zeit aus unstrukturierten Texten zu extrahieren.

Die Evaluierung zeigt, wie in Abschnitt 5.3 zusammenfassend beschrieben wird, dass der PBlaman-Ansatz in beiden Fallstudien angewendet werden kann. Dabei ist die Kategorisierung nach Beschuldigung im Blame-Graph konsistent mit den detaillierten Performance-Werten aus dem Performance-Report. Diese Ergebnisse legen nahe, dass der PBlaman-Ansatz auch bei anderen Anwendungen mit den angesprochenen Architekturstilen anwendbar ist.

Kapitel 6

Zunächst wird der PBlaman-Ansatz in Abschnitt 6.1 anhand der verbliebenen Anforderungen bewertet und auch den besten Ansätzen der verwandten Arbeiten gegenübergestellt. In Abschnitt 6.2 werden die prinzipbedingten Grenzen des PBlaman-Ansatzes diskutiert.

Kapitel 7

In Abschnitt 7.1 werden die Ergebnisse der Arbeit zunächst zusammengefasst. Schließlich gibt der Abschnitt 7.2 einen Ausblick. Dabei werden zunächst Verbesserungsmöglichkeiten für den PBlaman-Ansatz in den Bereichen Palladio-basierte Testfälle, Messung und Simulation, Visualisierung sowie Evaluierung aufgezeigt. Über PBlaman hinausgehend wird eine erweiterte Performance-Analyse und ein Ansatz zur dynamischen Generierung von Performance-Analyse-Systemen vorgestellt.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen dieser Arbeit erläutert. Zunächst werden einige Grundbegriffe in Abschnitt 2.1 eingeführt. In den folgenden Abschnitten werden daran anknüpfend weitere Zusammenhänge erläutert. In Abschnitt 2.2 werden komponentenbasierte Systeme eingeführt. In diesem Zusammenhang wird der Komponentenbegriff aus der Literatur hergeleitet und die für diese Arbeit wichtigen Aspekte von Komponenten werden verdeutlicht. Im Anschluss wird der Entwicklungsprozess für komponentenbasierte Systeme erläutert.

Im Abschnitt 2.3 wird dann das Palladio Komponentenmodell eingeführt, das bei PBlaman für die Performance-Kontrakte und die Simulation eines Testfalls anhand der Kontrakte verwendet wird. Zuletzt wird im Abschnitt 2.4 der Bereich der Performance-Tests dargestellt. Dabei werden nicht nur die Performance-Tests als solche sondern auch das Artefakt Testfall sowie seine Notation vorgestellt. Zuletzt wird noch kurz auf Besonderheiten bei Tests von komponentenbasierten Systemen eingegangen.

2.1 Grundlegende Begriffe

Die Performance-Blame-Analysis befasst sich mit der Analyse von im Test gefundenen Performance-Fehlern bei komponentenbasierten Systemen. Dementsprechend sind die Grundbegriffe einerseits im Bereich der komponentenbasierten Systeme, im Bereich des Testens und im Bereich der Performance-Messung zu suchen. Außerdem geht die hier vorgestellte Lösung PBlaman davon aus, dass eine Spezifikation simuliert wird. Somit muss sowohl die Spezifikation von komponentenbasierten Systemen und Testfällen als auch die Ausführung und Verwendung der implementierten Artefakte berücksichtigt werden. Im Folgenden stellen wir die Zusammenhänge für die drei Bereiche Spezifikation, Performance-Messung und Implementierung/Ausführung dar. Dabei wird in diesem Abschnitt lediglich ein Überblick über die relevanten

Begriffe und ihre Zusammenhänge gegeben. Die eingehende Erläuterung der Begriffe folgt in den Abschnitten 2.2 bis 2.4.

Zunächst stellt Abbildung 2.1 die Zusammenhänge für den Bereich der Spezifikation dar. In der Abbildung werden komponentenbasierte Systeme mithilfe von Komponenten, Komposition und Komponentenverteilung spezifiziert. Dazu kommt noch die Systembenutzung, die entweder für die modellbasierte Performance-Analyse oder aber für Testfälle Vorgaben macht. Die Testfälle hängen über die Performance-Metriken auch mit dem Bereich der Performance-Messung zusammen.

Komponenten spezifizieren ihre Funktionalität und ihre funktionalen Abhängigkeiten mithilfe von Schnittstellen (vgl. Unterabschnitt 2.2.1). Schnittstellen bestehen aus mehreren Operationen mit ihrer jeweiligen Operationssignatur. Für diese Arbeit ist insbesondere wichtig, dass verschiedene Operationen eindeutig identifiziert werden können, um die Messwerte für jede Operation kategorisieren können. Auf die dafür erforderliche Signatur mit ihren Parametern und Parametertypen wird hier nicht näher eingegangen. Die angebotenen Schnittstellen spezifizieren die Funktionalität der Komponente, wogegen die benötigten Schnittstellen die funktionalen Abhängigkeiten der Komponente festlegen. Beides zusammengenommen stellt den funktionalen Kontrakt der Komponente dar. Des Weiteren wird angenommen, dass jede Komponente Performance-Kontrakte aufweisen kann. Ein Performance-Kontrakt ordnet einer Operation zu, welche Performance sie im Bezug auf eine bestimmte Performance-Metrik haben soll. Die dargestellte Spezifikation von Performance-Kontrakten umfasst nicht die Einbeziehung von Kontextinformationen (vgl. Kapitel 1). In Abschnitt 2.3 wird detailliert auf die Realisierung der Performance-Kontrakte mithilfe des „Palladio Component Model“ eingegangen. Hier wird daher nur dargestellt, was allen Performance-Kontrakten gemeinsam ist. Weitere Kontrakte bezüglich anderer Qualitätseigenschaften sind für diese Arbeit nicht von Interesse, können aber Bestandteil einer Komponentenspezifikation sein.

Die Komponenten werden in einer Komposition verwendet (s. Abbildung 2.1). In der Komposition kann jede Komponente mehrfach zum Einsatz kommen (Element Komponenteneinsatz). Dabei werden die Komponenten mithilfe ihrer Schnittstellen gekoppelt (Element Komponentenkopplung), wodurch die Komposition entsteht. Zudem können mehrere Schnittstellen an bestimmten Komponenteneinsätzen als Systemschnittstellen gekennzeichnet sein. Die Operationen innerhalb einer Systemschnittstelle werden auch als Systemoperation bezeichnet. Die Systemschnittstellen stellen die angebotenen Schnittstellen des komponentenbasierten Systems dar. Benötigte Schnittstel-

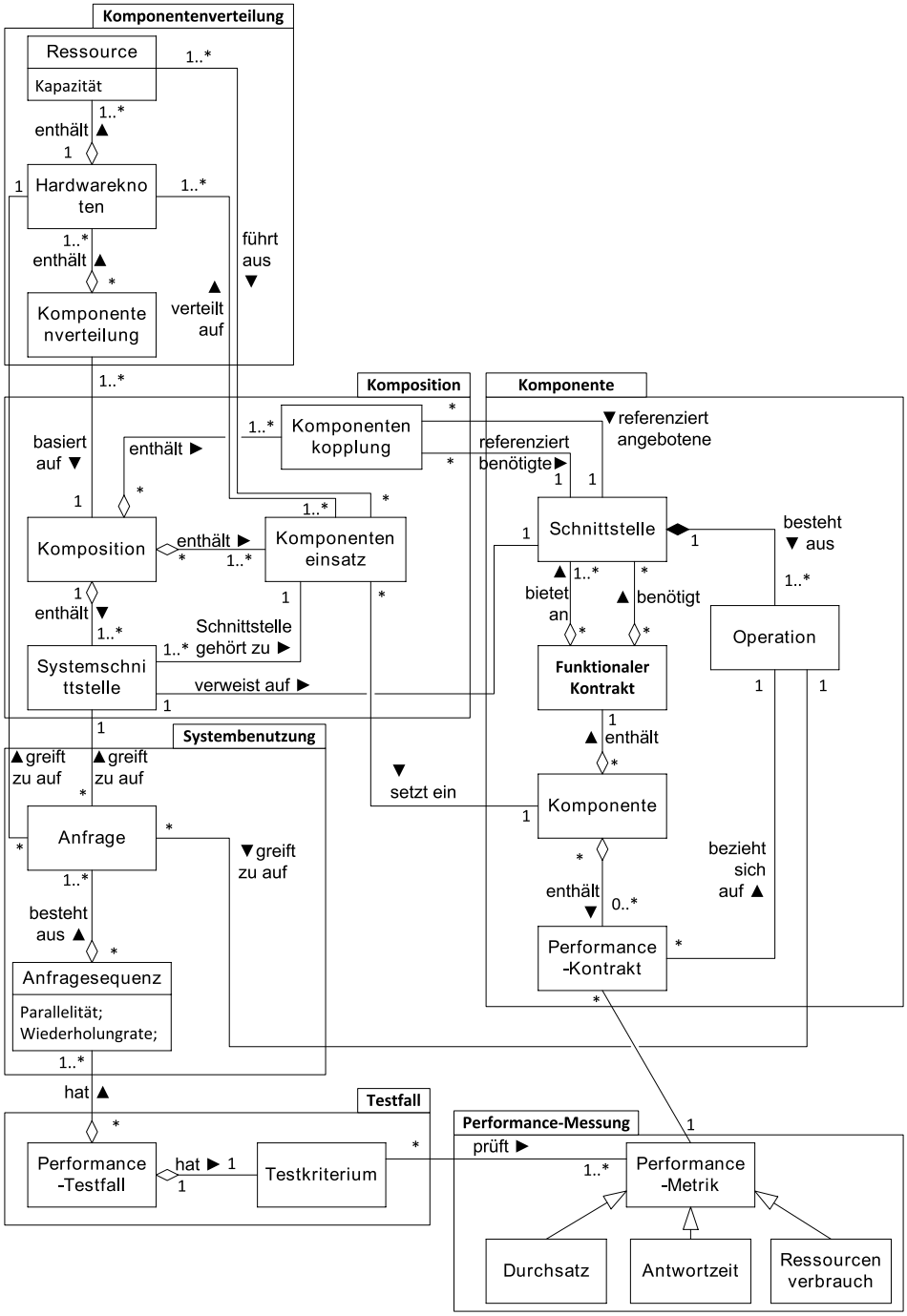

Abbildung 2.1: Begriffe im Bereich Spezifikation

len des Systems, also benötigte Schnittstellen, die nicht durch die Komposition gekoppelt werden, sind im Diagramm nicht explizit modelliert. Zum Zeitpunkt des Tests muss an jeder dieser Schnittstellen beispielsweise ein externer Dienst oder ein Platzhalter angeschlossen sein. Die externen Dienste bzw. Platzhalter sollen die zu erwartende Performance durch Kontrakte bekannt machen.

Aufbauend auf der Komposition, legt die Komponentenverteilung fest, wie die eingesetzten Komponenten auf die verschiedenen Hardwareknoten verteilt werden. Zusätzlich werden die einzelnen Hardwareknoten mit ihren verschiedenen Ressourcen und deren Kapazität definiert. Jeder Komponenteneinsatz wird dabei je Komponentenverteilung nur genau einmal auf einen Hardwareknoten verteilt und auf einer oder mehreren seiner Ressourcen ausgeführt. Falls mehrere Instanzen einer Komponente auf einem Hardwareknoten ausgeführt werden sollen, muss dies schon mithilfe der Komponenteneinsätze modelliert werden. Es handelt sich bei dem Komponenteneinsatz also im Sinne von Cheesman und Daniels [15] um Komponentenobjekte (s. dazu auch Unterabschnitt 2.2.1).

Ein komponentenbasiertes System kann über die Systemschnittstellen von außen, beispielsweise durch einen Testtreiber, aufgerufen werden (s. Abbildung 2.1). Eine Anfrage richtet sich dabei an eine bestimmte Operation aus der dazugehörenden Systemschnittstelle auf einem spezifischen Hardwareknoten. Die Abfolge der Anfragen, die an das System gerichtet werden, wird durch eine Anfragesequenz spezifiziert. Der Einfachheit halber stellt die Abbildung keine Details einer Anfragesequenz dar. Eine Anfragesequenz ist ein Kontrollfluss, der aus Anfragen und Wartezeiten besteht und auch bedingte Anweisungen und Schleifen zulässt. Außerdem legt eine Anfragesequenz fest, wie häufig die Sequenz parallel durchgespielt wird und sie gibt die Rate an, mit der die Sequenz jeweils wiederholt wird. Ein Performance-Testfall (vgl. Abschnitt 2.4) umfasst üblicherweise eine oder mehrere Anfragesequenzen, um die nötigen Eingaben an das System zu beschreiben [51]. Über die Anfragesequenzen und Anfragen ist der Testfall an ein bestimmtes System mit bestimmten Komponenten sowie einer spezifischen Komposition und Komponentenverteilung, welche die Hardware festlegt, gebunden. Wie schon zuvor erläutert (vgl. Kapitel 1) enthält jeder Testfall zudem ein Testkriterium, welches eine Performance-Metrik überprüft. Laut ISO 9126 [35] gibt es die Performance-Metriken Antwortzeit, Durchsatz und Ressourcenverbrauch.

Die einzelnen Spezifikationsartefakte werden von den in Abbildung 2.2 dargestellten Rollen erstellt. Der Systemarchitekt erstellt die Komposition, während der System-Deployer die Komponentenverteilung festlegt. Dagegen entwi-

ckelt der Komponentenentwickler die Komponente. Dabei werden die Elemente Komponente, Komposition und Komponentenverteilung stellvertretend für alle Elemente im Namensraum genannt. Zusammengefasst gelten Systemarchitekt und System-Deployer als Systementwickler. Der Systemtester legt den Testfall fest und damit auch indirekt die Anfragesequenz. Beim Erstellen der Anfragesequenzen wird er vom Domänenexperten unterstützt, der mit seinem Wissen typische Anfrageabläufe spezifizieren kann. Eine detaillierte Erläuterung der Rollen und Zuständigkeiten im Rahmen des Entwicklungsprozesses für komponentenbasierte Systeme ist in Unterabschnitt 2.2.2 zu finden.

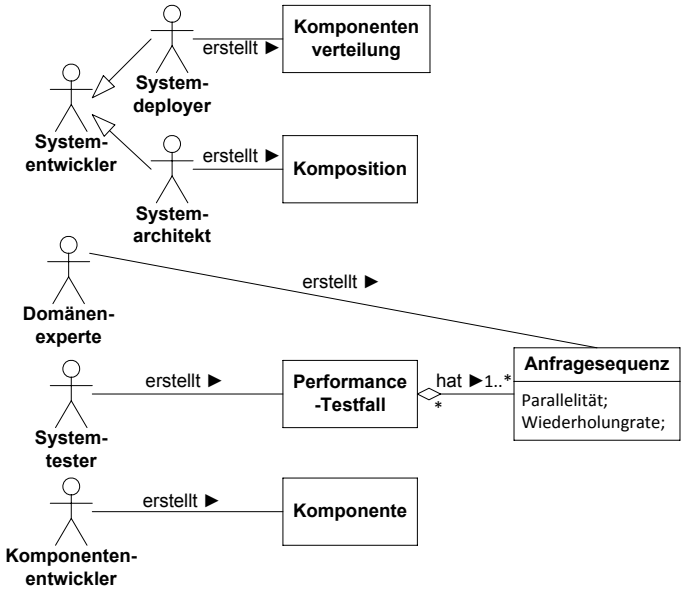

Abbildung 2.2: Wer erstellt welches Spezifikationsartefakt

Die Performance-Metriken geben an, was prinzipiell gemessen werden kann. Wenn ein Testfall ausgeführt wird, werden im Kontext von Beobachtungen Messwerte zu den Metriken ermittelt (s. Abbildung 2.3). Bei der Testfallausführung wird ein Ablaufverfolgungsprotokoll (engl. „trace“) zusammengestellt, das alle Beobachtungen aus dem Testfall sammelt. Zudem erstellt der Tester eine oder mehrere Fehlermeldungen, wenn der Testfall fehlgeschlagen ist und somit ein Fehler im System aufgetreten ist. Die Problematik fehlerhafter Testfälle wird in dieser Arbeit nicht berücksichtigt. Ein Testfall schlägt fehl, wenn das beobachtete Verhalten des Systems nicht das spezifizierte Testkriterium einhält. In dieser Arbeit wird davon ausgegangen, dass die Testkriterien mithilfe der Anforderungen erstellt werden können, da häufig das Verhalten

von Systemoperationen aus Sicht des Systembenutzers getestet wird. Jede Fehlermeldung gibt an, welcher Aspekt des Testkriteriums nicht erfüllt war und zu welcher Zeit bzw. bei welchem Schritt im Testfall der Fehler festgestellt wurde.

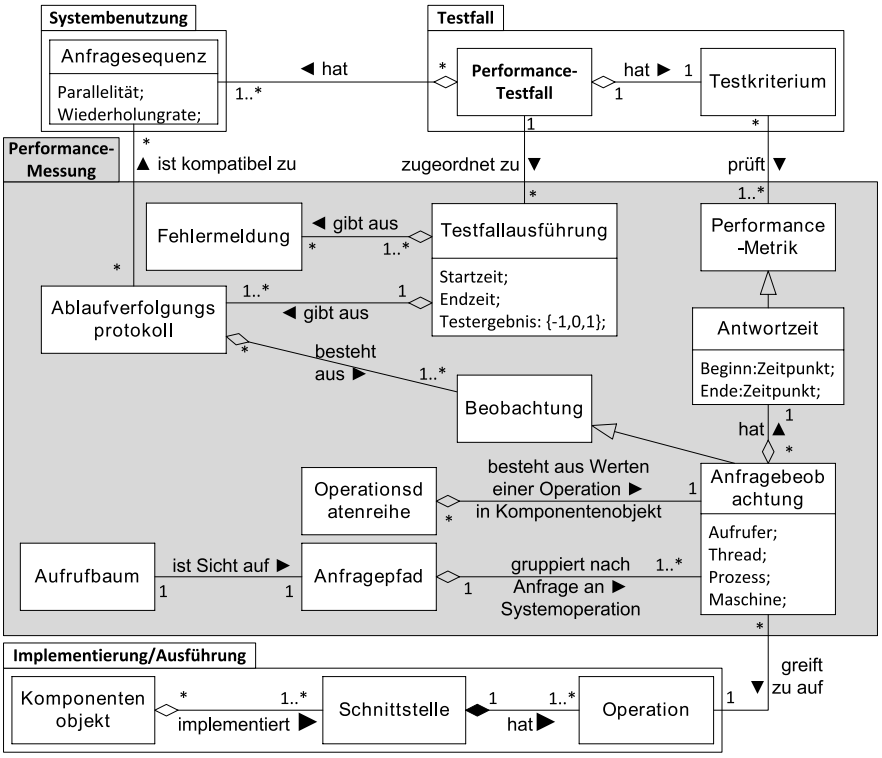

Abbildung 2.3: Begriffe im Bereich Messung im Zusammenhang mit der Spezifikation sowie der Implementierung/Ausführung

Das Ablaufverfolgungsprotokoll, das vom Testfall ausgegeben wurde, enthält alle Beobachtungen, die während des Testlaufs anfallen (s. Abbildung 2.3). Entsprechende Beobachtungen werden analog dazu auch bei der Performance-Analyse bzw. -Simulation gemacht (vgl. Unterabschnitt 2.3.6). Eine Beobachtung kann ein beliebiges Ereignis dokumentieren. Im Bezug auf die Performance beinhaltet sie einen Wert für eine Performance-Metrik. Alternativ lassen sich Performance-Metrikwerte aus den Beobachtungen ableiten. Die Metrik Antwortzeit legt lediglich fest, dass ein Zeitraum gemessen wird, aber nicht worauf genau sich dieser Zeitraum bezieht. Zusätzlich zum Metrikwert enthält eine Performance-Beobachtung deshalb den „Messkontext“. Eine Beobachtung kann beispielsweise die Bearbeitung der Anfrage an eine Komponentenoperation beobachten. Eine Anfrage erfolgt immer im Kontext

einer Operation eines Komponentenobjektes, das als Komponenteneinsatz spezifiziert ist. Bei einer Anfrage wird dieses Komponentenobjekt sowohl über die spezifizierte Operation als auch über die Maschinenkennung sowie die Prozess-ID und die Thread-ID identifiziert. Dies wird in der Klasse Anfragebeobachtung durch die Assoziationen und Attribute wiedergegeben. Die aufgerufene Operation kann dabei eine Systemoperation oder eine normale Komponentenoperation sein. Das Ablaufverfolgungsprotokoll umfasst so auch alle Anfragebeobachtungen, die sich auf von außen getätigten Anfragen an Systemoperationen beziehen, und wird damit einem zuvor definierten Anfrageablauf entsprechen.

Während ein Ablaufverfolgungsprotokoll alle Beobachtungen zu einem ausgeführten Testfall beinhaltet, müssen die Beobachtungen für die Analyse geordnet und gruppiert werden. Dies geschieht im Folgenden beispielhaft für Anfragebeobachtungen (s. Abbildung 2.3). Alle anderen Beobachtungen müssen also ausgefiltert werden. Alle Anfragebeobachtungen, die an eine Operation im selben Komponentenobjekt gerichtet sind, können zu einer Operationsdatenreihe zusammengefasst werden. Alternativ werden sogenannte Anfragepfade gebildet. Ein Anfragepfad gruppiert die Anfragebeobachtungen nach dem Systemoperationsaufruf, der die Anfrage bewirkt hat. In einem Anfragepfad werden die Anfragebeobachtungen nach der zeitlichen und kausalen Abfolge geordnet. Es können so Anfragepfade mit identischer Operationsaufrufreihenfolge aber mit unterschiedlichen Antwortzeiten vorkommen. Der Anfragepfad ist eine Liste von Anfragebeoachtungsobjekten. Abbildung 2.4 zeigt einen Anfragepfad, bei dem jede Zeile für die Beobachtung eines Operationsaufrufs steht. Der abgebildete Anfragepfad ist nach der Beendigung der Anfragen sortiert und bildet die Aufrufbeziehungen mittels ID und Verweis auf die ID in der Spalte „Aufrufer“ ab. Dabei bezeichnet der Wert in der „Aufrufer“-Spalte die ID der aufrufenden Anfragebeobachtung. Aus diesen Informationen lässt sich leicht ein Aufrufbaum erstellen (s. Abbildung 2.4). Der Aufrufbaum stellt eine Sicht auf den Anfragepfad dar, in der er den Anfrageobjekten im Anfragepfad die Information entnimmt, welche Operation von anderen Operationen aufgerufen wird.[1] Die Gruppierung bleibt hier erhalten, es entsteht jedoch ein Baum, der die Aufrufbeziehung der Operationen untereinander explizit widerspiegelt. Im Unterschied zu Operationsdatenreihen kann eine Operation mehrfach im Anfragepfad bzw. Aufrufbaum und auch in mehreren Anfragepfaden bzw. Aufrufbäumen auftauchen. Der Aufrufbaum

[1] Der Verweis auf die Beobachtung des Aufrufs, der diesen Aufruf bewirkt hat, ist nur eine Möglichkeit, die Aufrufhierarchie abzubilden. Eine zweiteilige Abbildung von Aufrufen getrennt nach Eintritt in die Operation und Rückkehr aus der Operation, findet sich ebenfalls häufig.

umfasst jede Anfrage einzeln, was dazu führen kann, dass der Aufrufbaum sehr groß wird. Deswegen wird in Performance-Profilern für die Darstellung stattdessen häufig der Aufrufkontextbaum (engl. „calling context tree“) verwendet [19]. Ein Aufrufkontextbaum fasst im Unterschied zum Aufrufbaum in jedem Knoten Aufrufe derselben Operation zusammen (s. Abbildung 2.4). Wenn also Operation A die Operation B in einer Schleife mehrfach aufruft, so wird im Aufrufkontextbaum an diesem Knoten A vermerkt, dass Operation A Operation B aufruft. Im Aufrufkontextbaum besteht eine gewisse Redundanz, da das Fragment „A ruft B auf“, häufiger wiederkehrt. Der Aufrufgraph legt die fraglichen Knoten zusammen und eliminiert so diese Redundanz (s. Abbildung 2.4). Das kann für verschiedene Analysealgorithmen zur Datenanalyse interessant sein.

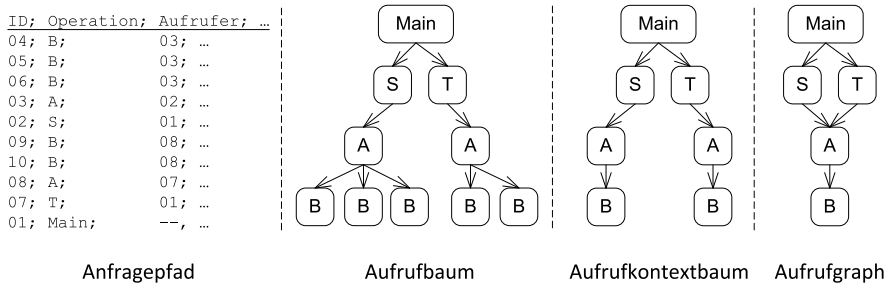

Abbildung 2.4: Verschiedene Auswertungen eines Ablaufverfolgungsprotokolls: (1) der *Anfragepfad* umfasst alle Anfragen, die von einer Anfrage an die Systemoperation „Main“ ausgelöst wurden; (2) der *Aufrufbaum* macht Aufrufbeziehungen zwischen den Operationen explizit; (3) der *Aufrufkontextbaum* fasst gleichartige Aufrufe an eine Operation zusammen; (4) der *Aufrufgraph* fasst redundante Pfade zusammen.

Der Bereich der Implementierung/Ausführung wird in Abbildung 2.5 näher dargestellt. Hier wird unter anderem der Komponentenbegriff erweitert, wobei Abbildung 2.5 die Operationen der Übersichtlichkeit halber auslässt. Für die Komponentenobjekte sind die Schnittstellen und Operationen schon in Abbildung 2.3 zu sehen. Im Bereich der Implementierung/Ausführung beinhalten die Komponenten die Operationen als kompilierten Quelltext oder Maschinencode. Dabei wird der Komponentenbegriff von Cheesman und Daniels [15] wiedergegeben, bei dem die Komponentenimplementierung ein oder mehrfach auf der Hardware installiert wird und dann bei Ausführung ein- oder mehrfach instanziiert wird (vgl. Unterabschnitt 2.2.1). Demgegenüber steht die Spezifikation, bei der die Komponente und die dazu gehören-

den Komponenteneinsätze unterschieden werden. Dabei kann es zu einer Komponentenspezifikation mehrere Komponentenimplementierungen geben. Einem Komponenteneinsatz aus dem Spezifikationsbereich steht hingegen genau ein Komponentenobjekt gegenüber. Dementsprechend darf der Komponenteneinsatz nur genau einmal (je Komponentenverteilung) auf einem Hardwareknoten verteilt werden. Ebenso darf ein Komponentenobjekt nur von den Prozessoren genau eines Servers ausgeführt werden. Die Begriffe Prozessor und Server sind Beispiele für die Realisierung von Ressourcen und Hardwareknoten aus der Spezifikation. Hier wird davon ausgegangen, dass jedes Komponentenobjekt direkt auf den Server-Ressourcen ausgeführt wird. In der Realität ist oftmals eine spezielle Ausführungsumgebung nötig, um Komponenten auszuführen. Dabei wird die Komponente innerhalb der Ausführungsumgebung ausgeführt und die Ausführungsumgebung läuft auf dem Betriebssystem des Servers. Dies müsste sowohl im Bereich Implementierung/Ausführung als auch im Bereich Spezifikation berücksichtigt werden.

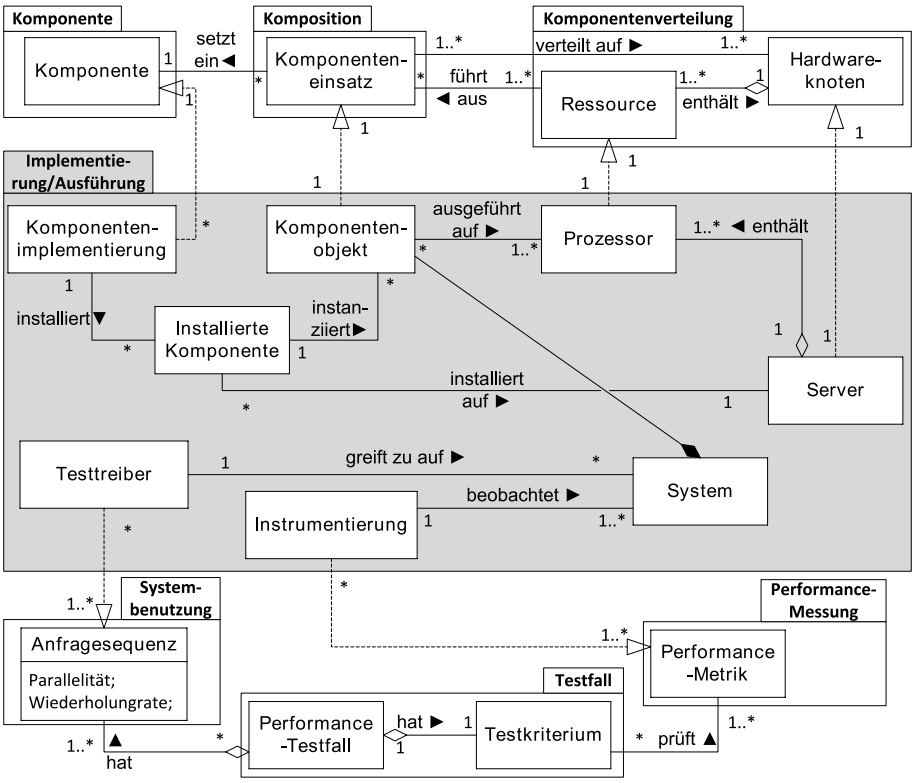

Abbildung 2.5: Begriffe im Bereich Implementierung im Zusammenhang mit der Spezifikation sowie der Performance-Messung

Des Weiteren wird in Abbildung 2.5 das komponentenbasierte System dargestellt. Beim Begriff System ist das ausgeführte System gemeint. Dieses System besteht aus einem Verbund von Komponentenobjekten, die auf einen oder mehrere Server verteilt sind. Das System realisiert so implizit die Komposition und die Komponentenverteilung. Das System wird im Performance-Test von einem Testtreiber verwendet, der die im Testfall spezifizierten Abläufe von Anfragen abarbeitet. Während des Tests werden die nötigen Messungen von der Instrumentierung des Systems vorgenommen. Eine Instrumentierung liefert das Messwerkzeug, mit dem eine oder mehrere unterschiedliche Performance-Metriken beobachtet werden können. Dabei ist die Instrumentierung meist technologieabhängig [51] und kann beispielsweise nur für C-, Java- oder CORBA-Programme verwendet werden. Eine bestimmte Instrumentierung lässt sich häufig für alle Systeme, die mit dieser Technologie entwickelt wurden, einsetzen.

Die einzelnen Artefakte im Bereich Implementierung/Ausführung werden von den in Abbildung 2.6 dargestellten Rollen erstellt. Der Komponentenentwickler liefert die komplette Komponente mit Spezifikation und Implementierung. Der System-Deployer legt nicht nur die Komponentenverteilung fest, sondern setzt sie auch um, indem er die Komponentenimplementierungen auf der vorgesehenen Hardware installiert. Damit sorgt der System-Deployer dafür, dass ein ausführbares System vorliegt. Schließlich wird der Tester den Testtreiber erstellen, das System mit der für den Testfall nötigen Instrumentierung ausstatten und den Test durchführen. Bei der Testdurchführung wird dann das System ausgeführt, wobei die installierten Komponenten zu Komponentenobjekten instanziiert werden. Während der Testausführung erstellt die Instrumentierung automatisch das Ablaufverfolgungsprotokoll. Die Analyse des Protokolls obliegt der Performance-Blame-Analysis. Dabei werden nach Bedarf Operationsdatenreihen, Anfragepfade, Aufrufbäume, Aufrufkontextbäume und Aufrufgraphen gebildet und weitere darauf aufbauende Metriken ermittelt.

2.2 Komponentenbasierte Systeme

Komponentenbasierte Systeme werden aus einzelnen Komponenten aufgebaut. Komponentenbasierte Systeme sind immer modular aufgebaut, da die Komponenten in sich abgeschlossene Einheiten sind, auf die nur über Schnittstellen zugegriffen werden kann. Generell weisen Komponenten nur explizit über Schnittstellen definierte Abhängigkeiten auf. Solche Komponenten lassen sich leicht austauschen, wodurch komponentenbasierte Systeme leicht

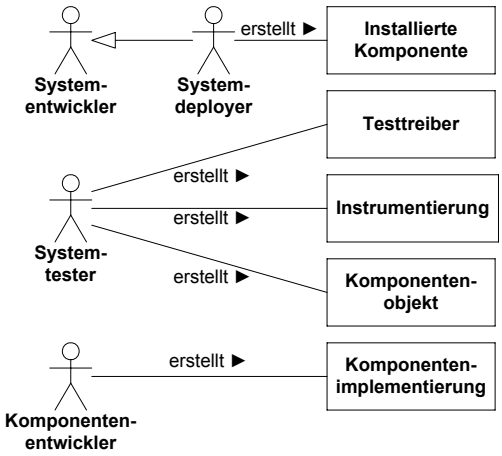

Abbildung 2.6: Wer erstellt welches Artefakt im Bereich Implementierung/-Ausführung

weiterentwickelt und angepasst werden können. Zudem können solche Komponenten auch leicht wiederverwendet werden. So werden Produktivitätsgewinne durch die Wiederverwendung als solche und durch Qualitätssicherungsmaßnahmen ermöglicht, die allen Verwendern (Systementwicklern) einer Komponente zugutekommen. Die Definition des Begriffs „Komponente“ und die Klarstellung der wichtigsten Eigenschaften von Komponenten für diese Arbeit erfolgen im Unterabschnitt 2.2.1.

Da komponentenbasierte Systeme nur aus Komponenten bestehen, lässt sich das System nach der Beschaffung der Komponenten schnell zusammensetzen und realisieren. Komponenten können entkoppelt von komponentenbasierten Systemen erstellt werden, so dass der größte Teil der Implementierungsleistung schon abschlossen sein kann, bevor das komponentenbasierte System entworfen wird. In diesem günstigen Fall kann das System selbst sehr schnell realisiert werden. Dies setzt aber einen Markt voraus, über den man alle nötigen Komponenten beziehen kann. Der Entwicklungsprozess für komponentenbasierte Systeme aus Sicht der Systementwicklung wird in Unterabschnitt 2.2.2 genau beleuchtet.

2.2.1 Komponentenbegriff

Die Definition des Begriffs „Komponente“ ist nicht einfach, da er in vielen unterschiedlichen Kontexten für unterschiedliche Dinge verwendet wird. Die-

se Arbeit orientiert sich zunächst an der bekannten Definition von Clemens Szyperski et al. [71]:

> „A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Nach dieser Definition ist eine Komponente eine Einheit, die mit anderen Komponenten zu einem System (oder einer neuen Komponente) zusammengesetzt, also komponiert, werden kann. Dabei sollen sich Komponenten lediglich auf Schnittstellen zu anderen Komponenten verlassen. Diese Schnittstellenspezifikation stellt also eine vertragliche Vereinbarung einer Komponente mit ihren Benutzerkomponenten dar. Diese Vereinbarungen können sowohl die funktionalen als auch die nichtfunktionalen Eigenschaften der verwendeten Komponente zum Gegenstand haben. Wichtig ist in diesem Zusammenhang auch, dass die Komponente ihre Erwartungen an die Umgebung vollständig explizit macht. Denn nur so kann die Komponente in ein System integriert werden, ohne dass für ihre Verwendung Wissen über die Interna der Komponente nötig ist. Dies ist insbesondere dann wichtig, wenn der Quelltext der Komponente nicht vorliegt. Das kommt vor allem dann vor, wenn die Komponente von Dritten wiederverwendet wird, also wenn die Komponente z. B. über einen Markt an (für den Komponentenhersteller) anonyme Kunden vertrieben wird. Bei einer solchen Komponente wird besonders deutlich, dass Komponenten ihre Implementierung genauso kapseln wie ihren Zustand zur Ausführung. Sie verbergen diese Interna hinter ihren Schnittstellen. Somit ist ein komponentenbasiertes System immer auch modular aufgebaut.

Zuletzt soll eine Komponente laut Szyperski et al. [71] auch unabhängig ausgeliefert werden können. Das bezieht sich darauf, dass die Komponenten über die Schnittstellen nur lose an andere Komponenten gekoppelt sind. So kann eine Komponente einzeln ausgeliefert werden und überall dort installiert und eingesetzt werden, wo ihre angebotenen Schnittstellen gefordert und ihre benötigten Schnittstellen erfüllt werden. Gao et al. [26] kennen neben solchen „wiederverwendbaren Komponenten" (engl. „reusable components") auch sogenannte „Kompositbausteine" (engl. „composite building blocks"), die zusätzlich zu den Eigenschaften einer wiederverwendbaren Komponente mithilfe eines Komponentenmodells definiert sind. Laut Gao et al. bildet ein Komponentenmodell das Rückgrat eines komponentenbasiert Systems. Es gibt vor, wie Komponenten zu implementieren, zu komponieren und auszuführen sind. Dabei übernehmen die Komponentenmodelle häufig auch die Kommunikation zwischen den Komponenten eines Systems. Gao et al.

nennen „.NET“ bzw. „COM“ von Microsoft, „CORBA“ von der Object Management Group sowie „Enterprise Java Beans“ (EJB) von Oracle (vormals Sun Microsystems) als die bekanntesten Komponentenmodelle. Neben diesen implementierungsorientierten Komponentenmodellen nennt Becker [4] noch dokumentenorientierte Komponentenmodelle, die zur Spezifikation genutzt werden. Darüber hinaus erlauben es diese Komponentenmodelle, die Spezifikation der Komponenten bzw. der Komposition auf ihre Eigenschaften zu analysieren und Schlüsse über ihre Laufzeiteigenschaften daraus zu ziehen. Beispiele für solche dokumentenorientierten Komponentenmodelle sind die UML [53] ggf. mit der MARTE-Erweiterung [52] für Performance-Aspekte und das Palladio Component Model [8].

Wie mit den Komponentenmodellen schon angeklungen ist, muss auch darüber nachgedacht werden, wie Komponenten von ihrer Spezifikation zur Ausführung gebracht werden. Während Cheesman und Daniels [15] keine genaue Definition des Begriffs Komponente geben, so identifizieren sie doch verschiedene Komponentenartefakte, die von der Spezifikation bis zur Ausführung auftreten. Abbildung 2.7 zeigt die unterschiedlichen Komponentenartefakte und ihre Beziehungen untereinander. Diese Unterscheidung von Komponentenartefakten zeigt sich auch in den grundlegenden Begriffen dieser Arbeit (vgl. Abschnitt 2.1), besonders für den Bereich der Implementierung und Ausführung (s. Abbildung 2.5). Die Komponentenspezifikation umfasst die unterstützten Schnittstellen, sowohl angebotene und benötigte Schnittstellen. Eine Komponentenspezifikation kann durch verschiedene Komponentenimplementierungen realisiert werden, die dann auf dem Markt zueinander in Konkurrenz treten. Eine Komponentenimplementierung wird dann auf verschiedenen Hardwareknoten (bzw. Laufzeitumgebungen) installiert. Die installierten Komponenten können dann schließlich jeweils mehrfach instanziiert werden. Dabei gehen Cheesman und Daniels davon aus, dass Komponentenobjekte die Eigenschaften gewöhnlicher Objekte aus der objektorientierten Programmierung teilen. Das heißt insbesondere, dass die Objekte unabhängig von ihrem Zustand identifiziert werden können. Darüber hinaus können unterschiedliche Komponentenobjekte einer installierten Komponente auch unterschiedliche Konfigurationen haben.

Im Rahmen dieser Arbeit werden die Komponenten in der Performance-Blame-Analysis alle so behandelt, als ob sie von Dritten stammen würden. Die Komponenten kapseln also ihre Funktionalität im Bezug auf Daten und Funktion als Blackbox. Somit ist entscheidend, dass für jede Komponente korrekte funktionale Schnittstellen (angebotene und benötigte) vorliegen. Dies erlaubt die Komposition der Komponenten über die Schnittstellen. Neben dieser Grundbedingung soll insbesondere für jede Komponentenoperation

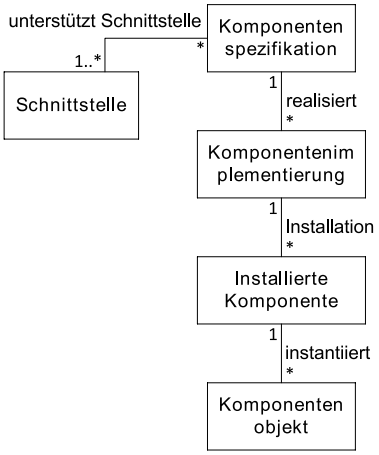

Abbildung 2.7: Komponentenartefakte und deren Beziehung zueinander nach Cheesman und Daniels [15]

auch ein Performance-Kontrakt vorliegen, um den in dieser Arbeit vorgestellten Ansatz anwenden zu können. Der Performance-Kontrakt soll mithilfe eines (dokumentenorientierten) Komponentenmodells formuliert sein, das eine stochastische Analyse bzw. eine Simulation der Spezifikation erlaubt und so die Ausgabe von Antwortzeitwerten ermöglicht. Idealerweise sollten die Performance-Kontrakte dem Systemarchitekten Raum geben, die Kontrakte auf den Kontext der Komponente anzupassen. Denn der Komponentenentwickler kann nicht wissen, in welchem Kontext die Komponente zum Einsatz kommt. Des Weiteren wird im Rahmen dieser Arbeit angenommen, dass jedes Komponentenobjekt eindeutig identifiziert werden kann. Dies erleichtert es, die Komponentenobjekte zur Laufzeit zu beobachten und die beobachteten Ereignisse mit Bezug auf das Komponentenobjekt und seinen Kontext zu dokumentieren.

2.2.2 Entwicklungsprozess

Cheesman und Daniels [15] haben aufbauend auf dem Rational Unified Process (RUP) [46] einen komponentenbasierten Entwicklungsprozess beschrieben, der die besondere Vorgehensweise bei komponentenbasierten Systemen widerspiegelt. Mit dem Rational Unified Process dient hier ein praxiserprobter Prozess als Grundlage. Der Rational Unified Process umfasst dabei sowohl die Softwareentwicklung als auch den zugehörigen Managementprozess. Hier wird ausschließlich der Bereich der Softwareentwicklung beleuchtet, bei der

die nötigen Arbeitsabläufe und Artefakte zur Erstellung der Software beschrieben werden. Der Rational Unified Process unterstützt iterative und inkrementelle Softwareentwicklung. Diese Eigenschaften übertragen sich auf die darauf aufbauenden Prozesse.

Der Prozess von Cheesman und Daniels wurde später von Koziolek und Happe [41] erweitert, so dass der Prozess auch nichtfunktionale Eigenschaften beachtet. Der Prozess kann dabei nicht nur für die besondere Beachtung der Performance verwendet werden, sondern für alle nichtfunktionalen Eigenschaften z. B. auch für die Zuverlässigkeit. Diese Arbeit geht aber ausschließlich auf die Performance ein. Der Prozess von Koziolek und Happe ist in Abbildung 2.8 zu sehen. Die Kästen in der Grafik stellen Arbeitsabläufe dar. Die dicken Pfeile zeigen, dass zwischen den Arbeitsabläufen vor- und zurückgewechselt werden kann. Die dünnen Pfeile zeigen den Fluss von Artefakten zwischen den Arbeitsabläufen an. Die Arbeitsabläufe „Anforderungen“, „Test“ sowie „Verteilung und Auslieferung“ werden weitgehend unverändert aus dem RUP [46] übernommen. Dabei werden die Arbeitsabläufe jedoch um den Umgang mit Performance-Aspekten erweitert. Darüber hinaus haben Happe und Koziolek den Entwicklungsprozess um einen „QoS-Analyse“-Arbeitsablauf erweitert, bei dem aufgrund der Komponentenspezifikation und Komposition eine modellbasierte Performance-Analyse durchgeführt wird. Zuletzt haben Sie auch ein Rollenmodell definiert und klar festgelegt, welche Rolle an welchem Arbeitslauf beteiligt ist. Im Folgenden werden die einzelnen Arbeitsabläufe einzeln vorgestellt. Dabei wird kurz vorgestellt, welche Rolle welche Aktivitäten im jeweiligen Arbeitsablauf übernimmt und welche Ein- und Ausgabeartefakte zum jeweiligen Arbeitsablauf gehören.

Arbeitsablauf: Anforderungen

Bei der Ermittlung der Anforderungen gilt es zum einen den Problembereich zu erfassen und so zu beschreiben, dass alle Beteiligten ein gemeinsames Verständnis erlangen. Zum anderen werden Anwendungsfälle erarbeitet, die die Anforderungen an das System festlegen. Dabei sollten auch Performance-Anforderungen erarbeitet werden. Die Anforderungen werden vom Domänenexperten ermittelt, da sein Fachwissen hierfür unverzichtbar ist.

Arbeitsablauf: Spezifikation

Die Spezifikation setzt auf dem Problembereichsmodell und den Anwendungsfällen auf und resultiert in den Komponentenspezifikationen und in der Kom-

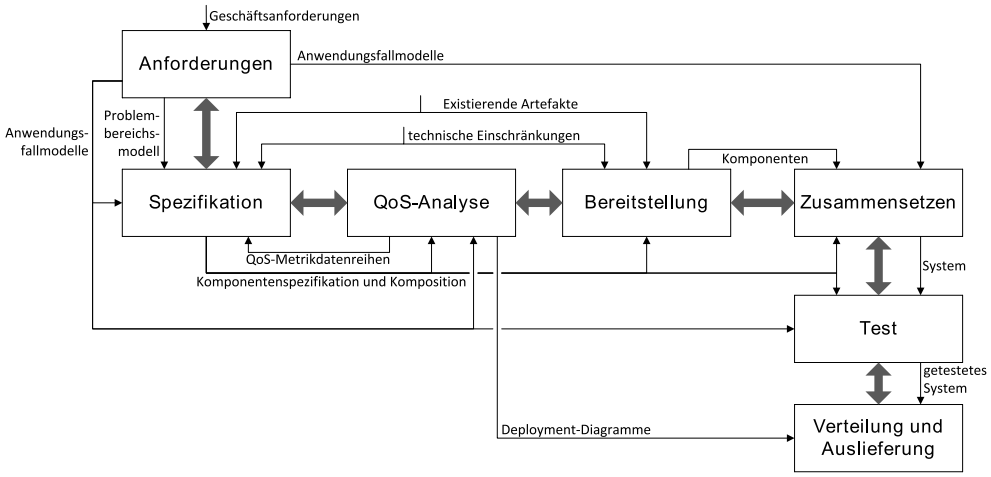

Abbildung 2.8: Entwicklungsprozess nach Koziolek und Happe [41]

position. Dabei spielt eine Rolle, welche Komponenten schon existieren. Auch technische Einschränkungen durch die Philosophie der Organisation oder das Wissen der Projektbeteiligten müssen hier berücksichtigt werden. Die Spezifikation wird vom Systemarchitekten angefertigt. Der Systemarchitekt identifiziert dabei zunächst, welche Komponenten es im System geben sollte und wie diese miteinander interagieren sollen. Dann legt er die Komponentenspezifikation im Detail fest. Zuletzt prüft er mithilfe von Analysemodellen, ob die Komponenten in der erstellten Architektur wie gewünscht interagieren. Hierbei werden auch die Erkenntnisse aus der QoS-Analyse einbezogen.

Arbeitsablauf: QoS-Analyse

In diesem Arbeitsablauf wird die geplante Architektur des Systems verfeinert. Dazu wird zunächst ein umfassendes Analysemodell erstellt. Der System-Deployer spezifiziert die Hardware- und Softwareressourcen des Systems und ordnet diese Ressourcen den Komponentenobjekten zu. Damit erstellt der System-Deployer die Deployment-Diagramme, die während der Auslieferung des Systems (und dem Test) weiterverwendet werden. Zudem modelliert der Domänenexperte das zu erwartende Benutzerverhalten in einem Verwendungsmodell. Er stützt sich bei der Entwicklung des Verwendungsmodells auf die zuvor festgelegten Anwendungsfälle. Der Komponentenentwickler liefert für seine Komponenten ein Modell des erwarteten Ressourcenverbrauchs in Abhängigkeit vom Kontext. Die Spezifikation vom Komponentenentwickler fließt über das Artefakt Komponentenspezifikation aus dem Arbeitsablauf

Spezifikation in die QoS-Analyse ein. Im nächsten Schritt werden die genannten Modelle in ein umfassendes Analysemodell integriert. Dies nimmt laut Happe und Koziolek [41] der Systemarchitekt vor. Becker [4] hat später noch die Rolle des QoS-Analysten hinzugefügt, der die Modellintegration und die weitere Analyse bzw. Simulation des Analysemodells übernimmt, da hierfür spezielles Wissen hilfreich ist. Dieses Analysemodell soll nun simuliert oder mithilfe stochastischer Methoden berechnet werden. Die Ergebnisse der Simulation oder Berechnung vergleicht der Systemarchitekt bzw. QoS-Analyst nun mit den nichtfunktionalen Anforderungen. Zuletzt erarbeitet der Systemarchitekt ggf. alternative Vorschläge für die Systemarchitektur. Im Rahmen dieser Arbeit nehmen wir an, dass dieser Arbeitsablauf mithilfe des Palladio Component Model (PCM) realisiert wird. Das PCM wird in Abschnitt 2.3 näher erläutert.

Arbeitsablauf: Bereitstellung

Bei der Bereitstellung sorgt der Systemarchitekt dafür, dass alle geplanten Komponenten auch zur Verfügung stehen. Er kann dabei die existierenden Komponenten berücksichtigen. Die fehlenden Komponenten muss er auf dem Markt einkaufen oder Projekte zu deren Erstellung innerhalb seiner Organisation anstoßen bzw. deren Fertigung außer Haus in Auftrag geben. Die Komponenten selbst werden im Rahmen eines anderen entkoppelten Softwareerstellungsprozesses von den Komponentenentwicklern erstellt. Dabei müssen die Komponentenspezifikation sowie die bereits zuvor angesprochenen technischen Einschränkungen berücksichtigt werden. Zur Bereitstellung gehört natürlich auch, dass der Systemarchitekt sicherstellen muss, dass die Komponenten den gewünschten Spezifikationen entsprechen. Dabei kann es nötig sein, dass der Systemarchitekt hier Kompromisse zulässt, da nicht jede Technologie die Spezifikationen eins zu eins umsetzen kann. Bei derartigen Änderungen sollte zumindest im Rahmen der QoS-Analyse geprüft werden, ob nichtfunktionale Anforderungen gefährdet werden. Um die Konsistenz der Komponenten mit der Spezifikation zu prüfen, kann der Systemarchitekt auch Tests ausführen, mit denen er nach entsprechenden Abweichungen sucht. Geeignet dazu sind Unit-Tests oder ggf. Integrationstests. Der Arbeitsablauf Bereitstellung resultiert in den implementierten Komponenten.

Arbeitsablauf: Zusammenstellen

Beim Zusammenstellen integriert der Systemarchitekt die Komponenten sowie schon existierende Komponenten und Systeme, wie in der Komposition

beschrieben, zu einem System. Dabei muss er darauf achten, dass das resultierende System die in den Anwendungsfällen beschriebenen Szenarien und Anforderungen umsetzen kann. Laut Cheesman und Daniels [15] gehört zu diesem Arbeitsablauf auch, dass die Benutzeroberfläche für das System realisiert wird. Das Resultat dieses Arbeitsablaufs ist das ausführbare System.

Arbeitsablauf: Test

Beim Test wird das ausführbare System oder ein Teil des Systems in einer Testumgebung installiert auf seine Qualität überprüft. Dazu werden Testfälle entworfen und durchgeführt, die eine bestimmte Qualitätseigenschaft des Systems oder Systemteils prüfen. Der Test ist ein Arbeitsablauf der die übrigen Arbeitsabläufe unterstützt. Die Tester hinterfragen in diesem Arbeitsablauf bisher vorliegende Entwicklungsergebnisse und untersuchen diese auf ihre Qualität [46]. Die Qualitätskontrolle prüft sowohl, ob die funktionalen und nichtfunktionalen Anforderungen eingehalten werden, als auch, ob die Entwicklungsartefakte konsistent aufeinander aufbauen. Dazu werden verschiedene Teststufen unterschieden: der Unit-Test, der Integrationstest, der Systemtest und der Akzeptanztest [46, 69]. Bei komponentenbasierten Systemen wird der Unit-Test meist schon aufseiten der Komponentenentwickler durchgeführt. Nach der Beschaffung der Komponenten kann sich der Systemarchitekt dazu entschließen, die Komponenten mithilfe von Unit-Tests einzeln auf ihre Qualität zu prüfen. Die Integrationstests werden parallel zum Zusammensetzen des Systems durchgeführt, da hier die zuvor beschafften Komponenten zu einem System integriert werden. Nach dem Zusammensetzen des Systems überprüft der Systemtest schließlich das Gesamtsystem auf seine Qualität. Zuletzt wird der Abnahmetest im Rahmen der Verteilung und Auslieferung durchgeführt. In dieser Teststufe überprüfen Endbenutzer die Qualität des Gesamtsystems.

Nach Cheesman und Daniels [15] richten sich die Tester nach den Anwendungsfällen und den Komponentenspezifikationen, um ihre Testfälle auszuarbeiten. Happe und Koziolek [41] führen Deployment-Diagramme als zusätzliches Artefakt ein, das aber erst bei der Verteilung und Auslieferung berücksichtigt wird. Dagegen schlagen Spillner und Linz [69] vor alle Spezifikationsdokumente, aus denen Anforderungen an das spätere Produktivsystem hervorgehen, für den Test zu berücksichtigen. Mithilfe der Deployment-Diagramme kann die Testumgebung die spätere Produktivumgebung abbilden oder wenigstens annähern, so dass die Deployment-Diagramme wichtige Informationen für einen Test beinhalten.

Beim Test werden verschiedene funktionale und nichtfunktionale Qualitätsmerkmale überprüft. Unter anderem werden auch Performance-Tests durchgeführt. Performance-Tests betreffen die Teststufen Integrationstest und Systemtest [51]. Wobei der Schwerpunkt auf dem Systemtest liegt. Näheres zu Performance-Tests kann Abschnitt 2.4 entnommen werden.

Wenn bei der Testdurchführung Fehler gefunden werden, stellt der Tester dies zunächst in Fehlermeldungen fest. Daraufhin muss der Fehler analysiert werden, um zu entscheiden, ob und von wem er mit welcher Priorität beseitigt werden soll. Dieses Vorgehen ist sowohl bei Spillner und Linz [69] als auch Kruchten [46] vorgesehen. Kruchten spricht davon, dass aufgrund der Testergebnisse Änderungsaufträge an das System aufgenommen werden. Bei der Erstellung der Änderungsaufträge wird die angesprochene Fehleranalyse durchgeführt. Bei komponentenbasierten Systemen entspricht die Fehleranalyse einer Blame-Analysis. Speziell bei Performance-Fehlern spricht man von einer Performance-Blame-Analysis. Bei komponentenbasierten Systemen muss eine Performance-Blame-Analysis klären, ob die System- oder Komponentenentwickler betroffen sind. Falls die Komponentenentwickler den Fehler beseitigen sollen, muss auch entschieden werden, welche Komponentenentwickler es im Einzelnen trifft. Die Einzelheiten, welche Aktivitäten die Performance-Blame-Analysis ausmachen und wie sie mit dem Test-Arbeitsablauf verbunden ist, finden sich in Abschnitt 3.1.

Arbeitsablauf: Verteilung und Auslieferung

Zuletzt wird das System an den Kunden ausgeliefert. In dieser Phase muss das System zu einem Produkt gemacht werden. Dazu gehört unter anderem das System in ein Installationspaket zu integrieren, das die Installation beim oder durch den Kunden (je nach System) möglichst einfach gestaltet. Es gehört aber auch eine Installations- und Benutzerdokumentation sowie Kursunterlagen für die etwaige Schulung der Benutzer dazu. Zu dieser Phase gehören auch Akzeptanztests, bei denen die Endbenutzer das System testen und so ein letztes Feedback geben können, bevor das System zur endgültigen Installation bereit ist. Um sinnvoll auf das Feedback reagieren zu können, empfiehlt sich ein iteratives Vorgehen, bei dem die Software mehrmals mit jeweils gewachsenem Funktionsumfang ausgeliefert wird.

Zuletzt installiert der System-Deployer das System in der Produktivumgebung und versetzt es in einen lauffähigen Zustand. Der System-Deployer installiert die Komponenten dabei zunächst auf den verschiedenen verfügbaren Hardwareknoten (bzw. den dort verfügbaren Laufzeitumgebungen). Dann

müssen ggf. Daten aus einem alten System migriert werden oder anderweitig initiale Daten eingepflegt werden. Schließlich kann das System ausgeführt werden.

2.3 Palladio Component Model (PCM)

Wie bereits in Unterabschnitt 2.2.2 beschrieben, lässt sich die Vorhersage von Laufzeiteigenschaften des zu entwickelnden komponentenbasierten Systems in den Entwicklungsprozess integrieren. Dazu haben Happe und Koziolek [41] den Arbeitsablauf „QoS-Analyse“ eingeführt, in dem ein Analysemodell arbeitsteilig erstellt und analysiert wird. Die Ergebnisse dieses Arbeitsablaufs können dann dazu verwendet werden, den Entwurf des komponentenbasierten Systems zu verbessern. Zuvor haben wir jedoch nicht genau erwähnt, welche Modelle sich für die Art der Analyse eignen. Im Folgenden wird daher das Palladio Component Model (PCM) eingeführt, mit dem sich die Performance und die Zuverlässigkeit eines zu entwickelnden komponentenbasierten Systems vorhersagen lassen. Diese Arbeit beschränkt sich dabei auf die Performance-Modellierung und -Analyse mithilfe des PCM.

Der Entwicklungsprozess von Komponenten und komponentenbasiertem System läuft entkoppelt ab. Daher müssen auch die Modelle von den unterschiedlichen Beteiligten am Entwicklungsprozess entworfen werden. Der Komponentenentwickler entwickelt zwei Modelle. Zunächst entwickelt er eine funktionale Spezifikation, die angibt, welche Schnittstellen seine Komponente anbietet, und benötigt. Derartige Spezifikationen für eine oder mehrere Komponenten werden im Artefakt „Komponenten-Repository“ (s. Unterabschnitt 2.3.1) des PCM gesammelt. Als Zweites modelliert er auch einen Performance-Kontrakt, dem zu entnehmen ist, welche Performance seine Komponente liefert. Im Rahmen dieser Arbeit wird davon ausgegangen, dass die Komponentenentwickler zusammen mit ihrer Komponentenimplementierung sowohl die Spezifikation der funktionalen Schnittstellen, also das Komponenten-Repository, als auch die hier angesprochenen Performance-Kontrakte ausliefern. Das PCM sieht als Performance-Kontrakt ein (abstrahiertes) Kontrollflussmodell der jeweiligen Komponentenoperation vor, die sogenannte Service-Effect-Specification (SEFF; s. Unterabschnitt 2.3.2). Die SEFFs stellen einen Performance-Kontrakt dar, in dem dort jeder Berechnung ein Ressourcenverbrauch zugeordnet werden kann. Dabei bezieht sich ein SEFF mithilfe von Parametern [58] auf den Kontext der Komponente [6]. So kann der Systemarchitekt den Kontext der Komponente später modellieren und so konkrete Werte für die Parameter des SEFFs bereitstellen.

Der Systemarchitekt bindet die Komponenten aus verschiedenen Quellen in ein Systemmodell (engl. „assembly model“; s. Unterabschnitt 2.3.3) ein. Im Systemmodell wird beschrieben, welche Komponentenobjekte im System vorhanden sind und wie sie miteinander gekoppelt sind. Der System-Deployer erstellt ein Ressourcenmodell (engl. „ressource environment“; s. Unterabschnitt 2.3.4), in dem die unterschiedlichen Hardwareknoten mit ihren vorhandenen Ressourcen spezifiziert sind. Darauf aufbauend ordnet der System-Deployer die Komponentenobjekte aus dem Systemmodell den Hardwareknoten zu. Dies wird im Verteilungsmodell (engl. „allocation model“; s. Unterabschnitt 2.3.4) beschrieben. Zuletzt spezifiziert der Domänenexperte anhand seiner Erfahrung das zu erwartende Benutzerverhalten und die voraussichtliche Benutzerlast im Verwendungsmodell (engl. „usage model“; s. Unterabschnitt 2.3.5).

Die vorgestellten Diagramme weisen zahlreiche Parallelen zu den Grundbegriffen aus Abschnitt 2.1 auf, daher werden in Unterabschnitt 2.3.7 beide Modelle miteinander verglichen. Die Diagrammarten des PCM werden zuvor in den Unterabschnitten 2.3.1 bis 2.3.5 genauer anhand des Beispiels aus Abschnitt 1.2 erläutert.

2.3.1 Komponenten-Repository

Das Komponenten-Repository wird vom Komponentenentwickler zur Verfügung gestellt. Er spezifiziert darin die von ihm erstellten oder vertriebenen Komponenten. Das Komponenten-Repository kann aber auch vom Systemarchitekten zum Entwurf der gewünschten Komponenten verwendet werden. Im Komponenten-Repository werden die Komponenten mit ihren angebotenen und benötigten Schnittstellen dargestellt. Das Komponenten-Repository in Abbildung 2.9 veranschaulicht dies. Dort werden die Komponenten `Store`, `StoreQuery`, `ProductDispatcher` und `EnterpriseQuery` spezifiziert. Dabei bietet die `Store`-Komponente die Schnittstelle `StoreInterface` an und benötigt die Schnittstellen `StoreQueryInterface` und `ProductDispatcherInterface`, die von den Komponenten `StoreQuery` und `ProductDispatcher` angeboten werden. In den Schnittstellen sind auch die Operationen mit ihren Signaturen spezifiziert.[2] Neben dem Operationsnamen sind auch die Ein- und Ausgabeparameter angegeben. Das PCM ermöglicht es sogar Datentypen zu spezifizieren. Dabei können einfache Typen als Namen berücksichtigt werden. Datenverbünde bestehen aus verschiedenen benannten Feldern beliebigen Typs. Zudem gibt es Listentypen, die eine Liste aus Elementen eines bestimmten Typs repräsentieren. Es ist

[2] In Abbildung 2.9 sind die Signaturen aus Platzgründen teilweise gekürzt.

allerdings keine Diagrammdarstellung zur Erstellung und Verwaltung von Datentypen vorhanden.

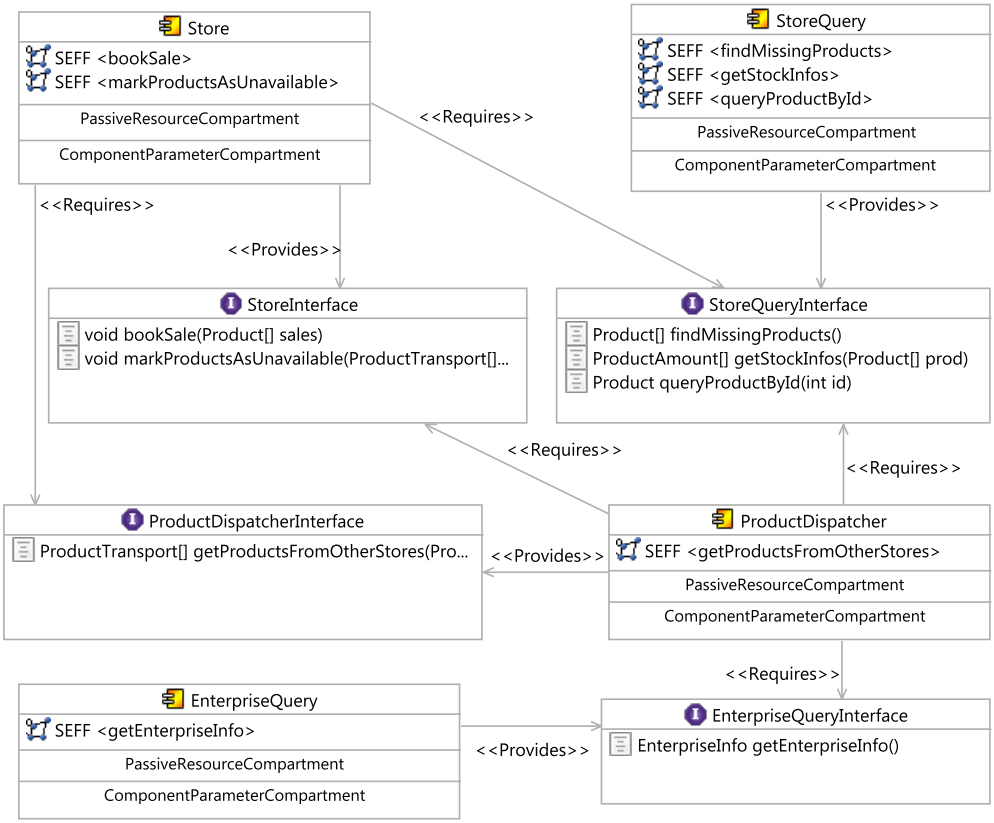

Abbildung 2.9: Beispiel für ein Komponenten-Repository

2.3.2 Service-Effect-Specification (SEFF)

Neben dem Komponenten-Repository geben die Komponentenentwickler auch eine Zusicherung über die Performance ihrer Komponenten ab. Dazu modellieren sie den Ressourcenverbrauch jeder Komponentenoperation entlang ihres (abstrahierten) Kontrollflusses in einer SEFF. Die SEFFs sind das Herzstück der Performance-Kontrakte. Die in SEFFs dargestellten Kontrollflüsse können bedingte Abläufe, Schleifen und parallele Abläufe enthalten. Innerhalb des Kontrollflusses können Berechnungen angestellt werden oder aber andere Operationen aufgerufen werden. Interne Berechnungen können dabei direkt Ressourcen verbrauchen. Dies kann zusätzlich über sogenannte

passive Ressourcen, wie beispielsweise einen Thread-Pool, gesteuert werden, die eine bestimmte Anzahl an Marken bereithalten. Diese Marken müssen reserviert und wieder freigegeben werden, wodurch in der späteren Simulation bzw. Analyse Wartezeiten auftreten können.

In Abbildung 2.10 wird eine SEFF für die Komponentenoperation `markProductUnavailable` aus der `Store`-Komponente dargestellt (vgl. Abbildung 2.9). Die Operation wird von der Zentrale aufgerufen, um einer Filiale mitzuteilen, dass verschiedene Produkte aus ihrem Lagerbestand an eine andere Filiale verschickt werden sollen. Die Zentrale gibt der Filiale mittels des Eingabeparameters „`requiredProductsAndAmount`“ an, welche Produkte in welcher Menge verschickt werden sollen. Die Filiale verringert also den Lagerbestand für die Produkte und reserviert so die Produkte für den Transport. Dazu iteriert die Komponentenoperation, wie in Abbildung 2.10 zu sehen, über alle zu versendenden Produkte. Der in der Schleife spezifizierte Ablauf wird also „`requiredProductsAndAmount.NUMBER_OF_ELEMENTS`“-Mal wiederholt. In der dargestellten Schleife wird mithilfe eines Aufrufs der `StoreQuery`-Komponente eine Objektrepräsentation des zugehörigen Produktdatensatzes aus der Datenbank initialisiert. Dann wird der Lagerbestand in dieser Objektrepräsentation und dadurch auch in der Datenbank verringert. In dieser Beispiel-SEFF sind Parameter für aufgerufene Komponenten und Ressourcenverbrauch zu sehen. Das `ExternalCallAction`-Element steht für den Aufruf anderer Komponenten. Dieser externe Aufruf richtet sich an eine Operation aus einer Schnittstelle, die im Komponenten-Repository als benötigte Schnittstelle der Komponente spezifiziert wurde. In der Beispiel-SEFF ist ein Aufruf der `queryProductById`-Operation der `StoreQuery`-Komponente vorgesehen. Die Antwortzeit und der Ressourcenverbrauch des spezifizierten Aufrufs sind der Spezifikation der aufgerufenen Komponentenoperation zu entnehmen. Im Gegensatz dazu steht das `InternalAction`-Element für den Ressourcenverbrauch in der Komponente durch eine interne Berechnung. Die Komponentenoperation verbraucht 5 CPU-Einheiten je Schleifendurchlauf. Die tatsächliche Zeit, die dazu benötigt wird um 5 CPU-Einheiten abzuarbeiten, hängt vom Ressourcenmodell (vgl. Unterabschnitt 2.3.4) ab.

2.3.3 Systemmodell

Die Komponenten aus dem Komponenten-Repository werden im Systemmodell zu einem System gekoppelt. Dabei sieht das PCM vor, dass Komponentenobjekte spezifiziert werden. Wenn also eine Komponente mehrfach im System vorkommen soll, so muss im Systemmodel jedes Vorkommen einzeln geplant

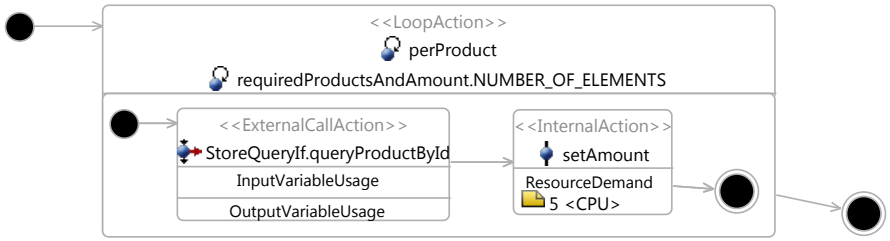

Abbildung 2.10: Herzstück des Performance-Kontrakts zu Nachricht 6 aus dem Anwendungsbeispiel (vgl. Abbildung 1.3) bestehend aus der Service-Effect-Specification (SEFF), die den Ressourcenverbrauch und den Aufruf weiterer Komponenten und Services modelliert.

werden. Im Systemmodell in Abbildung 2.11 kommen je zwei `Store`- und `StoreQuery`-Komponenten vor. Es soll eine Supermarktkette mit zwei Filialen modelliert werden. In jeder Filiale wird ein Hardwareknoten je eine Instanz der `Store`- und `StoreQuery`-Komponente ausführen. Im Systemmodell lässt sich zudem noch sehen, welche Schnittstellen wie gekoppelt werden. Dabei können auch Schnittstellen als Systemschnittstellen zur Verfügung gestellt werden. In Abbildung 2.11 erlaubt die Systemschnittstelle `StoreInterface` beispielsweise, dass Verkäufe in der Filiale gebucht werden können. Im Rahmen dieser Arbeit gehen wir davon aus, dass die Schnittstellen wenigstens so gekoppelt sind, dass ein bestimmtes Verwendungsmodell (s. Unterabschnitt 2.3.5) fehlerfrei simuliert (s. Unterabschnitt 2.3.6) werden kann. In Abbildung 2.11 sind ebenfalls nur die für das Beispiel aus Abschnitt 1.2 benötigten Kopplungen enthalten. Zudem müssen die für die Abarbeitung des Verwendungsmodells benötigten Systemschnittstellen zur Verfügung stehen.

2.3.4 Ressourcen- und Verteilungsmodell

Das Ressourcen- und das Verteilungsmodell werden jeweils vom System-Deployer erstellt. Das Ressourcenmodell gibt an, welche Hardwareknoten es gibt und wie sie verbunden sind. Abbildung 2.12 zeigt beispielsweise ein Ressourcenmodell, das drei Hardwareknoten spezifiziert, die über ein Netzwerk verbunden sind. Dabei enthält jeder Hardwareknoten eine Prozessorressource, die aus mehreren Kernen mit einer Verarbeitungsrate von 1000 CPU-Einheiten je Zeiteinheit bestehen. Somit würde die Verarbeitung von 5 CPU-Einheiten auf einem der angegebenen Rechenkerne 5/1000 Zeiteinheiten dauern (vgl.

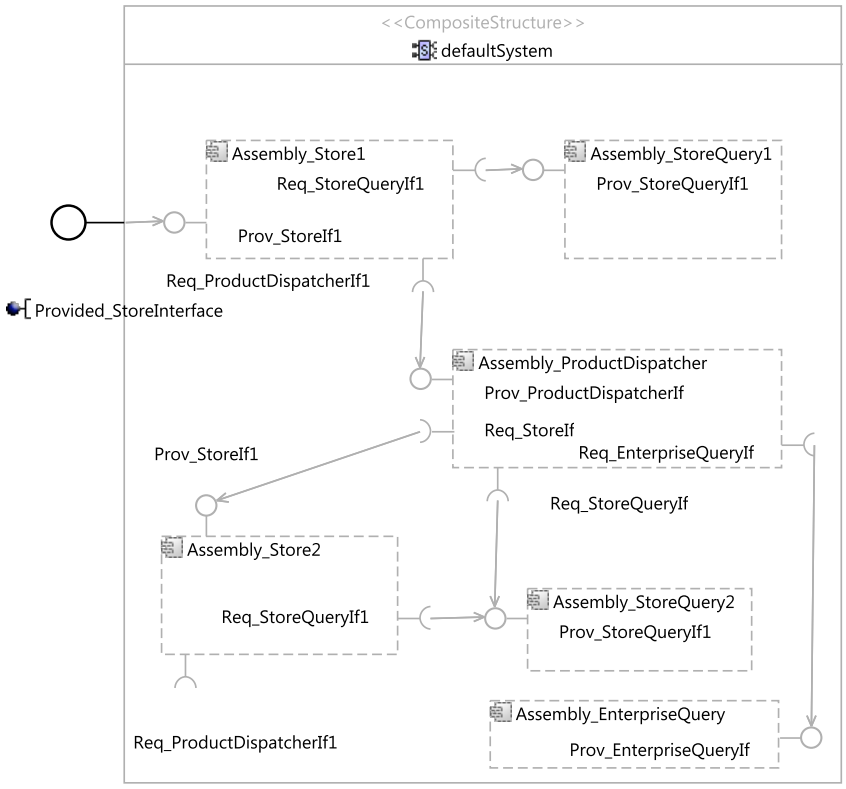

Abbildung 2.11: Beispiel für ein Systemmodell

Unterabschnitt 2.3.2). Für das Netzwerk sind die Latenz von 50/1000 Zeiteinheiten sowie der Datendurchsatz je Zeiteinheit angegeben. Im PCM können keine Einheiten angegeben werden, so dass beispielsweise die Bedeutung einer abstrakten Zeiteinheit per Konvention für das Projekt festgelegt werden muss. Im Beispiel könnte eine Zeiteinheit einer Sekunde entsprechen, so dass 5 CPU-Einheiten in 5 ms von einem Rechenkern bearbeitet werden und die Latenz des Netzwerks 50 ms beträgt.

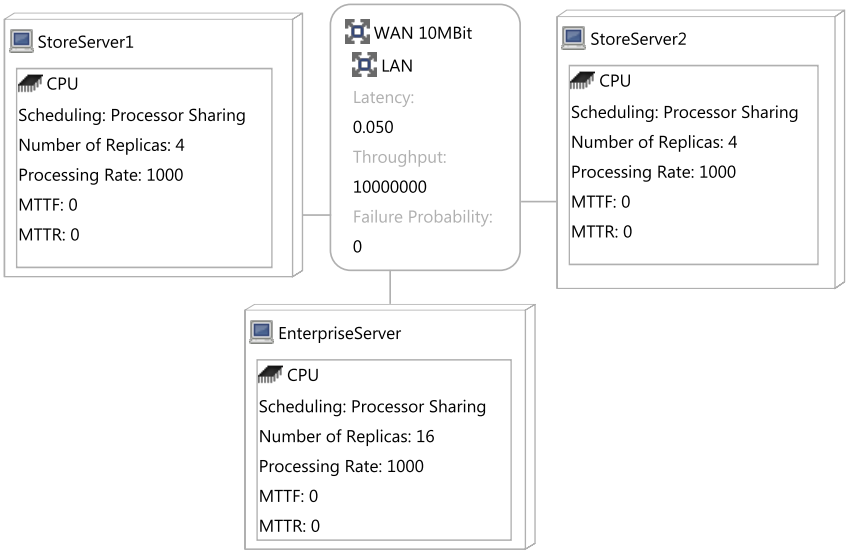

Abbildung 2.12: Beispiel für ein Ressourcenmodell

Das Verteilungsmodell baut auf dem Ressourcenmodell auf und koppelt die Komponentenobjekte aus dem Systemmodell mit den Hardwareknoten aus dem Ressourcenmodell. Ein Beispielverteilungsmodell ist in Abbildung 2.13 zu sehen. Dabei stellen die äußeren Rechtecke Hardwareknoten dar, die die jeweils dort ausgeführten Komponentenobjekte umschließen. Die dargestellten Hardwareknoten und Komponentenobjekte referenzieren das Ressourcenmodell (vgl. Abbildung 2.12) bzw. das Systemmodell (vgl. Abbildung 2.11).

2.3.5 Verwendungsmodell

Der Domänenexperte setzt sein Wissen ein, um zu modellieren, wie der Benutzer voraussichtlich mit dem System interagiert. Dazu legt er verschiedene Verwendungsmodelle an. Jedes Verwendungsmodell kann eine oder mehrere

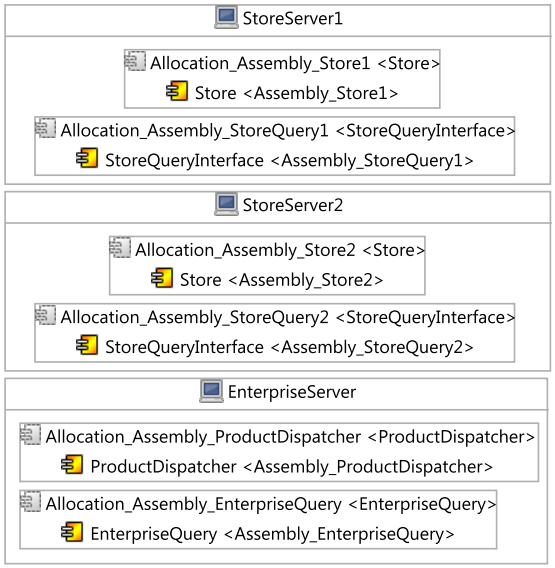

Abbildung 2.13: Beispiel für ein Verteilungsmodell

Szenarien beschreiben. Jedes Szenario gibt an, wie ein Benutzer das System in verschiedenen Schritten verwendet. Ähnlich wie beim SEFF können in einem Szenario bedingte Abläufe und Schleifen angeben werden. Der Benutzer kann Anfragen an das System richten oder aber zwischen Anfragen abwarten, um beispielsweise eine Auswahl zu treffen. Das Szenario gibt zwar die Handlung eines einzelnen Benutzers an, aber in der Simulation wird das Szenario von einer Vielzahl von Benutzern ausgeführt, um eine gewisse Systemlast zu erreichen. Dazu kann je Szenario eine offene oder geschlossene Benutzergruppe spezifiziert werden. Bei einer offenen Benutzergruppe tritt nach einem festgelegten Intervall ein neuer Benutzer auf, der das Szenario ausführt. Die Anzahl der Benutzer, die gleichzeitig mit dem System interagieren, ist bei offenen Benutzergruppen also theoretisch unbegrenzt. Demgegenüber ist die Benutzeranzahl in geschlossenen Benutzergruppen fest. Allerdings wird hier jeder Benutzer nach einem festen Pausenintervall das Szenario von vorn beginnen. Ein simples Beispiel für ein Verwendungsmodell ist in Abbildung 2.14 zu sehen. Hier wird eine Auslastung mit einer geschlossenen Benutzergruppe modelliert, bei der acht Benutzer je nach 5 Sekunden (bzw. abstrakten Zeiteinheiten) die Ausführung des Szenarios wiederholen. Das Szenario selbst besteht aus einem einzelnen Systemaufruf. Dabei wird die Methode `bookSale` aus der Systemschnittstelle (vgl. Systemmodell in Abbildung 2.11) mit dem Parameter `sales` aufgerufen, wodurch der Einkauf des jeweiligen Benutzers in der Filiale registriert wird. Das Verwendungsmodell ermöglicht es, wichtige

Charakteristika des Eingabeparameters oder sogar seinen Wert festzulegen. Im Beispiel wurde angegeben, dass zu einem Einkauf eine unterschiedliche Menge von Produkten gehören kann. Welche Operationsparameter zur Verfügung stehen, ist in der Schnittstelle im Komponenten-Repository festgelegt (vgl. Operationsschnittstelle im Komponenten-Repository in Abbildung 2.9).

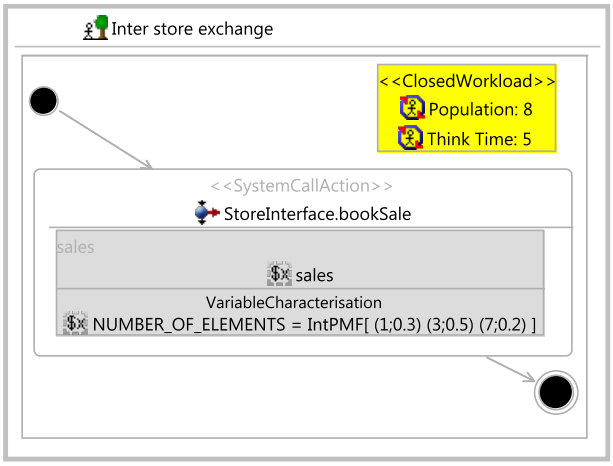

Abbildung 2.14: Beispiel für ein Verwendungsmodell

2.3.6 Simulation

Die vorgenannten Modelle werden vom QoS-Analysten zu einer vollständigen PCM-Instanz integriert. Diese PCM-Instanz lässt sich mithilfe der Palladio-Workbench[3] simulieren. Zunächst gibt der QoS-Analyst das Verwendungs- und das Verteilungsmodell an, das simuliert werden soll. Über das Verteilungsmodell sind auch gleichzeitig die zugehörigen Modelle, also Ressourcen- und Systemmodell sowie das Komponenten-Repository mit den SEFFs über die jeweiligen modellinternen Referenzen klar definiert. Da das Verwendungsmodell einen unendlichen Strom an Benutzern modelliert, muss ein Stoppkriterium als zusätzliche Eingabe angegeben werden. Standardmäßig gibt der QoS-Analyst hier die maximale Simulationszeit bzw. die maximale Anzahl von Beobachtungen vor. Zuletzt muss auch der Speicherort für die Beobachtungen angegeben werden. Die Palladio-Workbench hat im Laufe

[3] Im Laufe dieser Arbeit wurde die Palladio-Workbench 3.3 verwendet, die unter der folgenden Internetadresse erhältlich ist: http://sdqweb.ipd.kit.edu/wiki/PCM_3.3 (letzter Zugriff am 05.04.2014)

der Zeit viele weitere Funktionen erhalten, wie zum Beispiel die Berücksichtigung von Middleware-Komponenten [32], die auch im Zusammenhang mit der Simulation konfiguriert werden können. Diese zusätzlichen Funktionen werden im Laufe der Arbeit nicht weiter berücksichtigt.

Die Simulation wird durchgeführt, indem die angegebenen Modelle in ein Java-Programm transformiert werden. Das Java-Programm führt dann die im Verwendungsmodell angegebenen Abläufe als Jobs nebenläufig aus. Die Abarbeitung der getätigten Operationsaufrufe führt dann zu Ressourcenverbräuchen, die mithilfe einer Scheduling-Simulation mit den zur Verfügung stehenden Ressourcen verrechnet werden. Daraus werden automatisch Werte für Ressourcenverbrauch und -auslastung sowie den Durchsatz und die Antwortzeiten der Operationen ermittelt und gespeichert. Zusätzlich zu den beschriebenen Performance-Aspekten kann auch die Zuverlässigkeit eines Systems einbezogen werden, was aber in dieser Arbeit nicht weiter relevant ist. Sobald eines der angegebenen Stoppkriterien erreicht ist, endet die Simulation.

Das Ergebnis der Simulation sind verschiedene Performance-Datenreihen, die zum einen die Antwortzeit der Komponentenoperationen getrennt nach Aufrufmodellierung im SEFF und zum anderen die Ressourcenauslastung betreffen. Im Folgenden wird nun die Verwendung der Antwortzeitergebnisse näher beschrieben. Die Antwortzeitdatenreihen können dabei in Form verschiedener Diagramme angezeigt werden. Dies können beispielsweise ein X/Y-Diagramm mit Messungsnummer und Antwortzeit, ein Histogramm, eine kumulierte Verteilungsfunktion oder Informationen über Konfidenzintervalle sein. Die Palladio-Workbench ermöglicht es auch die Datenreihe und jedes der Diagramme bzw. die ihm zugrunde liegenden Daten zu exportieren, um eigene Analysen zu ermöglichen. Die Palladio-Workbench ist nicht dazu geeignet Testkriterien automatisch auszuwerten, sondern sie bietet dem QoS-Analysten die beschriebenen Daten und Visualisierungen, um selbst Schlüsse daraus zu ziehen.

2.3.7 Vergleich des PCMs mit den Grundbegriffen

Das Palladio Component Model lässt sich relativ leicht auf die Grundbegriffe abbilden. Das gilt insbesondere für den Spezifikationsteil. An vielen Stellen geht das PCM auch über die Grundbegriffe hinaus und beschreibt Sachverhalte in seinen Modellen genauer oder zusätzlich. Dies liegt daran, dass das PCM konkret zur Erstellung formaler Performance-Modelle von komponentenbasierten Systemen genutzt wird, die anschließend simuliert werden. Dadurch

ist der Anspruch an das PCM um einiges höher als an die Grundbegriffe, die lediglich dazu dienen, die wichtigsten Begriffe für diese Arbeit zu erläutern und in Beziehung zu setzen. Im Folgenden wird zunächst untersucht, ob und wie die Bereiche Komponente, Komposition und Komponentenverteilung aus dem PCM auf die Grundbegriffe abgebildet werden können. Danach gehen wir analog für den Bereich des Verwendungsmodells vor. Zuletzt analysieren wir auch den Bereich der Performance-Messungen.

Die Abbildung 2.15 vergleicht die Bereiche Komponente, Komposition und Komponentenverteilung aus dem PCM (linke Seite) mit dem Modell aus den Grundbegriffen (rechte Seite). Der Bereich der Komponente lässt sich sehr leicht auf das PCM abbilden. Hier ist in den Grundbegriffen lediglich der funktionale Kontrakt als Container für die Schnittstellen zusätzlich vorhanden. Bei beiden Modellen ist der funktionale Kontrakt existenzieller Bestandteil einer Komponente, aber die Grundbegriffe weisen darauf hin, dass Kontrakte für verschiedene Qualitätsmerkmale zu einer Komponente gehören. Der Performance-Kontrakt wird über eine SEFF im PCM abgebildet. Die SEFF ist dabei im PCM mit allen Details enthalten, wogegen dieser Begriff in den Grundbegriffen nur abstrakt als Zuordnung einer Operation zu einer Performance-Metrik modelliert wird. Während für die Kontrakte Unterschiede bestehen, gibt es eine direkte Entsprechung für die Komponenten, Schnittstellen und Operationen in beiden Modellen.

Ein wesentlicher Unterschied des PCMs zu den Grundbegriffen ist, dass ein System im PCM auch eine Komponente mit entsprechenden Schnittstellen ist. Beim PCM können auch Subsysteme und Komponenten aus verschiedenen Komponenten zusammengesetzt sein. Generell werden diese Elemente als komponierte Komponenten bezeichnet. Der Einfachheit halber haben die Grundbegriffe keine komponierten Komponenten definiert. Hier gibt es lediglich das Kompositionselement, das explizit eine Zusammenstellung von Komponenten kapselt. Diese Einschränkung kann dadurch umgangen werden, dass das Modell eines Systems mit komponierten Komponenten in eines mit ausschließlich einfachen Komponenten überführt wird. Die Transformation lässt sich beispielsweise realisieren, indem eine komponierte Komponente in eine Verbinderkomponente und die enthaltenen Komponenten zerlegt wird. Da in den Grundbegriffen komponierte Komponenten nicht berücksichtigt werden, gibt dort auch keine äquivalente Entsprechung für die Schnittstellendelegierung aus dem PCM. Die Schnittstellendelegierung wird benötigt, wenn eine innere Komponente direkt an die Schnittstelle der komponierten Komponente gekoppelt wird. Dies gibt es in den Grundbegriffen nur in Form der Systemschnittstelle, die eine Komponentenschnittstelle als Systemschnittstelle kennzeichnet. Neben diesen Unterschieden lässt sich für die

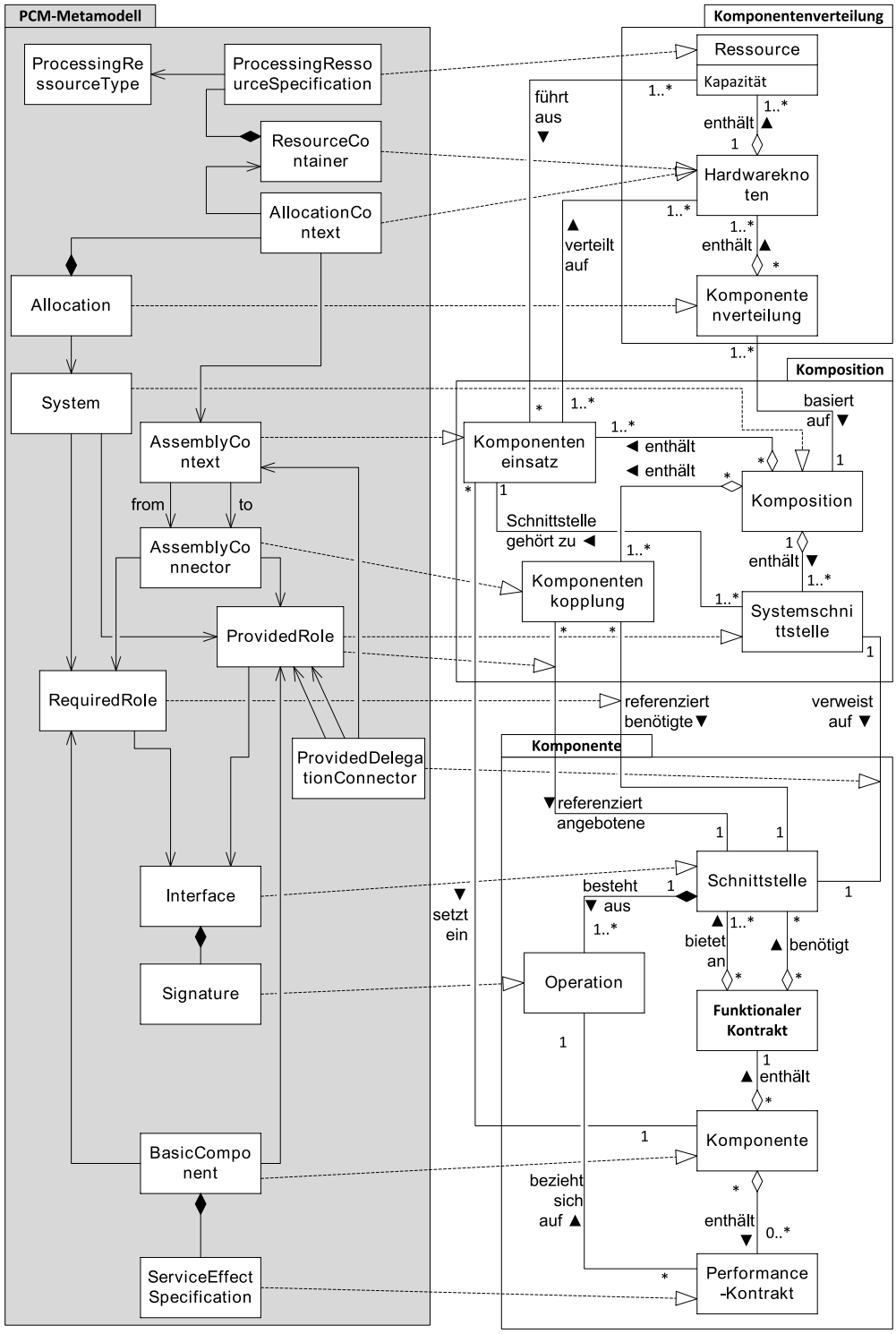

Abbildung 2.15: Vergleich der Bereiche Komponenten, Komposition und Komponentenverteilung mit dem PCM

Komponentenkopplung und den Komponenteneinsatz aus den Grundbegriffen wieder eine direkte Entsprechung im PCM finden.

Bei der Komponentenverteilung ist die Zuordnung zur „Allocation“ auf der PCM-Seite eindeutig. Aufseiten der Grundbegriffe werden die Komponentenobjekte direkt der ausführenden Hardware-Ressource zugeordnet. Beim PCM liegt zwischen der Ressource und der Zuordnung noch eine Zwischenschicht. Der „AllocationContext“ ordnet das Komponentenobjekt („AssemblyContext“) hier einem „RessourceContainer“ zu, der wiederum mehrere Ressourcen enthalten kann. Das Komponentenobjekt kann nun alle Ressourcen aus dem „RessourceContainer“ nutzen. Somit korrespondiert der Hardwareknoten bei den Grundbegriffen sowohl zum „AllocationContext“ als auch „RessourceContainer“ aus dem PCM. Im PCM können so Modelle von Hardware-Konfigurationen leichter wiederverwendet werden. Das PCM kennt nicht nur Ressourcen sondern auch verschiedene Ressourcentypen mit verschiedenen Eigenschaften. Darunter sind auch Netzwerkressourcen, die verschiedene Hosts untereinander verbinden. Dies wird von dem Modell, das in den Grundbegriffen dargestellt ist, nicht unterstützt.

In Abbildung 2.16 wird der Aspekt der Systemverwendung aufgegriffen. Dabei werden wieder die Elemente aus dem PCM auf der linken Seite den Grundbegriffen auf der rechten Seite zugeordnet. Bei den Grundbegriffen wird auch auf Performance-Testfälle eingegangen, die Schritte beschreiben, mit denen das System verwendet werden soll. Dies ist im PCM nicht enthalten, allerdings gibt es im PCM-Verwendungsmodell einige Parallelen. So kann das Element „UsageModel“ dem Performance-Testfall zugeordnet werden. Die Spezifikation von Testkriterien und deren Auswertung werden vom PCM nicht unterstützt. Die Anfragesequenz in den Grundbegriffen dient mehreren Zwecken. Zunächst spezifiziert sie das Nebenläufigkeitsverhalten der Sequenz. Dies geschieht im PCM über die sogenannte „Workload“. Zudem dient die Anfragesequenz als Container für die in der Sequenz geplanten Anfragen. Das übernimmt im PCM das Element „UsageScenario“ bzw. der Spezial-Container „ScenarioBehaviour“. Eine direkte Übereinstimmung gibt es wieder bei der Anfrage selbst. Dabei geht die Funktionalität des PCMs bei der Sequenz und den Anfragen deutlich über die Grundbegriffe hinaus. Beim PCM lässt sich ein Kontrollfluss abbilden, bei dem bedingte Anweisungen und Schleifen sowohl Anfragen als auch Wartezeiten steuern. Bei Kontrollflusselementen, wie Schleifen, verwendet das PCM „ScenarioBehaviour“-Element als Container für die dort gekapselten Kontrollflüsse, wie z. B. den Schleifenrumpf. Dies erklärt die oben erwähnte Trennung von „UsageScenario“ und „ScenarioBehaviour“. Darüber hinaus können im Rahmen der Anfragen beim PCM auch Übergabeparameter spezifiziert werden.

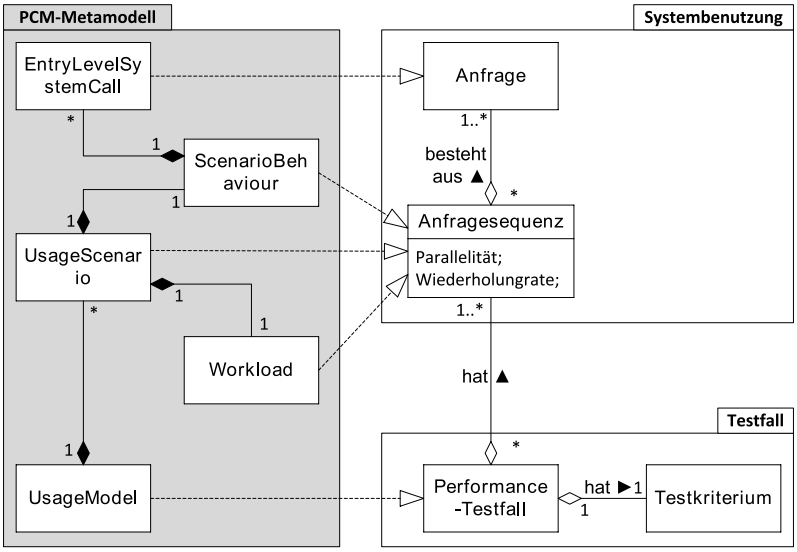

Abbildung 2.16: Vergleich der Bereiche Systembenutzung und Testfall mit dem PCM

In der Abbildung 2.17 wird der Bereich der Messung aus den Grundbegriffen (vgl. Abschnitt 2.1) auf der rechten Seite der Realisierung im PCM auf der linken Seite gegenübergestellt. Das PCM-Metamodell deckt den Bereich leider nicht ab, so dass sich die folgende Diskussion auf die Implementierung in der Palladio-Workbench (Version 3.3) im Paket `SensorFramework` stützt. In der PCM-Implementierung werden sogenannte Experimente durchgeführt. Ein Experiment spiegelt ein bestimmtes Verwendungsmodell wieder, dass ein bestimmtes System verarbeiten muss. Dieses Experiment kann dann häufiger ausgeführt werden. Eine Experimentausführung bedeutet, dass eine bestimmte Simulation (vgl. Unterabschnitt 2.3.6) ausgeführt wurde. Hier sind die Parallelen zum Testfall und der Testfallausführung offensichtlich. Der Umgang mit den Messwerten unterscheidet sich ab diesem Punkt beträchtlich zwischen den Grundbegriffen und der PCM-Implementierung.

Die Palladio-Workbench kennt sogenannte Sensoren, die beispielsweise Zeitspannen, wie zum Beispiel Antwortzeiten messen. Ein Sensor hat einen Namen, der auf den Operationsaufruf hinweist, dessen Antwortzeit hier wiedergegeben wird. Im Sensornamen wird der `AssemblyContext` (also die Komponentenverwendung) aus dem Systemmodell und das Aufrufobjekt aus dem SEFF jeweils über ihre ID referenziert. Eine explizite Referenzierung des PCM-Modells erfolgt nicht. Jeder Antwortzeitsensor (in Abbildung 2.17 `TimeSpanSensor`) wird von verschiedenen Messwertsammlungen referenziert,

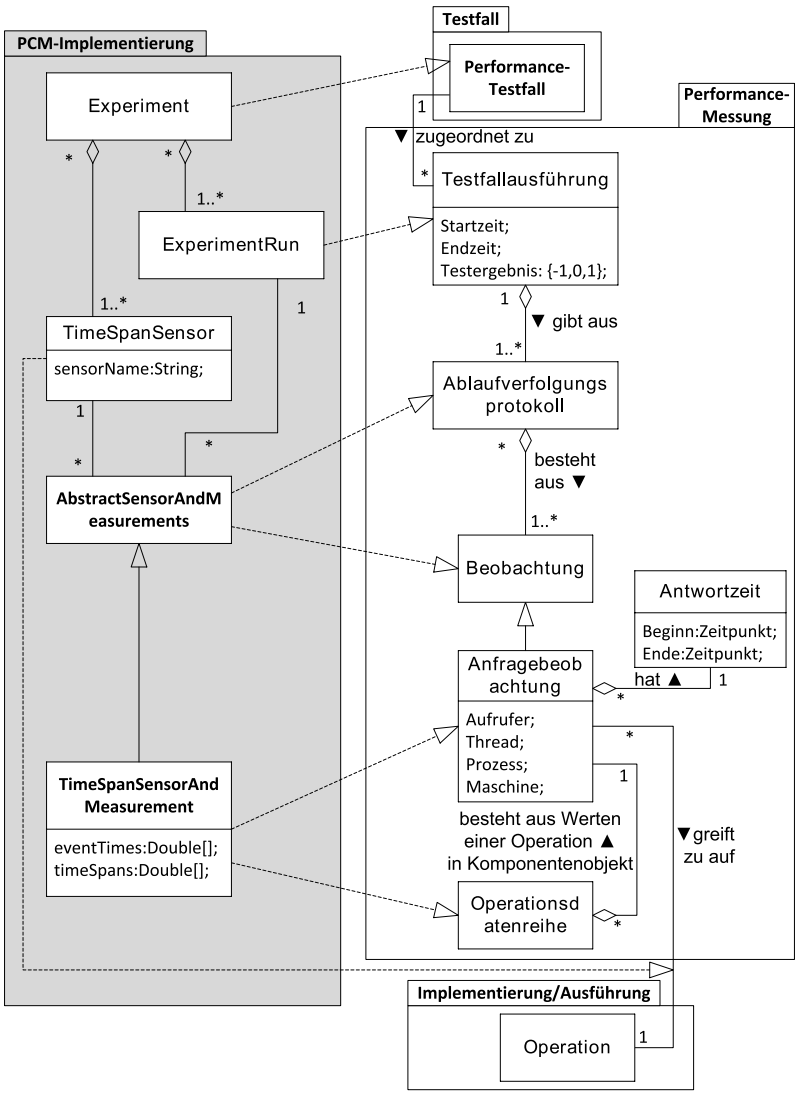

Abbildung 2.17: Vergleich des Bereichs Performance-Messung mit dem PCM

die je zu unterschiedlichen Experimentausführungen (in Abbildung 2.17 `ExperimentRun`) gehören. Man könnte nun sagen, dass die Messwertsammlungen zum Ablaufverfolgungsprotokoll korrespondieren, aber es gibt auch Unterschiede. Das Ablaufverfolgungsprotokoll umfasst alle Messwerte einer Testfallausführung und es gibt nur mehrere Ablaufverfolgungsprotokolle zu einer Testfallausführung, wenn dies aus technischen Gründen nötig ist. Dies ist z. B. der Fall, wenn mehrere Protokolle auf verschiedenen Hardwareknoten anfallen. Eine Messwertsammlung im PCM ist aber immer „sortenrein“, d.h., es sind beispielsweise nur Antwortzeiten eines Operationsaufrufs enthalten. Somit passt die Messwertsammlung eher zur Operationsdatenreihe. Daher wurde lediglich die abstrakte Oberklasse `AbstractSensorAndMeasurements` dem Ablaufverfolgungsprotokoll zugeordnet. Da diese Klasse aber zugleich ein Messwertcontainer ist, entspricht sie auch der Beobachtung. Dieser Unterschied liegt darin begründet, dass die Modellierung einzelner Messwerte für die Erklärung hilfreich ist, aber für die Implementierung eine zu hohe Laufzeit und Speicherauslastung bedeuten kann. Die abgeleitete Klasse `TimeSpanSensorAndMeasurement` lässt sich, wie ihre Oberklasse, zwei verschiedenen Elementen in den Grundbegriffen zuordnen. Zum einen nimmt die Klasse viele Messwerte sortenrein auf, daher entspricht sie der Operationsdatenreihe. Zum anderen wird die Klasse der Anfragebeobachtung zugeordnet, da sie die konkreten Messwerte enthält. Allerdings wird hier lediglich die Startzeit und die Dauer des Aufrufs[4] aufgezeichnet. Der Kontext wird beim PCM nicht explizit erfasst, sondern er wird, wie schon erwähnt, nur über den Sensornamen und die darin enthaltene implizite Referenz auf das PCM-Modell modelliert. Damit kann das PCM eine Messung des Kontexts nicht komplett ersetzen, da durch diese Modellierung lediglich die generelle Aufrufbeziehung von einer Komponente (nicht dem Komponentenobjekt) an eine bestimmte Schnittstelle festgelegt ist.

2.4 Performance-Test

Der Test ist eine Maßnahme zur Qualitätssicherung, bei der die Software ganz oder in Teilen ausgeführt wird, um ihre Qualitätseigenschaften zu überprüfen. Bei der Qualitätssicherung wird häufig zwischen statischen und dynamischen Maßnahmen [69] unterschieden. Bei statischen Maßnahmen werden Entwurfsdokumente oder der Quelltext analysiert, um so Rückschlüsse auf

[4] Es gibt zwei Möglichkeiten eine Antwortzeit zu erfassen. Auf der einen Seite können Start- und Endzeitpunkt und auf der anderen Seite Startzeitpunkt und Dauer gemessen werden. Der jeweils fehlende Wert kann in beiden Fällen aus den beiden gegebenen Werten berechnet werden.

die Qualität der Software zu ziehen. Bei dynamischen Maßnahmen wird die Software ausgeführt, es wird also getestet.

Im Rahmen des Tests werden die gesamte Software oder Teile davon ausgeführt. Dabei ist es vom Entwicklungsstand der Software abhängig, welcher Teil der Software getestet werden kann. In Unterabschnitt 2.4.1 werden daher die sogenannten Teststufen, in denen die verschiedenen Teile getestet werden, erläutert. Beim Test werden einzelne reproduzierbare Szenarien, sogenannte Testfälle, ausgeführt. Jeder Testfall prüft einen Aspekt der Software in einem spezifischen Szenario. Welche Informationen ein Testfall umfasst, wird zusammen mit der Testfallnotation gemäß dem Standard IEEE 829 [34] in Unterabschnitt 2.4.2 erklärt. Bei einem Test werden neben der Funktionalität einer Software auch ihre nichtfunktionalen Eigenschaften untersucht. Da sich diese Arbeit auf die Untersuchung der Performance-Eigenschaften eines Systems fokussiert, wird in Unterabschnitt 2.4.3 auf die Besonderheiten des Performance-Tests eingegangen. Schließlich werden in Unterabschnitt 2.4.4 noch kurz die Besonderheiten für den Test komponentenbasierter Systeme zusammengefasst.

2.4.1 Teststufen

Man unterscheidet beim Test sogenannte Teststufen, bei denen unterschiedliche Teile und Aspekte einer Software getestet werden. Die Unterscheidung nach Teststufen wurde zuerst für das V-Modell nach Boehm [10] festgelegt, dass den jeweiligen Entwicklungsstufen immer auch eine Teststufe gegenüberstellt. Dabei gibt es die Teststufen Komponententest, Integrationstest, Systemtest und Akzeptanztest.

Der Komponententest prüft, ob jede Komponente für sich genommen ihre Spezifikation erfüllt. Hier steht der Begriff Komponente [69] lediglich für die kleinste spezifizierte Softwareeinheit und hat nichts mit der in Unterabschnitt 2.2.1 diskutierten Definition zu tun. Daher wird der Komponententest auch oft als Unit-Test bezeichnet. Da in dieser Arbeit die Komponente nach der Definition in Unterabschnitt 2.2.1 als kleinste Softwareeinheit behandelt wird, ist die Analogie im Rahmen dieser Arbeit trotz der unterschiedlichen Terminologie zutreffend. Der Komponententest wird im Rahmen der komponentenbasierten Entwicklung (vgl. Unterabschnitt 2.2.2) zuerst von den Komponentenentwicklern durchgeführt. Dabei prüfen sie die Komponente isoliert von ihrem späteren Einsatz mithilfe eines speziellen Testtreibers, der die Tests anstößt, und mithilfe von Platzhaltern, die benötigte Komponenten simulieren. Der Komponententest wird häufig von Entwicklern durchgeführt,

da sie am nächsten an der Implementierungsebene sind. So können Sie ihr Wissen direkt in den Test einbringen, um beispielsweise Platzhalter zu realisieren. Außerdem kann auch der Systemarchitekt einen Komponententest im Rahmen der Beschaffung der Komponenten durchführen, um zu überprüfen, ob die beschafften Komponenten auch die zugesicherten Eigenschaften haben.

Der Integrationstest prüft, ob ein Verbund von Komponenten wie gewünscht zusammenarbeitet [69]. Dabei werden die relevanten funktionalen und nichtfunktionalen Eigenschaften des Komponentenverbunds untersucht. Zusätzlich wird aber auch der Datenfluss zwischen den Komponenten geprüft. Der Integrationstest wird bei komponentenbasierter Entwicklung aufseiten der Systementwickler während der Zusammenstellung (vgl. Unterabschnitt 2.2.2) des Systems durchgeführt. Je nach gewählter Integrationsstrategie müssen die Systementwickler spezielle Testtreiber oder Platzhalter erstellen. Spezielle Testtreiber werden benötigt, wenn eine Schnittstelle des Komponentenverbunds getestet werden soll, die keine Systemschnittstelle ist. Spezielle Platzhalter werden für die nicht gekoppelten benötigten Schnittstellen des Komponentenverbunds benötigt.

Der Systemtest untersucht das System als Ganzes [69]. Es wird überprüft, ob das System den funktionalen und nichtfunktionalen Anforderungen genügt. Dabei führen die Systementwickler das System in einer Testumgebung aus, die möglichst realistisch sein soll. Der Systemtest ist die Teststufe, die nach dem Zusammenstellen des Systems ausgeführt wird. Bei einem iterativ inkrementellen Entwicklungsprozess, wie dem RUP [46], kann der Systemtest auch auf jeden Meilenstein angewendet werden.

Schließlich prüft der Abnahmetest das Gesamtsystem in einer Installation beim Kunden. Die Produktivumgebung steht dabei häufig nicht zur Verfügung, da der Kunde die Kontinuität seines Betriebs gewährleisten muss. Beim Abnahmetest wird die Einhaltung der vertraglich geregelten Anforderungen und gesetzlichen Normen durch den Kunden überprüft. Dabei werden Testfälle vom Kunden erstellt und ausgeführt. Da Kunden nicht immer in der Lage sind den Test selbst durchzuführen, kann die Ausführung der Tests auch von den Systementwicklern übernommen werden.

2.4.2 Testfälle und Testfallnotation

Jeder Testfall legt ein spezifisches Prüfszenario für das zu prüfende Testobjekt fest. Das Testobjekt kann je nach Teststufe (vgl. Unterabschnitt 2.4.1) eine Komponente, ein Komponentenverbund oder das Gesamtsystem sein. Ein

Testfall ist ein Dokument, das festhält unter welchen Bedingungen, welche Eingaben an das System gerichtet werden und welche Ergebnisse zu erwarten sind. Ein Testfall wird aus der Spezifikation des Testobjekts abgeleitet. Dabei werden zunächst die Idee und der Ablauf des Testfalls entworfen. Im nächsten Schritt wird er durch Hinzufügen der Testdaten konkretisiert. Dann wird der Testfall realisiert, indem eventuell nötige Testtreiber und Platzhalter erstellt werden. Schließlich wird der Testfall ausgeführt und die Ausführung wird entweder gelingen oder fehlschlagen.

Nach dieser kurzen Einführung in den Zweck, die Inhalte und den Lebenszyklus eines Testfalls soll nun der Inhalt näher untersucht werden. Spillner und Linz [69] charakterisieren den Inhalt der Testfälle wie folgt:

> „Umfasst folgende Angaben: die für die Ausführung notwendigen Vorbedingungen, die Menge der Eingabewerte (ein Eingabewert je Parameter des Testobjekts) und die Menge der erwarteten Sollwerte, die Prüfanweisung (wie Eingaben an das Testobjekt übergeben und Sollwerte abzulesen sind) sowie die erwarteten Nachbedingungen."

Diese Inhaltscharakterisierung fasst die notwendigen Inhalte eines Testfalls zusammen. Die konkrete Ausgestaltung des Testfalls kann aber für die Organisation oder sogar für das Projekt individuell festgelegt werden. Dabei können die von Spillner und Linz erwähnten Bestandteile eines Testfalls erweitert werden. Im Folgenden wird die Testfallnotation gemäß des Standards IEEE 829 [34] betrachtet, die eine konkrete Ausgestaltung von Testfällen vorgibt. Beim Standard IEEE 829 wird die gesamte Testdokumentation standardisiert. Dies umfasst neben der Testfallnotation beispielsweise auch die Teststrategie. Testfälle werden laut IEEE 829 maßgeblich in drei Dokumenten spezifiziert: dem Testfallentwurf (im Standard „level test design"), der Testfallspezifikation (im Standard „level test case") und der Testfallablaufspezifikation (im Standard „level test procedure").

Im Testfallentwurf (im Standard „level test design") werden die Qualitätsmerkmale des Systems festgelegt, die getestet werden sollen. Darüber hinaus wird auch das Verfahren, wie getestet werden soll, generell festgelegt. Dabei werden die einzelnen Testfälle kurz genannt und beschrieben. Im Testfallentwurf werden also mehrere Testfallspezifikationen bzw. Testfallablaufspezifikationen referenziert. Der Testfallentwurf legt fest, nach welchem Kriterium beurteilt wird, ob das zu testende Qualitätsmerkmal des Systems insgesamt erfolgreich getestet wurde.

Tabelle 2.1 stellt die Struktur der Testfallspezifikation (im Standard „level test case" genannt) nach IEEE 829 dar. Als Erstes umfasst ein Testfall eine

eindeutige Identifikationsnummer, die schon im Testfallentwurf festgelegt wurde und mit deren Hilfe Querverbindungen zu anderen Testfällen angegeben werden können. Dann wird das Ziel des Testfalls genau spezifiziert. Dies kann auch die Priorität des Testfalls umfassen. Im dritten Abschnitt werden die Eingaben für den Testfall detailliert festgelegt. Dabei werden unter Umständen die Werte direkt festgelegt oder aber die verwendeten Dateien, Datenbanken, Generatoren, Speicherbereiche, Terminal-Nachrichten, etc. eindeutig identifiziert. Es werden auch die Beziehungen zwischen den Eingaben angegeben. Das umfasst beispielsweise den zeitlichen Ablauf der Eingaben. Als Nächstes werden die erwarteten Ausgaben und das erwartete Verhalten dieses Testfalls beschrieben. Dabei werden die exakten Werte (ggf. mit Toleranz) angegeben, die die erwarteten Ausgaben bzw. das erwartete Verhalten charakterisieren. Im Abschnitt „Environmental needs“ wird die Testumgebung für diesen Testfall bzw. eine Gruppe von Testfällen spezifiziert. Dabei wird genau festgelegt, welche Software und welche Hardware beteiligt sind und wie die Software auf der Hardware verteilt ist. Die Konfiguration der beteiligten Software und Hardware wird hier auch spezifiziert. Die beteiligte Software umfasst auch Testwerkzeuge, wie z. B. Lastgeneratoren und Platzhalter. Als Nächstes werden Anforderungen an die nachgelagerte Testfallablaufspezifikation, wie z. B. besondere Kenntnisse der Tester, formuliert. Schließlich wird dokumentiert, welche Testfälle vor dem hier spezifizierten Testfall ausgeführt werden müssen. Dabei wird auch jeweils die Art der Abhängigkeit kurz beschrieben.

Tabelle 2.1: Die Struktur einer Testfallspezifikation („level test case“ genannt) nach dem Standard IEEE 829 [34], bestehend aus sieben Teilen.

2	**Details (once per test case)**
2.1	Test case identifier
2.2	Objective
2.3	Inputs
2.4	Outcome(s)
2.5	Environmental needs
2.6	Special procedural requirements
2.7	Inter-case dependencies

Die Testfallablaufspezifikation (im Standard „level test procedure“) legt die Schritte fest, die im Rahmen der Testfallausführung nötig sind. Die Testfallablaufspezifikation steht immer im Bezug zu der anderen Testdokumentation, insbesondere der Testfallspezifikation. Die Referenzen zu den Bezugsdoku-

menten werden in der Testfallablaufspezifikation angegeben. Zunächst stellt die Testfallablaufspezifikation im Detail dar, welche Voraussetzungen es für die Ausführung des Testfalls gibt. Dies umfasst die Testfallspezifikation, Datenbanken, Werkzeuge, etc. Zusätzlich können auch besondere Anforderungen an die Ausführung angegeben werden, beispielsweise Anforderungen an die Tester. Schließlich legt die Testfallablaufspezifikation die auszuführenden Schritte im Testfall fest. Die Schritte umfassen dabei unter anderem Anweisungen für das Starten des Systems, das Aufwärmen der Caches, die Ausführung der verschiedenen Systemfunktionen, die Messung verschiedener Werte und schließlich das Herunterfahren des Testsystems. Dabei kann angegeben sein, welche und wie starke Variationen im Ablauf erlaubt sind.

2.4.3 Performance-Tests

Beim Performance-Test wird überprüft, ob die Leistung der Software den Anforderungen entspricht. Dabei werden Anforderungen im Bezug auf die Performance-Metriken Antwortzeit, Durchsatz und Ressourcenverbrauch [35] evaluiert. Diese Art von Test weist einige Besonderheiten auf. Aufgrund von verschiedenen Effekten, wie Caching, Scheduling und parallel laufenden anderen Rechenprozessen, sind einzelne Messungen der genannten Metriken üblicherweise Schwankungen ausgesetzt. Daher sollen die geforderten Werte im Rahmen von Performance-Testfällen immer mehrfach gemessen werden, um Ausreißer kompensieren zu können und in jedem Fall zu repräsentativen Werten zu kommen. Dies ist insbesondere bei komplexeren Testfällen wichtig, die mehrere parallele Anfragen an ein System beinhalten.

Weitere Eigenarten des Performance-Test stellt Molyneaux [51] in seinem Buch heraus. Zunächst weist er darauf hin, dass Performance-Testfälle meist nur mithilfe von Testautomatisierung durchgeführt werden können. Beim Performance-Test werden häufig viele Benutzer gleichzeitig an einem System simuliert. Da meist weder so viel Personal noch so viele (Hardware-)Clients zur Verfügung stehen, um die nötige Last zu erzeugen, ist Testautomatisierung unabdingbar. Dabei muss der Lastgenerator ebenso leistungsfähig sein, wie das getestete System. Er muss ggf. auf mehreren Hardwareknoten verteilt werden, um ausreichend Last zu generieren. Die zur Verfügung stehende Verbindung zwischen dem Lastgenerator und dem zu testenden System sollte die Anfragelast zudem nicht künstlich ausbremsen. Beim Performance-Test müssen also nicht nur die richtigen Testwerkzeuge eingesetzt werden, sondern auch die Testumgebung muss im Rahmen der Testfallerstellung sorgfältig geplant werden. In diesem Zusammenhang weist Molyneaux auch darauf hin,

dass für die Testfälle eine ausreichende Menge an realistischen Testdaten zur Verfügung stehen sollte.

Ein zweiter Aspekt, den Molyneaux [51] beschreibt, ist, dass Performance-Tests erst dann zum Einsatz kommen können, wenn das System eine gewisse Stabilität erreicht hat. Das zu testende System sollte also zuvor auf funktionale Fehler getestet werden und erst wenn es einen gewissen Reifegrad erreicht hat, beginnt der Tester mit Performance-Tests. Dabei sollten wenigstens die Bereiche, die durch Performance-Testfälle getestet werden sollen, schon erfolgreich auf ihre Funktionalität überprüft worden sein. Molyneaux weist auch darauf hin, dass auch schlechte Performance als funktionaler Fehler gewertet werden kann. Seine Argumentation basiert dabei auf dem Halteproblem. Wenn ein System auf eine Anfrage nie eine Antwort liefert, ist dies ein funktionaler Fehler, wenn es lediglich lange dauert, ist dies, je nach Dauer, ein Performance-Fehler. Da man aber nicht ewig auf eine etwaige Antwort des Systems warten kann, sollten die Entwickler eine Grenze bestimmen, ab wann solche Fehler als funktionale Fehler gewertet werden. Solche Fehler sollten möglichst vor dem Performance-Test ausgeschlossen werden.

Molyneaux [51] schlägt verschiedene Arten von Performance-Tests vor. Dabei rät er auch dazu, die Tests in einer bestimmten Reihenfolge durchzuführen. Tabelle 2.2 nennt die Testarten in der empfohlenen Reihenfolge. Zunächst wird dabei der „Baseline“-Test durchgeführt. Bei dieser Testart wird eine Referenz-Performance für spätere Tests aufgenommen. Dabei wird jede Anfragesequenz einzeln mit nur einem Benutzer durchgeführt. Hier liegt also ein sehr simples Lastprofil vor. In diesem Test soll die Performance in der günstigsten Situation festgestellt werden. Als Nächstes wird der Lasttest durchgeführt. Bei diesem Test wird die Anwendung mit einer definierten Last getestet, um zu überprüfen, ob die gewünschte Performance auch unter Last erreicht wird. Zunächst wird dabei jede Anfragesequenz einzeln bewertet. Dies minimiert die gegenseitige Beeinflussung von verschiedenen Anfragen. Somit wird die nachfolgende Fehleranalyse einfacher. Erst im nächsten Schritt wird das System mit einer Gruppe aus mehreren Anfragesequenzen, also einem möglichst realistischen Anfragemix, getestet. Für die Fehleranalyse schlägt Molyneaux, zusätzlich zu den Tabelle 2.2 aufgeführten Tests, einen Isolationstest vor. Beim Isolationstest versucht der Tester den Lasttest mit einer möglichst reduzierten Anfragesequenz bzw. Anfragesequenzgruppe auszuführen, bei der der Performance-Fehler dennoch auftritt. Der in dieser Arbeit vorgestellte Ansatz zur Performance-Blame-Analysis kann als Isolationstest mit besonderer Analyse zur Eingrenzung des Fehlers gesehen werden. Das heißt, dass Ansätze zur Performance-Blame-Analysis üblicherweise zur Fehleranalyse von Lasttests zum Einsatz kommen. Wie ein Isolationstest im Einzelnen durchgeführt

werden soll, wird von Molyneaux nicht näher erläutert. Im nächsten Schritt wird der Stabilitätstest durchgeführt. Der Stabilitätstest ist ein besonders lange laufender Lasttest, bei dem die Performance des Systems über einen langen Zeitraum beobachtet wird, um eine sich schleichend verschlechternde Performance erkennen zu können. Schließlich soll noch ein Stresstest ausgeführt werden. Beim Stresstest wird die Last schrittweise erhöht, bis zu dem Punkt, an dem das System oder seine Testumgebung an Grenzen stößt. So kann die Kapazität des Systems genau festgestellt werden.

Tabelle 2.2: Testarten und deren Reihenfolge im Performance-Test nach Molyneaux [51]

#	Testart	Lastprofil	Teststufe
1	Baseline	einzelne Anfragesequenz	Integrationstest
2	Lasttest	einzelne Anfragesequenz	Integrationstest
3	Lasttest	mehrere Anfragesequenzen	Integrations-/Systemtest
4	Stabilitätstest	mehrere Anfragesequenzen	Integrations-/Systemtest
5	Stresstest	mehrere Anfragesequenzen	Systemtest

Zusätzlich zu den Testarten und ihrer Abfolge zeigt Tabelle 2.2, ab welcher Teststufe das jeweilige Testverfahren eingesetzt werden kann. Dabei gilt generell, dass der Komponententest als grundsätzlicher Funktionstest vor dem Performance-Test abgeschlossen sein sollte. Wenigstens der zu testende Teil des Systems sollte implementiert und funktional getestet sein. Je größer also die Testabdeckung der Anfragesequenzen ist, desto größer ist der Teil des Systems der implementiert und getestet sein muss, um den Performance-Test durchzuführen. Die ersten drei Performance-Testarten können schon während des Integrationstests angewendet werden. Den Stabilitätstest kann man zwar schon während des Integrationstests verwenden, es ist allerdings empfehlenswert, damit bis zum Systemtest zu warten. Die Stabilität eines Systems lässt sich erst dann wirklich beurteilen, wenn es vollständig implementiert ist, da Wechselwirkungen zwischen den verschiedenen Komponenten auftreten können. Der Stresstest dient als Vorbereitung für die Auslieferung, da hier Anforderungen getestet werden, die über die Kundenanforderungen hinausgehen. Somit sollte der Stresstest erst ab dem Systemtest angewendet werden.

2.4.4 Test von komponentenbasierten Systemen

Im Folgenden werden einige Eigenarten des Tests von komponentenbasierten Systemen nach Gao et al. [26] vorgestellt. Zunächst einmal kommen die Komponenten aus unterschiedlichen Quellen. Sie können in derselben Organisation entwickelt werden wie das komponentenbasierte System oder aber von Dritten beschafft werden. Wenn eine Komponente von Dritten beschafft wird, so liegt der Quelltext meist nicht vor und auch keine anderen Artefakte, die nähere Angaben über die Interna der Komponente beinhalten. Eine richtige Fehleranalyse ist auf dieser Grundlage nicht möglich, so dass Fehlermeldungen bei einem Verdacht, dass die Komponente an dem Fehler beteiligt ist, an die jeweiligen Komponentenentwickler weitergegeben werden müssen. Die Analyse, ob die Komponente am Fehler beteiligt ist, nennt sich „Blame-Analysis“ (vgl. Abschnitt 3.1).

Wenn Komponenten von Dritten zum Einsatz kommen, so passen diese Komponenten häufig nicht direkt ins System. Im einfachsten Fall muss die Konfiguration einer solchen Komponente entsprechend angepasst werden. Es kann aber auch nötig sein, der Komponente einen Adapter vorzuschalten, der die Ein- und Ausgaben der Komponente bearbeitet. Dies kann soweit gehen, dass eine neue Komponente entsteht, die einen großen Teil der Arbeit erledigt und die ursprüngliche Komponente lediglich intern verwendet. In extremen Fällen kann es auch nötig sein, die beschaffte Komponente direkt zu verändern. Dafür ist natürlich der Zugang zum Quelltext der Komponente zwingend erforderlich. Jedes dieser Szenarien muss durch Tests adäquat abgedeckt werden.

Zudem setzen komponentenbasierte Systeme häufig auf implementierungsorientierten Komponentenmodellen auf. Diese Komponentenmodelle stellen eine Middleware zur Verfügung, die vorgibt, wie Komponenten implementiert werden. Des Weiteren übernimmt die Middleware häufig noch weitere Dienste für eine Komponente, wie zum Beispiel die Instanziierung und den Aufruf anderer Komponenten. Die Middleware vereinheitlicht dabei die Inanspruchnahme der Dienste und ermöglicht beispielsweise heterogene Komponentenlandschaften, in denen Komponenten über verschiedene Hardwareknoten verteilt sind oder aber in verschiedenen Programmiersprachen erstellt sind. Dabei hat die Middleware natürlich auch einen Einfluss auf die Performance und muss auch bei einem Performance-Test bzw. bei der Analyse eines Performance-Fehlers berücksichtigt werden.

Ein weiterer Aspekt ist, dass sich Komponenten unabhängig vom Gesamtsystem weiterentwickeln können. Die Entwickler bzw. Betreiber des Gesamtsys-

tems können sich dann dazu entschließen, die betreffenden Komponenten zu aktualisieren. Dies hat zwei Effekte. Zum einen kann die Aktualisierung einer Komponente die Aktualisierung weiterer Komponenten voraussetzen. Zum anderen kann der Entwickler bzw. Betreiber die alte Version der Komponente(n) noch verfügbar halten, so dass vorübergehend zwei verschiedene Systemversionen genutzt werden können. Diese Aspekte müssen auch beim Testen berücksichtigt werden.

Kapitel 3

Problemanalyse

Zunächst wird in Abschnitt 3.1 die Problemstellung Performance-Blame-Analysis durch Grundlagenliteratur motiviert und für diese Arbeit präzisiert. Die Präzisierung baut dabei auf den in Abschnitt 2.1 erläuterten Grundbegriffen auf. Dann werden aufgrund dieser Problembeschreibung Anforderungen an eine Lösung abgeleitet (s. Abschnitt 3.2). Verwandte Arbeiten, die diese oder eine ähnliche Problemstellung angehen, werden in Abschnitt 3.3 vorgestellt und im Hinblick auf die zuvor abgeleiteten Anforderungen bewertet. In Unterabschnitt 3.3.4 werden zudem Anforderungen an Visualisierungen von Performance-Blame-Analysis-Ergebnissen aufgestellt. Die verschiedenen Visualisierungen der verwandten Arbeiten sowie Visualisierung aus dem Bereich der Performance-Profiler werden dann im Hinblick auf diese Visualisierungsanforderungen bewertet. Zuletzt wird in Abschnitt 3.4 eine vergleichende Bewertung der verwandten Arbeiten durchgeführt. Dabei werden sowohl die vorgestellten Performance-Analyse-Ansätze nach den für sie geltenden Anforderungen untereinander verglichen als auch die Visualisierungen anhand der Visualisierungsanforderungen. Aufgrund dieser Analyse werden in Unterabschnitt 3.4.3 die verbleibenden Anforderungen für den in dieser Arbeit vorgestellten Ansatz PBlaman entwickelt.

3.1 Präzisierung Performance-Blame-Analysis

Zunächst wird im Folgenden die Problemstellung der „Blame-Analysis“ bzw. „Performance-Blame-Analysis“ mithilfe der Literatur erläutert und motiviert. Dazu erwähnen wir auf der einen Seite Literatur, die die Problemstellung Blame-Analysis bzw. Performance-Blame-Analysis kennen und motivieren. Um dies zu untermauern, wird auf der anderen Seite Literatur angeführt, die die Einbettung der Blame-Analysis bzw. Performance-Blame-Analysis in übliche Prozesse illustriert.

Eine grundsätzliche Erläuterung der „Blame-Analysis“ findet sich bei Crnkovic et al. [16]. Sie beschreiben zunächst die allgemeine Blame-Analysis als

eine Problemstellung, die in der Wartungsphase auftritt. Sie charakterisieren die Blame-Analysis als Analyse, welche Fehlerursache einer bestimmten Fehlerwirkung zugrunde liegt. Neben der eigentlichen Analyse sehen Crnkovic et al. auch das Problem zu klären, welche der unterschiedlichen Entwicklergruppen für die Fehleranalyse zuständig ist. Dabei müssen Systementwickler und verschiedene Gruppen von Komponentenentwicklern berücksichtigt werden.

Nach dieser Definition lässt sich feststellen, dass die Blame-Analysis eine Tätigkeit der Fehleranalyse bei komponentenbasierten Systemen ist. Das heißt, es wurde zunächst im Betrieb oder im Test ein Fehler festgestellt, dessen Ursache nun gefunden und beseitigt werden soll. Im Folgenden wird angenommen, dass der Fehler im Test aufgetreten ist oder dass das Fehlerszenario in einem Testfall niedergelegt werden kann. Damit liegt als Eingabe für die Blame-Analysis neben der Fehlerbeschreibung auch ein Testfall inklusive der Testumgebung vor, mit der die Testbedingung überprüft werden kann. Des Weiteren liegen die System-Spezifikation und die System-Implementierung vor. Die Hauptaufgabe der Blame-Analysis ist, zu klären, welche Entwicklergruppe sich mit dem Fehler befassen soll, um ihn zu beseitigen. Damit ist die Blame-Analysis an der Schnittstelle zwischen dem Test und der Entwicklung angesiedelt.

Wie schon festgestellt wurde, ist die Blame-Analysis eine Tätigkeit, die die Schnittstelle zwischen Test und Entwicklung betrifft. Spillner und Linz [69] erläutern, wie die Interaktion zwischen Testern und Entwicklern üblicherweise aussieht, wenn Fehler gefunden wurden. Eine vereinfachte Darstellung dieser Interaktion bietet Abbildung 3.1.[1] Der Tester gibt aufgrund des durchgeführten Tests eine Fehlermeldung nebst Testfall an die Entwickler weiter. Jede Fehlermeldung beschreibt einen entdeckten Fehler, während der Testfall genau angibt, bei welchen Eingaben und welchem System der Fehler auftritt. Die Entwickler analysieren und beheben den Fehler, aufgrund dieser Angaben. Schließlich unterrichten sie den Tester davon, dass eine Korrektur erarbeitet wurde, die dann nachgetestet wird.

Spillner und Linz [69] gehen von einer einzelnen Entwicklergruppe aus, an die der Fehler übergeben wird. Somit ist hier keine Blame-Analysis erforderlich. Im Falle eines komponentenbasierten Systems mit verschiedenen Entwicklergruppen muss nach dem Test und vor der Übergabe an die Entwickler eine

[1] Die Darstellung lässt die Rolle des Testmanagers aus, die die Fehlermeldung einer Qualitätsprüfung unterzieht und diese als unberechtigt markieren kann. Auch bei der Fehleranalyse kann sich herausstellen, dass die Fehlermeldung unberechtigt ist. Außerdem können sich die Entwickler entschließen, den Fehler zu beobachten bis weitere Erkenntnisse vorliegen.

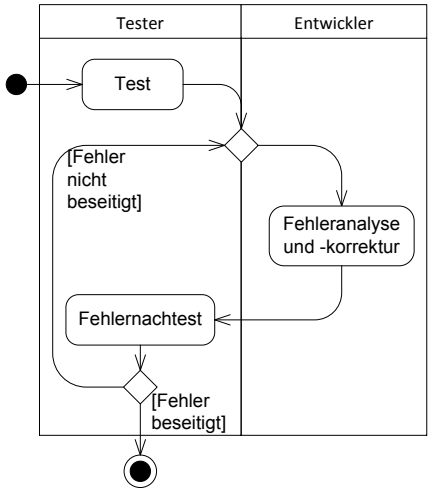

Abbildung 3.1: Interaktion zwischen Tester und Entwickler (vereinfacht) nach Spillner und Linz [69]

Blame-Analysis stattfinden, um zu steuern, welche Entwicklergruppe den Fehler behebt. Dieser Fall ist in Abbildung 3.2 illustriert. Der Systemarchitekt entscheidet im Rahmen der Blame-Analysis, welche Entwicklergruppe für die Fehlerbehebung zuständig sein soll. Dabei wird zunächst entschieden, ob zunächst die Systementwickler oder doch die Komponentenentwickler die Fehlerbehebung übernehmen sollen. Falls die Komponentenentwickler den Fehler beheben sollen, wird entschieden, welche Komponentenentwickler beteiligt werden sollen. Da es vorkommen kann, dass mehrere Komponenten fehlerhaft sind, kann es sinnvoll sein, verschiedene Komponentenentwickler gleichzeitig mit der Behebung zu beauftragen.

Bisher haben wir die Problemstellung der Blame-Analysis definiert und ihre Einbettung in den Entwicklungsprozess beleuchtet. Die Tätigkeiten während der Blame-Analysis sind noch nicht erläutert worden. Dazu ist es erforderlich sich auf eine bestimmte Fehlerart einzuschränken, da die Fehleranalyse je nach Fehlerart sehr unterschiedlich sein kann. Diese Arbeit soll die Blame-Analysis für Performance-Fehler, also die „Performance-Blame-Analysis“, erarbeitet werden. Gao et al. [26] bestätigen, dass die Problemstellung Blame-Analysis auch auf Performance-Fehler zutrifft. Die größte Herausforderung, die Gao et al. bei der Performance-Blame-Analysis identifiziert haben, ist, die Performance-Probleme bei einzelnen Komponenten zu erkennen.

Performance betrifft dabei die Metriken Antwortzeit, den Durchsatz und den Ressourcenverbrauch des Systems (vgl. ISO 9126 [35]). Diese Arbeit fokussiert

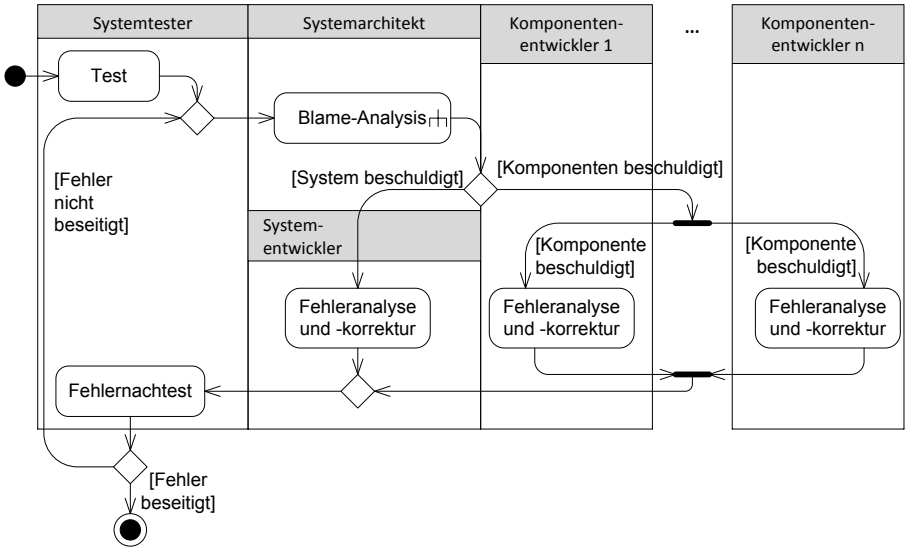

Abbildung 3.2: Die Blame-Analysis wird der Fehleranalyse vorgeschaltet, da bei komponentenbasierter Entwicklung unklar ist, welche der verschiedenen Entwicklergruppen verantwortlich ist.

sich auf die Performance-Metrik Antwortzeit. Die Metrik Antwortzeit eignet sich besonders für die Performance-Blame-Analysis, da sich Antwortzeiten explizit auf Anfragen an System- bzw. Komponentenoperationen beziehen. Damit ist auch der Bezug zu einer Komponente direkt hergestellt. Bei der Metrik Ressourcenverbrauch ist eine direkte Verbindung zu einer Komponente oder Komponentenoperation nur schwer herzustellen. Die Metrik Durchsatz hingegen kann aus den Antwortzeiten abgeleitet werden, so dass ohnehin ein Zusammenhang mit der Antwortzeit besteht. Daher wird das Problem der Performance-Blame-Analysis zunächst für die Metrik Antwortzeit angegangen und dann wird die Übertragbarkeit auf die anderen Metriken geprüft.

Wie schon beschrieben, ist es bei der Performance-Blame-Analysis besonders wichtig, die Performance einzelner Komponenten zu bewerten. Die Performance-Analyse verspricht Systembereiche mit Optimierungspotenzial, z. B. bezüglich der Antwortzeit, ausfindig zu machen. Damit ließe sich also das Optimierungspotenzial von Komponenten identifizieren und so eine Performance-Blame-Analysis durchführen. DeRose et al. [17] haben eine typische Vorgehensweise für die Performance-Analyse skizziert. Demnach gliedert sich eine Performance-Analyse in fünf Schritte: 1.) Instrumentierung, 2.) Messung, 3.) Analyse, 4.) Präsentation und 5.) Optimierung. Im Instrumentierungsschritt werden Messfühler installiert, mit denen im zweiten Schritt

die Performance gemessen wird. In der Analyse werden die Messergebnisse verdichtet und analysiert, so dass das gesuchte Performance-Problem zutage tritt. Im Präsentationsschritt werden die Analyse-Ergebnisse z. B. in Tabellen oder Grafiken aufbereitet, so dass sie intuitiv verständlich sind. Im letzten Schritt wird das System so optimiert, dass das beobachtete Problem nicht mehr auftritt.

Der von DeRose beschriebene Prozess kann in abgewandelter Form als Blaupause für die Performance-Blame-Analysis dienen. Die Performance-Blame-Analysis erhält die Fehlermeldung und den Testfall als Eingabe. Dabei enthält der Testfall unter anderem die Testumgebung und er identifiziert die getestete Systemimplementierung. Trotzdem kann eine zusätzliche Instrumentierung nötig sein, um über den Test hinausgehende Analysen zu ermöglichen. Die Messung ist dann erforderlich, wenn die gewünschten Daten nicht schon im Test gewonnen wurden. Das Herzstück des Prozesses bildet die Analyse und die Präsentation. Hier werden die Daten so aufbereitet, so dass der Systemarchitekt einfach entscheiden kann, welche Entwicklergruppe zur Fehlerbehebung beitragen kann. Die Optimierung findet in der Performance-Blame-Analysis nachgelagert im Rahmen der Fehlerbehebung statt (vgl. Abbildung 3.2). Stattdessen entscheidet der Systemarchitekt im letzten Prozessschritt aufgrund der aufbereiteten Analyseergebnisse, welche Entwicklergruppe(n) die weitere Fehlerbehebung übernehmen. Damit ergibt sich der in Abbildung 3.3 angegebene Unterprozess, der im Rahmen der Interaktion zwischen Tester und Entwicklern von einem Systemarchitekten ausgeführt wird (vgl. Abbildung 3.2).

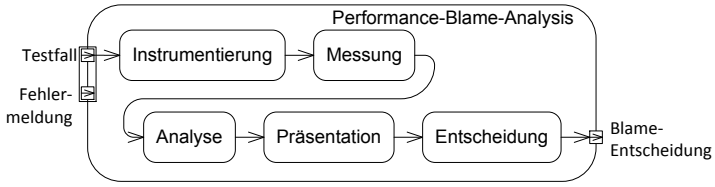

Abbildung 3.3: Die Tätigkeiten in der Aktivität Performance-Blame-Analysis in Anlehnung an den Performance-Analyseprozess von DeRose [17]

Die Performance-Blame-Analysis wird immer auf der Seite der Systementwicklung vorgenommen (vgl. Abbildung 3.2), idealerweise vom Systemarchitekten. Da der Systemarchitekt sowohl die Anforderungen kennt als auch den Kontakt zu den Komponentenentwicklern hat, ist er besonders geeignet für diese Aufgabe. Wie in Abbildung 3.3 angegeben, ist das Ergebnis der

Performance-Blame-Analysis die Entscheidung des Systemarchitekten, ob und welche einzelnen Komponenten für den Fehler verantwortlich sind oder ob die Verwendung bzw. Interaktion der Komponenten im System den Fehler auslöst. Aus der Entscheidung des Systemarchitekten lässt sich dann folgern, welche Entwicklergruppe die weitere Fehleranalyse vornehmen sollte.

In diesem Abschnitt wurde erläutert, was eine Performance-Blame-Analysis ist, wozu sie dient, wer sie durchführt und was ihre Ein- und Ausgaben sind. Darauf aufbauend werden im Abschnitt 3.2 Anforderungen an Performance-Blame-Analysis-Ansätze abgleitet.

3.2 Anforderungen an Performance-Blame-Analysis-Ansätze

Im Folgenden werden Anforderungen an Ansätze zur Performance-Blame-Analysis aufgestellt. Die Anforderungen sollen zunächst dazu dienen existierende Ansätze, mit denen eine Performance-Blame-Analysis möglich ist, zu bewerten (s. Abschnitt 3.3). Anschließend sollen auf Grundlage dieser Bewertung die noch verbleibenden Anforderungen für die Performance-Blame-Analysis identifiziert werden (s. Unterabschnitt 3.4.3). Dabei stützen sich die Anforderungen auf die Präzisierung der Performance-Blame-Analysis, wie sie in Abschnitt 3.1 vorgenommen wurde. Der grobe Prozessablauf eines Performance-Blame-Analysis-Ansatzes wurde in Abbildung 3.3 dargestellt und im Folgenden werden Anforderungen für die Bereiche Analyse, Präsentation und Entscheidung erarbeitet. Diese Arbeit setzt voraus, dass Instrumentierung und Messung gewährleistet werden können. Der Fokus der Anforderungen liegt auf den Schritten Analyse und Präsentation. Der Schritt Entscheidung wird immer manuell durch den Systemarchitekten bearbeitet. Damit sind die Anforderungen für diesen Schritt derart, dass er die Entscheidung auf Grundlage der Artefakte aus den vorangegangenen Schritten korrekt und effizient treffen kann. Somit wirken die Anforderungen an den Entscheidungsschritt primär zurück auf die vorhergehenden Schritte.

In der Präzisierung der Performance-Blame-Analysis (s. Abschnitt 3.1) heißt es, dass die Performance-Blame-Analysis dazu dient, eine Fehlermeldung zur weiteren Analyse und zur Behebung eines Performance-Fehlers an die richtige Entwicklergruppe weiterzugeben. Eine Fehlermeldung beschreibt eine Fehlerwirkung, die bei der Ausführung eines bestimmten Testfalls beobachtet worden ist. Die Performance-Blame-Analysis muss sich auf diesen Testfall beziehen:

Anforderung A1 (muss):
Die Performance-Blame-Analysis bezieht sich auf einen bestimmten Testfall.

Außerdem muss die Performance-Blame-Analysis entscheiden, welche Entwicklergruppe den Fehler weiter untersuchen soll. Die Übergabe des Fehlers an die passende Entwicklergruppe zur Korrektur ist in Abbildung 3.2 dargestellt. Die Performance-Blame-Analysis muss also dazu in der Lage sein, den Fehler entweder den Systementwicklern oder aber den verschiedenen Komponentenentwicklergruppen zuordnen zu können. Damit ergeben sich folgende zwei Anforderungen, die erfüllt sein müssen:

Anforderung A2 (muss):
Die Performance-Blame-Analysis kann die Performance einzelner Komponenten im Bezug auf einen Testfall bewerten. Damit lässt sich der Fehler einer oder mehreren Komponentenentwicklergruppen zuordnen.

Anforderung A3 (muss):
Die Performance-Blame-Analysis kann sicher entscheiden, ob der Fehler von den Systementwicklern und nicht von den Komponentenentwicklern korrigiert werden soll.

In der Präzisierung heißt es, dass die Performance-Blame-Analysis der eigentlichen Fehleranalyse vorgeschaltet ist. Die resultierenden Analyseartefakte dürfen schon weiterführende Ergebnisse liefern, die nicht zwingend für die Zuordnung zur passenden Entwicklergruppe erforderlich sind. Damit lassen sich die folgenden optionalen Anforderungen formulieren, die die Anforderungen A2 und A3 weiterführen:

Anforderung A4 (optional):
Die Performance-Blame-Analysis kann die Performance einzelner Komponentenoperationen im Bezug auf einen bestimmten Testfall bewerten.

Anforderung A5 (optional):
Die Performance-Blame-Analysis kann die Verwendung der Komponenten im Gesamtsystem im Bezug auf einen bestimmten Testfall näher bewerten. Die Analyse beurteilt also, ob beispielsweise die Komposition oder die Komponentenverteilung für die untersuchte Fehlerwirkung verantwortlich ist.

In den bisherigen Anforderungen wurde auf die Anwendung eines Performance-Blame-Analysis-Ansatzes bei einem komponentenbasierten System eingegangen. Idealerweise umfasst diese Analyse direkt die Bewertung von Komponenten und Komponentenoperationen. Viele herkömmliche Performance-Analyse-Ansätze sind aber nicht auf komponentenbasierte Systeme spezialisiert. Ihre Ergebnisse beziehen sich beispielsweise auf Funktionsaufrufe oder Objekte. Diese Ergebnisse sind nicht nutzlos, müssen aber manuell vom Systemarchitekten auf Komponenten abgebildet werden. Somit sollte idealerweise die folgende Anforderung erfüllt sein:

Anforderung A6 (optional):
Die Performance-Blame-Analysis kennt Komponentenoperationen und Komponenten und führt die Analyse für diese Elemente durch.

Je nach Herkunft der Komponente steht ihr Quelltext bei der Analyse möglicherweise nicht zur Verfügung. Eine Performance-Blame-Analysis soll Komponenten unabhängig von der Verfügbarkeit ihres Quelltexts analysieren können. Insbesondere muss eine Performance-Blame-Analysis mit Komponenten zusammenarbeiten, deren Quelltext nicht vorliegt:

Anforderung A7 (muss):
Die Performance-Blame-Analysis bewertet auch die Performance von Komponenten, deren Quelltext nicht vorliegt.

Ein Ansatz zur Performance-Blame-Analysis soll den Systemarchitekten bei seiner Tätigkeit möglichst gut unterstützen. Für den Systemarchitekten ist es hilfreich, wenn der Ansatz weitestgehend automatisiert abläuft. Dabei sind die Bereiche Instrumentierung und Messung zur Erhebung von Daten zur Laufzeit schon vielfach behandelt worden. Zum Beispiel Magpie [1] und Kieker [73, 74] bieten entsprechende Möglichkeiten der Instrumentierung und Messung der erforderlichen Daten. Zudem geben auch Sambasivan et al. [63] noch weitere Beispiele von Ansätzen, die sich mit dem Thema beschäftigen. Also ist hier die entscheidende Frage, ob die Analyse und die Präsentation der Ansätze hinreichend automatisiert sind:

Anforderung A8 (optional):
Die Schritte Analyse und Präsentation der Performance-Blame-Analysis sollen weitestgehend automatisiert ablaufen.

Die in Anforderung A8 thematisierte Automatisierung kann den Systemarchitekten nur zu einem Teil entlasten. Vor der Anwendung der Performance-Blame-Analysis muss der Systemarchitekt häufig noch Eingaben für den

verwendeten Ansatz sammeln. Es ist wünschenswert, dass dieser Aufwand niedrig ist und dass sich Eingaben wiederverwenden lassen:

Anforderung A9 (optional):
Die Eingaben für den Performance-Blame-Analysis-Ansatz können mit wenig Aufwand erstellt werden und können wiederverwendet werden.

Die angesprochene Performance-Bewertung durch die Performance-Blame-Analysis muss dem Systemarchitekten präsentiert werden, damit dieser abschließend entscheiden kann, welche Entwicklergruppe die weitere Fehleranalyse und -behebung vornehmen soll. Dazu sollen die Ergebnisse in einem Ergebnisdokument zusammengefasst werden. Das Ergebnisdokument soll aufbereitete Tabellen und Grafiken umfassen, die die Analyseergebnisse anschaulich präsentieren. Die Analyseergebnisse werden üblicherweise die Performance-Daten zusammenfassen. Dies kann mithilfe von statistisch zusammengefassten Werten wie Mittelwert oder Median geschehen oder aber mithilfe von klaren Empfehlungen, welche Komponenten für die Fehlerwirkung verantwortlich zu machen sind. Daraus ergibt sich die nächste Anforderung:

Anforderung A10 (muss):
Die Analyseergebnisse der Performance-Blame-Analysis werden in einem Ergebnisdokument zusammengefasst.

Für den Systemarchitekten ist dieses Ergebnisdokument besonders hilfreich, wenn dort direkt zu lesen ist, welche Komponenten nach der Analyse für den Fehler verantwortlich zu machen sind. Daher fordert die nächste Anforderung diesbezüglich eine klare Ja/Nein-Entscheidung:

Anforderung A11 (optional):
Im Ergebnisdokument ist klar dokumentiert, ob und wenn ja welche Komponenten (bzw. welche Komponentenoperationen) durch die Performance-Blame-Analysis als mögliche Ursachen für die untersuchte Fehlerwirkung identifiziert wurden.

Bei großen Systemen mit vielen Komponenten oder komplexen Testfällen, in denen viele unterschiedliche Komponentenoperationen zum Einsatz kommen, kann der Systemarchitekt im Wust der Ergebnisse schnell den Überblick verlieren. Hier gilt es, sorgfältig die passendsten Darstellungen textueller oder grafischer Natur auszuwählen, um ein übersichtliches Ergebnisdokument zu erstellen. Dabei sollte zum Beispiel neben Detaildarstellungen auch eine

Übersichtsdarstellung präsentiert werden, mit deren Hilfe sich der Systemarchitekt schnell einen Überblick über die Ergebnisse verschaffen kann. Daraus folgt die optionale Anforderung:

Anforderung A12 (optional):
Die Analyseergebnisse der Performance-Blame-Analysis werden im Ergebnisdokument übersichtlich dargestellt, damit der Systemarchitekt die Ergebnisse schnell erfassen kann.

Schließlich muss der Systemarchitekt entscheiden, welche Komponenten er für die Ursache der untersuchten Fehlerwirkung hält. Daher ist es hilfreich, wenn die etwaige Empfehlung der Performance-Blame-Analysis nicht isoliert präsentiert wird, sondern auch die Entscheidungsgrundlage der Analyse gezeigt wird. So kann sich der Systemarchitekt bei Zweifeln selbst von der Richtigkeit der Ergebnisse überzeugen:

Anforderung A13 (optional):
Entscheidungsrelevante Performance-Metriken werden mit in die Ergebnispräsentation einbezogen, um eine etwaige Ja/Nein-Entscheidung der Performance-Blame-Analysis nachvollziehbar zu machen.

Die genannten Anforderungen sollte ein idealer Performance-Blame-Analysis-Ansatz erfüllen. Im Abschnitt 3.3 wird nun untersucht, ob und zu welchem Grad diese Anforderungen schon von existierenden Ansätzen erfüllt werden. Zuletzt wird dann in Abschnitt 3.4.3 beschrieben, welche der Anforderungen die verwandten Ansätze noch offen lassen. Diese Verbesserungspotenziale gilt es in dieser Arbeit auszuschöpfen.

3.3 Verwandte Arbeiten

Zuvor wurde der Begriff der Performance-Blame-Analysis erläutert und für diese Arbeit präzisiert (vgl. Abschnitt 3.1). In diesem Abschnitt werden verwandte Arbeiten in relevanten Bereichen beschrieben, die sich als Performance-Blame-Analysis-Ansätze eignen. Dabei werden Ansätze beschrieben, die die Schritte „Analyse“ und „Präsentation“ einer Performance-Blame-Analysis unterstützen (vgl. Abbildung 3.3). Diese verwandten Arbeiten werden im Hinblick auf die in Abschnitt 3.2 erarbeiteten Anforderungen bewertet. Damit lässt sich der aktuelle Stand der Forschung im Hinblick auf die Problemstellung der Performance-Blame-Analysis erfassen und es lassen sich noch verbliebene Anforderungen ausmachen (s. Abschnitt 3.4.3).

Die Forschung zum Problem der Performance-Blame-Analysis speist sich im Wesentlichen aus dem Bereich der Performance-Analyse aufgrund von Ablaufverfolgungsprotokollen. Dieser Bereich gliedert sich in drei Kategorien:

1. Analyse eines einzelnen Ablaufverfolgungsprotokolls
2. Analyse durch Vergleich zweier Ablaufverfolgungsprotokolle
3. Analyse durch Vergleich eines Ablaufverfolgungsprotokolls mit einer Spezifikation

Die erste Kategorie der Performance-Analyse-Ansätze arbeitet auf einem Ablaufverfolgungsprotokoll und versucht, Besonderheiten darin zu entdecken. Dies können generell Komponenten oder Komponentenoperationen mit hoher Antwortzeit sein oder aber einzelne Aufrufe, die entgegen ihrem zumeist beobachteten Verhalten eine hohe Antwortzeit aufweisen. In der zweiten Kategorie wird ein Ablaufverfolgungsprotokoll mit einem anderen verglichen. Dabei dient ein Ablaufverfolgungsprotokoll als Normalzustand und das andere Protokoll wird im Vergleich dazu auf Abweichungen untersucht. Ähnlich wird auch in der dritten Kategorie vorgegangen. Doch hier gilt eine vorgegebene Spezifikation als erwartetes Ergebnis und das Ablaufverfolgungsprotokoll wird damit verglichen.

Im Folgenden werden die Arbeiten nach diesen Kategorien gegliedert aufgegriffen (s. Abschnitte 3.3.1 bis 3.3.3). Dabei wird der jeweilige Ansatz zunächst erläutert. Dann wird geklärt, welchen Beitrag der jeweilige Ansatz zu den angegebenen Anforderungen (s. Abschnitt 3.2) an die Performance-Blame-Analysis liefert. Anschließend wird in Abschnitt 3.3.4 gesondert auf die Visualisierung von Performance-Daten im Bereich der Performance-Analyse, des Performance-Profiling sowie der Softwarekartografie eingegangen. Zuletzt werden die präsentierten Ansätze vergleichend im Bezug auf die Anforderungen bewertet (s. Abschnitt 3.4), so dass Gemeinsamkeiten und Unterschiede deutlich werden.

3.3.1 Analyse eines einzelnen Ablaufverfolgungsprotokolls

Im Folgenden werden Ansätze vorgestellt, die ein einzelnes Ablaufverfolgungsprotokoll analysieren. Dabei wird auch geprüft, wie gut der jeweilige Ansatz die in Abschnitt 3.2 erarbeiteten Anforderungen umsetzt.

Ein Performance-Analyse-Ansatz für Entwickler wurde von Srinivas und Srinivasan [70] vorgestellt. Dieser Ansatz analysiert die Aufrufe von Methoden mit ihren kausalen Abhängigkeiten. Die Aufrufbeziehungen, also die kausalen Abhängigkeiten, werden in einem Aufrufkontextbaum dargestellt. Der Ansatz aggregiert die beobachteten Antwortzeiten in seiner Analyse zu Komponentenoperationswerten. Dazu muss der Systemarchitekt für jede Komponente vorgeben, welche Klasse als Stellvertreter für die jeweilige Komponente steht. In der Analyse werden dann alle Aufrufe, die nicht an Komponentenstellvertreter gerichtet sind, wie interne Berechnungen gehandhabt. Der Ansatz annotiert den Aufrufkontextbaum mit den jeweiligen Antwortzeiten (der Ansatz spricht hier allgemein von Kosten, da auch andere Metriken wie z. B. CPU-Zeit möglich sind). Dabei werden die Antwortzeiten relativ zur Gesamtantwortzeit des initialen Aufrufs, also der Wurzel des Aufrufkontextbaums, angegeben. Nun werden, wie angesprochen, die Antwortzeiten für die Komponentenoperationen berechnet, indem die Antwortzeit der von ihnen aufgerufenen internen Methoden hinzugerechnet wird. Die resultierenden relativen Antwortzeiten je Komponentenoperation stellen das Ergebnis der Analyse dar. Der Ansatz stellt diese relativen Antwortzeiten, dann in einer Rangliste der höchsten Antwortzeiten mit dem jeweiligen Operationsnamen tabellarisch dar. In der Rangliste sind die Komponentenoperationen aufgelistet, deren relative Antwortzeit einen vom Systemarchitekten bestimmten Schwellwert überschreitet. Der Schwellwert dient also vornehmlich dazu, die Anzahl der präsentierten Ergebnisse zu beschränken.

Der Ansatz von Srinivas und Srinivasan [70] bewertet die Messdaten, die bei einer bestimmten Testfallausführung beobachtet wurden (s. Anforderung A1). Der Ansatz berechnet dazu relative Antwortzeiten für die vom Entwickler angegebenen Komponenten und bringt die höchsten relativen Antwortzeiten in eine Rangliste ein. Damit kann der Ansatz die Performance für Komponenten und Komponentenoperationen bewerten (s. Anforderungen A2 und A4). Allerdings verlässt sich der Ansatz darauf, dass die zu beschuldigenden Komponenten auch eine hohe Antwortzeit aufweisen und somit oben in der Rangliste zu finden sind. Wenn das nicht der Fall ist, kann der Ansatz mit seiner Rangliste nur wenig Hilfestellung leisten. Da der Ansatz lediglich hohe Antwortzeiten von Komponentenoperationen feststellt, kann niemals ausgeschlossen werden, dass Komponentenoperationen die Ursache der Fehlerwirkung sind (s. Anforderung A3). Eine weitergehende Bewertung der Verwendung von Komponenten beispielsweise in Komposition und Komponentenverteilung erfolgt nicht (s. Anforderungen A5). Der Ansatz kann nur eingeschränkt mit Komponenten umgehen, die nicht im Quelltext vorliegen (s. Anforderung A7). Er baut auf einem Aufrufkontextbaum auf Klassenebene

auf und ist somit darauf angewiesen, dass dieser, zum Beispiel durch einen Profiler, erhoben werden kann. Die im Ansatz realisierte Performance-Analyse ist automatisiert (s. Anforderung A8), stützt sich allerdings auf entscheidende Eingaben. Der Entwickler muss zunächst die als Komponenten zu untersuchenden Klassen angeben (s. Anforderung A6). Zusätzlich muss auch ein Schwellwert für relative Antwortzeiten festlegt werden. Der Schwellwert hat dabei einen entscheidenden Einfluss auf die Ausgabe und beeinflusst die Länge der ausgegebenen Antwortzeitrangliste maßgeblich. Die Eingaben des Ansatzes sind mit einigem Aufwand zu ermitteln und die Angabe, welche Klassen als Komponenten gelten, kann wiederverwendet werden (s. Anforderung A9). Der Schwellwert kann zwar wiederverwendet werden, muss aber ggf. bei jeder Verwendung des Ansatzes angepasst werden. Die Antwortzeitrangliste, das Ergebnisdokument des Ansatzes (s. Anforderung A10), listet lediglich die Komponentenoperationen mit ihren relativen Antwortzeiten tabellarisch auf. Der Entwickler muss selbst untersuchen, ob eine Komponentenoperation zur untersuchten Fehlerwirkung beiträgt (s. Anforderung A11). Dabei bietet die Antwortzeitrangliste zwar eine Übersicht, weitere Details lassen sich aber nicht einblenden (s. Anforderung A12). Die Ergebnisse des Ansatzes beschränken sich auf die Antwortzeitrangliste. Somit finden sich im Ergebnis keine zusätzlichen entscheidungsrelevanten Performance-Metriken (s. Anforderung A13). Es ergibt sich damit für diesen Ansatz die folgende Bewertung:

Anforderung	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
Srinivas et al. [70]	+	o	-	o	-	+	o	+	o	+	-	-	-

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Der Ansatz „Magpie“ wurde von Barham et al. vorgestellt [1]. Magpie befasst sich damit, wie man aus Ablaufverfolgungsprotokollen Anfragepfade isolieren kann und diese wiederum sinnvoll zu Clustern zusammenfassen kann. Zudem ordnet Magpie den Anfragepfaden noch den dabei angefallenen Ressourcenverbrauch zu. Magpie verlässt sich für Instrumentierung und Messung auf die Monitoring-Lösung „Event Tracing for Windows“. Für die Analyse setzt Magpie auf Ablaufverfolgungsprotokolle mit dessen Datenformat auf. Das Datenformat erlaubt verschiedene Beobachtungen (z. B. für eine SQL-Anfrage oder eine COM-Komponentenanfrage) mit jeweils eigenen Parametern. Dabei verlässt sich der Ansatz darauf, dass alle Komponenten und die für deren Ausführung nötige Ausführungsumgebung ausreichend instrumentiert sind. Die Monitoring-Lösung bringt als Grundlage eine Instrumentierung des Betriebssystems mit. Ausgehend vom Ablaufverfolgungsprotokoll erkennt Magpie die

Anfragepfad-Zugehörigkeit anhand der zeitlichen Abfolge und anhand von sogenannten Ereignisschemata (engl. „event schema“). Die Ereignisschemata müssen dabei wenigstens für jede beteiligte Ausführungsumgebung vorliegen, z. B. für den Web-Server und den SQL-Server. Ein Ereignisschema gibt an, welche Parameter (z. B. Thread-ID oder Session-ID) unterschiedlicher Beobachtungen bei übereinstimmenden Werten einen Zusammenhang identifizieren. Dabei muss der Zusammenhang keine Aufrufbeziehung sein. Beispielsweise kann ein Zusammenhang beschreiben, dass ein Thread Daten von einer Netzwerkverbindung empfängt. Die Zusammenhänge werden durch Magpie explizit gemacht, es entsteht aber kein Aufrufbaum, da hier nicht nur Aufrufe berücksichtigt werden und auch keine konsistente Klammerung besteht. Vielmehr muss von einem Zusammenhangsgraphen gesprochen werden. Magpie fokussiert auf die Analyse von Ressourcenverbräuchen und isoliert im nächsten Schritt Thread-Synchronisation und Ressourcenverbräuche je Anfrage. Diese beiden Informationen werden zusätzlich von Scheduling-Artefakten, die aufgrund von konkurrierenden Zugriffen entstehen, befreit. Die so entstandenen Ressourcenverbrauchsgraphen werden zuletzt noch aufgrund ähnlichen Ablaufs und Ressourcenverbrauchs zu Clustern zusammengefasst. Mithilfe der Cluster lassen sich dann selten vorkommende Zusammenhangsgraphen isolieren und analysieren. Als Resultat können die verschiedenen Zusammenhangsgraphen grafisch dargestellt werden. In der Auswertung zeigen Barham et al. auch, welche Zusammenhangsgraphencluster Magpie erstellt und ordnen diese beispielsweise nach angefragter URL.

Da sich Magpie [1] auf ein Ablaufverfolgungsprotokoll stützt, besteht ein klarer Testfallbezug (s. Anforderung A1). Magpie kann seltene Ressourcenverbrauchsmuster und Abläufe in der Anfragebearbeitung durch die Cluster erkennen. Die darin enthaltenen Aufrufe der Komponentenoperationen sind ggf. für die untersuchte Fehlerwirkung verantwortlich (s. Anforderung A4). Allerdings muss der Systemarchitekt manuell in den von Magpie erfassten Beobachtungen recherchieren, welche Komponentenoperationen aufgerufen wurden. Von den Komponentenoperationen lässt sich auf die Komponenten schließen, aber eine explizite Übertragung auf Komponentenebene findet nicht statt (s. Anforderung A2). Seltenes Verhalten ist aber kein sicherer Indikator dafür, dass die Komponentenoperation für die Fehlerwirkung verantwortlich ist. Somit lässt sich dies für die Komponentenoperationen auch nie wirklich ausschließen (s. Anforderung A3). Eine Untersuchung der Komponentenverwendung ist zwar gegeben, wird aber nicht direkt ausgewiesen (s. Anforderung A5). Es bedürfte einer weiteren Untersuchung der Magpie-Ergebnisse, um beispielsweise Aussagen über die Wirkung der Komponentenverteilung oder die Wechselwirkungen mit dem Betriebssystem treffen zu

können. Magpie kann mit Komponenten, die nicht im Quelltext vorliegen, umgehen, da Magpie lediglich die Instrumentierung der beteiligten Komponenten auf Ausführungsumgebungsebene erfordert (s. Anforderung A7). Durch die Instrumentierung der Ausführungsumgebung sind auch die in der Ausführungsumgebung anzutreffenden Entitäten bekannt, so dass Magpie Komponenten kennen kann (s. Anforderung A6). Dies hängt aber davon ab, ob die Komponenten in der Instrumentierung erfasst und protokolliert werden. Der Magpie-Ansatz wird vollständig von einer Werkzeugkette abgedeckt (s. Anforderung A8). Magpie benötigt die Ereignisschemata als Eingabe. Diese sind im Wesentlichen von der eingesetzten Ausführungsumgebung abhängig und können für diese vorformuliert und wiederverwendet werden (s. Anforderung A9). Magpie setzt neben den Ereignisschemata lediglich ein Ablaufverfolgungsprotokoll im dazu passenden Datenformat voraus. Die Ausgabe von Magpie umfasst Statistiken über die Cluster und eine Visualisierung von Zusammenhangsgraphen (s. Anforderung A10). Dabei werden entweder Zusammenhangsgraphen mit allen Details dargestellt oder aber nur Zusammenhangsgraphen, die die Thread-Synchronisation und den Ressourcenverbrauch beinhalten. Magpie bietet sowohl eine Übersicht über die Cluster als auch eine Darstellung der Zusammenhangsgraphen, die die Übersicht weiter detaillieren. Beides ist sehr detailreich und daher nicht sehr übersichtlich (s. Anforderung A12). Je Cluster werden die Anzahl der Elemente, sein Durchmesser und ein stellvertretender Zusammenhangsgraph angegeben. Eine Bewertung bezüglich der Fehlerursache muss manuell erfolgen (s. Anforderung A11). Den Anfragepfaden wird zusätzlich ihr Ressourcenverbrauch zugeordnet (s. Anforderung A13). Es ergibt sich damit die folgende Bewertung:

Anforderung	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
„Magpie“ [1]	+	o	-	o	o	o	+	+	o	+	-	o	+

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Ein weiterer Performance-Analyse-Ansatz für Entwickler wurde von Rutar und Hollingsworth [62] vorgestellt. Dieser Ansatz berechnet einen aggregierten Antwortzeitwert (die Analyse anderer Metriken ist auch möglich), den sogenannten „Blame“-Wert, für einzelne Variablen und Datenstrukturen im Quelltext basierend auf dem Datenfluss. Der Blame-Wert fasst dabei den Anteil an der Antwortzeit zusammen, die durch Operationen auf der Variablen bzw. Datenstruktur verbraucht wurde. Die Messung des Zeitverbrauchs erfolgt dabei auf der detailliertesten Ebene. Jede Operation auf einer Variablen, z. B. eine Addition mit einer Konstanten oder einer andere Variable, erhöht

den Blame-Wert dieser Variablen. Auch Schleifen und bedingte Anweisungen haben implizit durch mehrfache Ausführung oder Auslassung von Anweisungen Einfluss auf den Datenfluss und tragen zu den Blame-Werten bei. Um den Blame-Wert für Datenstrukturen zu ermitteln, werden die Blame-Werte für die Datenstrukturteile aggregiert. Diese Blame-Werte werden mithilfe von statischer Analyse und mit Messungen zur Laufzeit ermittelt. Das Ergebnis dieses Ansatzes ist eine oder mehrere Ranglisten mit absteigenden Blame-Werten von Variablen an je einem bestimmten Punkt im Quelltext. Dabei umfasst die jeweilige Rangliste alle an diesem Punkt sichtbaren Variablen.

Mit der Messung zur Laufzeit stellt der Ansatz von Rutar und Hollingsworth einen Bezug zu einem bestimmten Testfall her (s. Anforderung A1). Die Analyse erfolgt auf dem Niveau der Variablen und Datenstrukturen. Eine Aggregierung erfolgt noch bis zur Funktionsebene, so dass sich manuell die Blame-Werte für Komponentenoperationen ermitteln lassen (s. Anforderung A4). Dieser Ansatz kennt jedoch keine Komponenten (s. Anforderung A6), daher ist eine Übertragung der Blame-Werte auf Komponenten nicht vorgesehen (s. Anforderung A2). Um Erkenntnisse über Komponenten und deren Verwendung in Gesamtsystem zu erhalten, wäre eine weitere Analyse nötig (s. Anforderungen A3 und A5). Da der Ansatz auf der statischen Analyse fußt, ist eine Analyse von Komponenten, deren Quelltext nicht vorhanden ist, nicht sinnvoll (s. Anforderung A7). Die Analyse ist weitestgehend automatisiert (s. Anforderung A8). Der Entwickler muss lediglich zuvor die gewünschte Performance-Metrik (z. B. Antwortzeit oder CPU-Zeit) festlegen und er kann weitere optionale Einstellungen vornehmen (s. Anforderung A9). Das Ergebnis des Ansatzes umfasst eine oder mehrere Blame-Wert-Ranglisten mit allen an diesem Quelltext-Punkt sichtbaren Variablen (s. Anforderung A10). Eine klare Bewertung, ob die Variable für die untersuchte Fehlerwirkung verantwortlich ist, erfolgt nicht (s. Anforderung A11). Das Ergebnis stellt nur den Blame-Wert je Variable dar, so dass es schwierig ist, eine Bewertung von Komponenten und Komponentenoperationen vorzunehmen, wenn man nicht deren Interna kennt. Die Rangliste bietet lediglich eine Übersicht über diese Blame-Werte ohne weitere Details einblenden zu können (s. Anforderung A12). Ebenso wird das Ergebnis nicht mit weiteren wichtigen Performance-Metriken kombiniert (s. Anforderung A13). Es ergibt sich damit die folgende Bewertung:

Anforderung	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
Rutar et al. [62]	+	-	-	o	-	-	-	+	+	+	-	-	-

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

3.3.2 Analyse durch Vergleich zweier Ablaufverfolgungsprotokolle

Im Folgenden werden Ansätze vorgestellt, die zwei Ablaufverfolgungsprotokolle miteinander vergleichen. Ein Protokoll aus vorherigen Testläufen dient als Referenz für ein aktuelles Protokoll. Nach der Beschreibung des jeweiligen Ansatzes wird geprüft, wie gut der jeweilige Ansatz die in Abschnitt 3.2 erarbeiteten Anforderungen umsetzt.

Der Ansatz namens „RanCorr“ von Marwede et al. [48] führt den Vergleich der zwei Ablaufverfolgungsprotokolle nicht selbst durch, sondern überträgt die Ergebnisse eines solchen Vergleichs auf Operationen, Komponentenobjekte und Hardwareknoten. Im Ansatz wird ein Protokollkomparator (im Paper „Anomaliedetektor“ genannt) eingesetzt, der feststellt, ob das Verhalten der jeweiligen Operationen im aktuellen Protokoll vom Referenzprotokoll abweicht und dies als Zahlwert angibt. Laut der Autoren können verschiedene Protokollkomparatoren verwendet werden. In ihrer Evaluierung setzen die Autoren eine Variante des Protokollkomparators von Rohr et al. [60] ein. Dabei überträgt RanCorr die Ergebnisse dieses Protokollkomparators auf verschiedene Elemente der Software. RanCorr berücksichtigt die kausalen Beziehungen zwischen den Operationsaufrufen und arbeitet auf einem Aufrufgraphen. Zusätzlich werden die Operationsaufrufe auf die Ebene der Komponentenobjekte und der Hardwareknoten abstrahiert. Dabei gilt, wenn Operation A aus Komponentenobjekt K1 die Operation B aus Komponentenobjekt K2 aufruft, so ruft das Komponentenobjekt K1 das Komponentenobjekt K2 auf. Bei Hardwareknoten erfolgt die Abstraktion analog mittels der Beziehungen dort ausgeführter Komponentenobjekte. Die Ergebnisse des Protokollkomparators werden mithilfe verschiedener Formeln auf Operationen, Komponentenobjekte und Hardwareknoten übertragen. Die Formeln für die Operationen berücksichtigen, dass sich abweichendes Verhalten einer Operation meist auch auf die aufgerufenen Operationen überträgt und dass normales Verhalten einer Operation auf die aufrufenden Operationen zurückfällt. Schließlich ergeben die Formeln jeweils den Abweichungswert, der beschreibt, ob die Operation, das Komponentenobjekt oder der Hardwareknoten Abweichungen zum vorherigen Ablaufverfolgungsprotokoll aufweist. Diese Ergebnisse werden dann auf eine Visualisierung des Aufrufgraphen übertragen, die neben den Operationen auch Komponentenobjekte und Hardwareknoten umfasst. In der Aufrufgraphvisualisierung sind die einzelnen Operationen von Komponentenobjekten umschlossen, die ihrerseits wiederum von Hardwareknoten umschlossen sind. Die Farbe der jeweiligen Operation, Komponentenobjekte und des jeweiligen Hardwareknotens korreliert dabei auf einer Farbskala von

Grün zu Rot mit dem errechneten Abweichungswert: je roter desto größer ist die Abweichung.

Der RanCorr-Ansatz basiert auf dem Vergleich zweier Ablaufverfolgungsprotokolle durch einen Protokollkomparator. Wenn die Protokolle den Durchlauf desselben Testlaufs widerspiegeln, ist der Testfallbezug gegeben (s. Anforderung A1). RanCorr berechnet Abweichungswerte für Komponentenoperationen (s. Anforderung A4), aber auch für Komponentenobjekte (s. Anforderung A2 und A6) und Hardwareknoten. Dabei ist eine Komponente dann fehlerhaft, wenn sich wenigstens ein Komponentenobjekt während der Ausführung fehlerhaft verhält. Für die Komponenten muss kein Quelltext vorliegen (s. Anforderung A7), um sie analysieren zu können. RanCorr ist nicht in der Lage direkt die Komposition oder die Komponentenverteilung zu bewerten (s. Anforderung A5). Jedoch kann angenommen werden, dass der Fehler im Bereich der Systementwickler liegt, wenn für keine Komponentenoperation bzw. kein Komponentenobjekt eine Abweichung gefunden wurde (s. Anforderung A3). Dies setzt voraus, dass das Referenzablaufverfolgungsprotokoll fehlerfrei ist. Bei dieser Art der Bewertung gilt zudem, dass nur Abweichungen festgestellt werden, nicht aber in welche Richtung diese gehen. Eine Abweichung kann also auch eine Verbesserung darstellen. RanCorr errechnet die Abweichungswerte und deren Darstellung automatisch, ist aber auf einen vorherigen Lauf eines Protokollkomparators angewiesen (s. Anforderung A8). Für den Protokollkomparator muss zusätzlich zum aktuellen Ablaufverfolgungsprotokoll ein passendes Referenzablaufverfolgungsprotokoll vorliegen. Wenn das Referenzablaufverfolgungsprotokoll nicht vorliegt, muss es aufwendig erstellt werden und kann nur schwer wiederverwendet werden (s. Anforderung A9). RanCorr errechnet Abweichungswerte z. B. für jede Komponentenoperation. Daraus lässt sich zwar eine Rangliste der größten Abweichungen ableiten, eine konkrete Entscheidung jedoch nicht (s. Anforderung A11). Die Ergebnisse von RanCorr werden in einer ausführlichen Aufrufgraphdarstellung visualisiert (s. Anforderung A10). Die Aufrufgraphdarstellung ist sehr ausführlich und stellt die drei Ebenen Komponentenoperation, Komponentenobjekte und Hardwareknoten dar. Die Darstellung stellt alle für die Performance-Blame-Analysis wichtigen Details dar und verbirgt diese wenn nötig durch Abstraktion auf eine höhere Ebene (s. Anforderung 12). Dabei werden die verschiedenen Ebenen berücksichtigt und bei Operationen werden auch zusätzliche Daten dargestellt. Z. B. werden die Aufrufhäufigkeit der Operation und die Verteilung der Ergebnisse des Protokollkomparators für diese Operation mit eingebracht (s. Anforderung A13). Es ergibt sich damit die folgende Bewertung:

Anforderung	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
„RanCorr“ [48]	+	o/+	o	o/+	-	+	+	+	-	+	o	+	+

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Der Ansatz von Jiang et al. [38] schlägt vor, die Ablaufverfolgungsprotokolle aus zwei Testläufen miteinander zu vergleichen. Das aktuelle Ablaufverfolgungsprotokoll und das Referenzablaufverfolgungsprotokoll bestehen dabei aus je einer Log-Datei. Jiang et al. gehen davon aus, dass die Log-Dateien Beobachtungen von beliebigen Ereignissen mit einem Zeitstempel dokumentieren. Performance-Metriken müssen aus einem solchen Ablaufverfolgungsprotokoll erst noch abgeleitet werden. Antwortzeiten werden bei Jiang et al. aus der Dauer zwischen zwei aufeinanderfolgenden Ereignissen berechnet. Daher muss eine Log-Datei für eine Performance-Blame-Analysis auch Aufruf und Verlassen der Komponentenoperationen dokumentieren. Jiang et al. erstellen aus diesen Log-Dateien Anfragepfade. Dazu wird zunächst für jeden Testlauf analysiert, welche verschiedenen Beobachtungen in den Zeilen der Log-Dateien identifiziert werden können und auf welche Weise diese miteinander in Verbindung gebracht werden können. Dies erfolgt über dynamisch generierte Informationen in den Log-Zeilen, wie zum Beispiel eine User-ID, eine Thread-ID oder einen Zeitstempel. Die verschiedenen gebräuchlichen Beobachtungstypen in der Log-Datei werden zwar automatisch extrahiert, allerdings muss der Systemarchitekt manuell die Zusammenhänge zwischen den identifizierten Ereignistypen herstellen. Dann werden die Log-Ereignisse mithilfe der Zusammenhänge zu Anfragepfaden zusammengestellt. Für jeden dieser Anfragepfade und für jedes Paar von im Pfad aufeinanderfolgenden Beobachtungen wird eine Antwortzeitdatenreihe erstellt, die angibt, wie lange jede Ausführung des Pfads bzw. Pfadabschnitts gedauert hat. Diese Antwortzeitdatenreihen werden nun weiter analysiert. Dabei wird jeweils die Datenreihe aus dem aktuellen Testlauf mit der eines früheren Testlaufs verglichen. Dafür verwenden Jiang et al. einen statischen Test (einen Zweistichproben-t-Test), um Abweichungen zu erkennen. Falls Abweichungen vorliegen, wenden sie einen abgewandelten statistischen Test (einen t-Test, der Konfidenzintervalle ausgibt) an, der feststellt, ob die Datenreihen aus dem aktuellen Test höhere oder niedrigere Werte beinhaltet als die Vergleichsdatenreihe. Auf Basis dieser Angaben generieren Jiang et al. einen Report, der zusammenfasst, welche Pfade die größten Abweichungen zwischen den beiden Testläufen aufweisen, wie die Werteverteilung der jeweiligen Datenreihen im Vergleich aussieht und wie die Abweichung der einzelnen Schritte des Pfades ist.

Beim Ansatz von Jiang et al. [38] ist der Bezug zu einem Testfall explizit gegeben (s. Anforderung A1). Der Ansatz operiert auf beliebigen Log-Dateien und daher ist eine Zuordnung zu Komponenten nur implizit gegeben (s. Anforderung A6). Da dieser Ansatz Antwortzeiten nur für ganze Anfragepfade und für im Pfad nacheinander auftretende Beobachtungen von Ereignissen ermittelt, sollten möglichst nur Aufruf und Verlassen der Komponentenoperationen als Ereignisse für die Performance-Blame-Analysis berücksichtigt werden. Trotzdem kann der Ansatz nicht für alle Komponentenaufrufe eine Antwortzeit ermitteln. Dies gilt für interne Komponentenaufrufe, die ihrerseits noch wenigstens eine Komponentenoperation aufrufen. Bei diesem Ansatz wird aus den Anfragepfaden eben kein Aufrufbaum ermittelt, da dieser Ansatz alle Arten von Log-Ereignissen zur Analyse zulässt. Da der Ansatz nicht die Antwortzeit aller Komponentenoperationen einbezieht, kann er die Performance von Operationen (s. Anforderung A4), Komponenten (s. Anforderung A2) und des gesamten Systems (s. Anforderung A3 und A5) nur unzureichend bewerten. Da hier lediglich Log-Dateien untersucht werden, braucht der Quelltext einer Komponente nicht vorzuliegen (s. Anforderung A7). Der Ansatz von Jiang et al. automatisiert die Analyse und kann auch das Ergebnisdokument (s. Anforderung A10) automatisiert produzieren (s. Anforderung A8). Der Ansatz benötigt jedoch die Eingabe, wie die verschiedenen Log-Ereignisse zusammenhängen. Diese Eingabe lässt sich schnell erstellen und kann bei übernommenen Komponenten wiederverwendet werden. Zusätzlich muss für den Ansatz ein passendes Referenzablaufverfolgungsprotokoll vorliegen, das ggf. erst aufwendig erstellt werden muss und das sich nur schwer wiederverwenden lässt (s. Anforderung A9). Das Ergebnisdokument des Ansatzes stellt dabei zunächst in einer Tabelle die Anfragepfade dar, die die höchste Abweichung haben. Durch eine Farbcodierung ist dabei angegeben, ob es sich um eine Verbesserung oder Verschlechterung handelt (s. Anforderung A11). Eine Analyseansicht zu dem Pfad zeigt dabei einen Vergleich der Datenreihen aus beiden Testläufen mithilfe von Diagrammen an (s. Anforderung A13). In einer Detaildarstellung können dann die einzelnen Pfadschritte untersucht werden und für jeden Schritt lässt sich ebenfalls eine Analyseansicht anzeigen. Diese abgestufte Auflistung von Pfaden und Pfadschritten mit den jeweiligen Möglichkeiten zur Detaillierung der Ergebnisse ist sehr übersichtlich (s. Anforderung A12). Der Ansatz von Jiang et al. erfüllt die Anforderungen wie folgt:

Anforderung	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
Jiang et al. [38]	+	-	-	-	-	-	+	+	-	+	o	+	+

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Der Ansatz von Sambasivan et al. [63] analysiert die Abweichungen zweier Ablaufverfolgungsprotokolle im Hinblick auf Funktionalität und Antwortzeit. Insbesondere kann die Analyse geänderte Reihenfolgen und Häufigkeiten von Anfragen sowie Antwortzeitabweichungen erkennen. Letzteres ist für die Performance-Blame-Analysis von Interesse. Für die initiale Erfassung und Analyse der Ablaufverfolgungsprotokolle setzt der Ansatz auf bestehende Arbeiten (z. B. Stardust [72] oder ggf. Magpie [1]), um ein Ablaufverfolgungsprotokoll zu ermitteln. Der Ansatz ermittelt zunächst Anfragepfade im Ablaufverfolgungsprotokoll und fasst dabei Pfade mit einer identischen Beobachtungsabfolge zusammen. Für jede Anfragepfadgruppe wird ein Anfrageparallelitätsgraph (im Paper „request flow graph“ genannt) gebildet, der auch explizit Parallelität abbildet. Dabei enthält ein Anfrageparallelitätsgraph je ein Ereignis für den Beginn und das Beenden der Bearbeitung einer Operation, ohne den Zusammenhang explizit zu machen. Für jedes benachbarte Ereignispaar im Anfrageparallelitätsgraphen bildet der Ansatz eine Antwortzeitdatenreihe. Der Ansatz untersucht Antwortzeitabweichungen, wenn der jeweilige Anfrageparallelitätsgraph im alten und im aktuellen Ablaufverfolgungsprotokoll eine identische Beobachtungsabfolge aufweist. Dann wird mithilfe eines statistischen Tests, des Kolmogoroff-Smirnow-Tests (KS-Test) [66], festgestellt, ob die Antwortzeitdatenreihen der Wurzeln der identischen Anfrageparallelitätsgraphen aus beiden Ablaufverfolgungsprotokollen voneinander abweichen. Falls eine Abweichung festgestellt wurde, werden zusätzlich die einzelnen Ausführungsschritte des kritischen Pfads eines Anfrageparallelitätsgraphen untersucht. Der kritische Pfad wird aus der dem Anfrageparallelitätsgraphen zugrunde liegenden Anfragepfadgruppe gewählt. Der kritische Pfad eines Anfrageparallelitätsgraphen ist der Pfad mit der höchsten Summe der Antwortzeiten der jeweiligen Einzelschritte auf dem Pfad. Für jeden dieser Ausführungsschritte wird wiederum ein statistischer Test durchgeführt, um festzustellen, ob für diesen Schritt im Anfrageparallelitätsgraphen eine Abweichung zu verzeichnen ist. Dann wird eine Rangliste der Anfrageparallelitätsgraphen mit den insgesamt größten Abweichungen zusammengestellt. Für Antwortzeitabweichungen richtet sich die Ranglistenwertung nach der Anzahl der gemessenen Ausführungen im aktuellen Ablaufverfolgungsprotokoll und nach der Differenz der mittleren Antwortzeiten zwischen beiden Ablaufverfolgungsprotokollen. Das Ergebnisdokument

umfasst eine Rangliste der Anfrageparallelitätsgraphen mit den größten Abweichungen. Dabei werden Antwortzeitabweichungen und funktionale Abweichungen berücksichtigt. Für jede Abweichung lässt sich der originale Anfrageparallelitätsgraph mit dem abweichenden Anfrageparallelitätsgraphen grafisch gegenüberstellen. Beim abweichenden Anfrageparallelitätsgraphen sind Antwortzeitabweichungen farblich gekennzeichnet. Alternativ kann ein Anfrageparallelitätsgraph mittels der sogenannten Gleisplanvisualisierung (engl. „train-schedule visualization“) dargestellt werden, um die Kommunikation zwischen Knoten und die Zeitverbräuche besser darzustellen (s. auch Abschnitt 3.3.4).

Der Ansatz von Sambasivan et al. [63] bezieht sich auf einen Testfall, vorausgesetzt beide Ablaufverfolgungsprotokolle stammen von der Ausführung desselben Testfalls (s. Anforderung A1). Mit dem Ansatz lassen sich Änderungen von Anfrageparallelitätsgraphen bezüglich der Antwortzeit von benachbarten Ereignispaaren und der Funktionalität finden. Da die benachbarten Ereignispaare nicht alle Komponentenoperationen erfassen, kann dieser Ansatz nur einen Teil aller Komponentenoperationen bewerten (s. Anforderung A4). Hier werden generell Abweichungen identifiziert, das Ranglistenmaß umfasst aber implizit die Art der Abweichung. Der Systemarchitekt kann sich so meist sicher sein, dass oben in der Rangliste nur Verschlechterungen zu finden sind. Es muss auch hier angemerkt werden, dass sich das Verfahren nur eignet, wenn das Referenzablaufverfolgungsprotokoll fehlerfrei ist. Da nicht alle Komponentenoperationen bewertet werden, ist auch eine Bewertung der Komponenten (s. Anforderung A2) oder gar des Gesamtsystems schwierig (s. Anforderungen A3 und A5). Dabei fehlt dem Ansatz lediglich die explizite Berücksichtigung der Operationen. Da der Ansatz auch funktionale Unterschiede aufspürt, also Anfrageparallelitätsgraphen, die nun anders abgearbeitet werden, könnte mit wenig zusätzlichem Aufwand ermittelt werden, wie sich die Komponentenverwendung verändert hat oder wie sich eine geänderte Komposition oder Komponentenverteilung auswirkt. Die Komponenten und Komponentenoperationen müssen nicht im Quelltext vorliegen, es müssen lediglich Beobachtungen auf diesem Abstraktionsniveau möglich sein (s. Anforderung A7). Einen eigenen Komponentenbegriff kennt dieser Ansatz jedoch nicht (s. Anforderung A6). Bei diesem Ansatz ist sowohl die Analyse als auch die Generierung der Präsentation automatisiert (s. Anforderung A8). Als Eingabe erfordert dieser Ansatz ein Referenzablaufverfolgungsprotokoll, das ggf. neu erstellt werden muss und nur schwierig wiederverwendet werden kann (s. Anforderung A9). Als Ergebnis erhält der Entwickler eine Rangliste mit Abweichungen, die er einzeln detailliert darstellen lassen kann (s. Anforderung A10). Eine Bewertung der einzelnen Abweichungen im Sinne der

untersuchten Fehlerwirkung ist implizit im Ranglistenmaß verankert. Das Ranglistenmaß basiert auf der Differenz zwischen der mittleren Antwortzeit aus dem Referenzprotokoll und dem aktuellen Protokoll, so dass der Pfad meist nur bei schlechterem Verhalten einen Werte größer Null erhält (s. Anforderung A11). Die einzelnen Anfrageparallelitätsgraphen können entweder mithilfe von Knoten und Kanten oder als Gleisplanvisualisierung dargestellt werden. Dieser Ansatz bietet eine Übersicht durch die Rangliste und kann jeden Ranglisteneintrag durch eine sehr übersichtliche Vergleichsdarstellung zweier Anfrageparallelitätsgraphen detaillieren (s. Anforderung A12). Die Diagramme enthalten aber neben den Antwortzeiten keine zusätzlichen entscheidungsrelevanten Performance-Werte (s. Anforderung A13). Zusammenfassend ergibt sich aus dieser Begründung die folgende Bewertung:

Anforderung	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
Sambasivan et al. [63]	+	-	-	-	-	-	+	+	-	+	o	+/o	-

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

3.3.3 Analyse durch Vergleich eines Ablaufverfolgungsprotokolls mit einer Spezifikation

Im Folgenden werden Ansätze vorgestellt, die ein Ablaufverfolgungsprotokoll mit einer Spezifikation vergleichen. Die Spezifikation kann dabei vom Systemarchitekten oder von den Komponentenentwicklern stammen. Nach der Beschreibung des jeweiligen Ansatzes wird geprüft, wie gut der jeweilige Ansatz die in Abschnitt 3.2 erarbeiteten Anforderungen umsetzt.

Der Ansatz „Pip“, der von Reynolds et al. [59] vorgestellt wurde, vergleicht ein Ablaufverfolgungsprotokoll bzw. die Anfragepfade aus dem Protokoll mit einer Spezifikation. Bei Pip bestehen die Anfragepfade aus drei Typen an Beobachtungen. Zunächst wird die Operationsbearbeitung (bei Pip allgemein als Aufgaben (engl. „tasks“) bezeichnet) mit dem Beginn und Ende ihrer Verarbeitung beobachtet. Davon losgelöst werden Nachrichten, die zum Beispiel die Operationsbearbeitung auslösen bzw. dessen Ergebnis zum Aufrufer transportieren, als Sende- und Empfangsereignis beobachtet. Dabei müssen zusammengehörige Sende- und Empfangsereignisse über eine ID gekoppelt werden können. Des Weiteren werden Log-Ausgaben als isolierte Ereignisse beobachtet. Eine Anfrage, wie sie in Abschnitt 2.1 beschrieben ist, wird im Pip-Ablaufverfolgungsprotokoll als empfangene Nachricht, Beginn und Ende der Operationsverarbeitung und gesendete Nachricht aufgezeichnet.

Die Autoren haben eine eigene Spezifikationssprache entworfen, mit der sich Erwartungen an die beobachteten Pfade spezifizieren lassen. Dabei soll sich die Sprache eignen, um Erwartungen sowohl an aufrufbasierte prozedurale bzw. objektorientierte Systeme als auch an ereignisbasierte Systeme zu formulieren. Der Ansatz sieht zunächst vor jeden Pfad durch eine Menge von in der Sprache spezifizierten Akzeptoren zu prüfen. Neben Akzeptoren kann es auch Nicht-Akzeptoren geben, die Pfade verwerfen. Pfade gelten als strukturell valide, wenn sie von wenigstens einem Akzeptor und keinem Nicht-Akzeptor erkannt werden. Dann werden Performance-Erwartungen an die validen Pfade geprüft. (Nicht-)Akzeptoren eignen sich dazu, die Reihenfolge von Aufgaben, das Senden und Empfangen von Nachrichten sowie Log-Ausgaben im Rahmen der Ausführung eines Threads zu spezifizieren. Dabei können Threads auf einem oder mehreren Hosts eines Systems ein- oder mehrfach laufen. Threads dienen auch als Container zwischen denen Nachrichten ausgetauscht werden. Bei den Nachrichten wird in der Spezifikationssprache lediglich angegeben, an welchen Thread oder Host sie gehen bzw. von welchem Thread oder Host sie kommen. Zusätzlich können Annahmen über die Performance eines so spezifizierten Pfads gemacht werden. Die Auswertung von Performance-Eigenschaften ist dabei an die Performance-Metriken sowie statistische (u. a. Minimum, Maximum, Mittelwert und Standardabweichung) und mathematische Operationen auf diesen Metriken gebunden. Das Ergebnis dieser Analyse sind Pfade die entweder akzeptiert oder aufgrund von strukturellen bzw. Performance-Eigenschaften abgelehnt wurden. Ein Pfad kann in Pip in einer Baumdarstellung visualisiert werden, die die Struktur des Pfads übersichtlich darstellt. Alternativ kann auch eine Gleisplanvisualisierung eines Pfads angezeigt werden, bei der die Kommunikation der Threads und parallele Abläufe besser dargestellt werden. Zusätzlich können verschiedene Ansichten von Performance-Datenreihen, wie zum Beispiel (kumulative) Verteilungsfunktionen, zu einem Pfad bzw. zu einer Aufgabe angezeigt werden.

Pip baut auf den Beobachtungen einer Testausführung auf (s. Anforderung A1). Bei Pip kann all das bewertet werden, was als Pfad-Erwartung spezifiziert ist. Spezifizieren lässt sich die Performance von Komponentenoperationen (s. Anforderung A4). Des Weiteren lassen sich auch Erwartungen an die Interaktion von Komponenten, die auf verschiedenen Hardware-Knoten verteilt sind, angeben (s. Anforderung A5). Allerdings muss der Systemarchitekt selbst erkennen, welche Pfadteile zu welcher Komponente gehören, da Pip keine Komponenten kennt (s. Anforderung A6). Eine Bewertung von Komponenten als solches kann aber über die Komponentenoperationen erfolgen (s. Anforderung A2). Falls die Performance der Komponentenoperationen umfassend

und korrekt spezifiziert wurde und falls auf dieser Grundlage alle Komponentenoperationen von Pip als fehlerfrei bewertet wurden, so ist der Fehler im Bereich der Systementwickler zu suchen (s. Anforderung A3). Pip kann generell mit Blackbox-Komponenten umgehen, ist aber darauf angewiesen, dass die Ausführungsumgebung oder aber die eingesetzten Komponenten mit einer Pip-spezifischen Log-Bibliothek die für Pip benötigten Beobachtungen protokolliert (s. Anforderung A7). Die Konstruktion der Pfade aus den beobachteten Pip-Ereignissen und die Auswertung der Erwartungsspezifikation aufgrund der Pfade sind automatisiert (s. Anforderung A8). Dabei muss die Erwartungsspezifikation zunächst erstellt werden, um den Ansatz zu nutzen. Die Spezifikation ist aufwendig zu erstellen, da sie den Ablauf auf Ebene der Komponentenoperationen vollständig angeben muss und zudem die jeweils erwartete Performance beinhalten muss. Dabei müssen ähnliche Abläufe alle separat spezifiziert werden, da die Spezifikationssprache keinerlei Wiederverwendung, z. B. durch Variablen oder Parameter, unterstützt. Die Pfad-Erwartungen, insbesondere Performance-Erwartungen, können nur schwer auf andere Kontexte übertragen werden, da sie nicht vom Kontext entkoppelt spezifiziert sind (s. Anforderung A9). Dabei entscheidet Pip klar, ob die spezifizierten Erwartungen von den vorliegenden Pfaden erfüllt wurden (s. Anforderung A11). Pip verfügt über eine GUI, die die Ergebnisse aufbereitet und präsentiert (s. Anforderung A10). Dabei werden Pfade durch das System einzeln als Baum oder Gleisplanvisualisierung visualisiert und dazu zeigen Textfenster Einzelheiten zu den gemessenen Performance-Metriken an. Zusätzlich lassen sich für einzelne Aufgaben oder Pfade weitere Diagramme anzeigen, wie zum Beispiel die Verteilungsfunktion der Antwortzeit oder das Verhalten über Zeit (s. Anforderung A13). Die Darstellung der Ergebnisse scheint vollständig und durchdacht, könnte aber besser organisiert sein, da beispielsweise keine Übersicht über die Pfade gegeben wird und wichtige Zusatzinformationen lediglich textuell je Ereignis im Pfad dargestellt werden (s. Anforderung A12). Es ergibt sich damit die folgende Bewertung:

Anforderung	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
Reynolds et al. [59]	+	o/+	+	+	o	-	+	+	o/-	+	+	o	+

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Der Ansatz von Groenda [31, 30] testet, ob eine Komponentenimplementierung konform zu ihrem Performance-Kontrakt ist. Der Performance-Kontrakt soll dabei als Palladio-Komponentenmodell (PCM) vorliegen. Das Herzstück des Kontraktes bildet dabei die abstrahierte Beschreibung des Kontrollflusses je Komponentenoperation in sogenannten Service-Effect-Specifications

(SEFFs). Der Detailgrad der Analyse wird vom Detailgrad dieser Kontrollflussbeschreibung bestimmt. Groenda ordnet die Kontrollflusselemente eines SEFF den Elementen des generalisierten abstrakten Syntaxbaums der Implementierung zu. Damit legt er fest, welche Modellelemente zu welchen Quelltextblöcken korrelieren. Auf dieser Grundlage generiert Groenda Testfälle, bei denen die Antwortzeit, oder besser gesagt die Ausführungsdauer, für jeden Quelltextblock beobachtet wird. Für jeden dieser Blöcke erhält Groenda somit eine Performance-Datenreihe. Dann leitet er Performance-Datenreihen für die korrespondierenden Elemente aus der Simulation der PCM-Modelle ab. Die Paare aus Testdatenreihe und Simulationsdatenreihe für jeden Block werden dann mithilfe eines statistischen Tests auf Gleichheit untersucht. Groenda stellt die Ergebnisse dieses statistischen Tests, also bestanden / nicht bestanden, in den SEFF-Diagrammen dar, so dass der Systemarchitekt leicht erkennen kann, welche Teile des Performance-Kontrakts nicht zur Implementierung passen. Leider geht Groenda nicht näher auf die Anwendung des statischen Tests und die Aggregation der Werte aus den verschiedenen Tests ein.

Der Ansatz von Groenda [31, 30] bezieht sich nicht auf einen bestimmten Testfall (s. Anforderung A1), sondern versucht eine allgemeingültige Aussage über eine bestimmte Kombination von Komponentenimplementierung und Performance-Kontrakt zu treffen. Er bewertet die Performance von Komponentenoperationen bzw. von Blöcken innerhalb der Komponentenoperation (s. Anforderung A4) und beurteilt so, ob der Kontrakt zur Komponente passt (s. Anforderung A2). Der Ansatz eignet sich nicht dazu eine Aussage über Systemverhalten zu treffen, da der Kontrakt im Bezug auf die Komponente isoliert bewertet wird und nicht im Kontext betrachtet wird (s. Anforderung A3). Das Systemverhalten kann so nicht bewertet werden (s. Anforderung A5). Mit PCM als Grundlage arbeitet Groendas Ansatz explizit mit einem Komponentenbegriff (s. Anforderung A6). Da Groenda einen Abgleich eines SEFF mit dem Quelltext durchführt, ist der Ansatz auf den Quelltext angewiesen (s. Anforderung A7). Das Generieren der Testfälle, die Testfallausführung, die Beobachtung der Antwortzeiten, deren Auswertung und die Ergebnisdarstellung innerhalb der SEFFs sind automatisiert (s. Anforderung A8). Als Eingabe dienen die Komponenten mit dem zugehörigen PCM-Kontrakt, beides soll vom Komponentenentwickler zugeliefert werden. Zusätzlich muss der Systemarchitekt noch Vorgaben bezüglich der Testfallgenerierung und Auswertung, z. B. die Testfallgenerierungsstrategie und die Fehlertoleranz des statistischen Tests, machen (s. Anforderung A9). Groendas Ansatz vergleicht die Ergebnisse der Tests mithilfe von statistischen Tests mit den Palladio-Simulationsergebnissen und entscheidet, ob der Kontrakt zur Komponen-

te passt, bezieht sich dabei aber nicht auf den zu untersuchenden Testfall (s. Anforderung A11). Diesen Entscheidungen werden dann in den SEFF-Diagrammen dargestellt, so dass man je SEFF-Element sehen kann, wie die Entscheidung lautet (s. Anforderung A10). Die Darstellung innerhalb des Kontrakts ist übersichtlich und zeigt direkt, welche Teile des Kontrakts nicht zur Implementierung passen (s. Anforderung A12). Allerdings fehlt die weitere Detaillierung darüber hinaus. Der Ansatz untermauert die Entscheidung nicht durch weitere Performance-Metriken (s. Anforderung A13). Insgesamt ergibt sich damit die folgende Bewertung:

Anforderung	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
Groenda [31, 30]	-	o	-	o	-	+	-	+	+	+	o	+/o	-

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Bei der Performance-Blame-Analysis geht es hauptsächlich um das Testen und die Analyse von Performance-Fehlern. Im komponentenbasierten Softwareengineering gibt es mehrere Ansätze zu sogenannten eingebauten Tests (engl. „built-in tests“), die Beydeda [9] sowie Gao et al. [26] zusammengefasst haben. Eingebaute Tests sind Testfälle oder Testfallgeneratoren, die von den Komponentenentwicklern entwickelt und dann mit den oder in den Komponenten ausgeliefert werden. Die eingebauten Tests sind technisch entweder innerhalb der Komponente oder zuschaltbar in Form eines Wrappers realisiert. Sie werden vom Softwarearchitekten im Zielkontext ausgeführt, bzw. generiert und ausgeführt. Dazu benötigt der Softwarearchitekt keinerlei Wissen über den inneren Aufbau der Komponente. Bei eingebauten Tests ist häufig der Quelltext der Tests erhältlich, damit der Softwarearchitekt sehen kann, was genau getestet wird und damit er Vertrauen zu den Tests und damit auch zur Komponente aufbauen kann. Eingebaute Tests enthalten vorgegebene Testkriterien, die automatisch in einem Durchlauf der eingebauten Tests überprüft werden, um die Testergebnisse (bestanden, nicht bestanden, und ungültig) zu liefern.

Bei eingebauten Tests besteht zwar ein Bezug zu Testfällen jedoch nur zu den eingebauten Tests, ein bestimmter gewünschter Testfall kann nicht untersucht werden (s. Anforderung A1). Es ist gar nicht möglich beliebige Sachverhalte zu überprüfen. Es können so keine Rückschlüsse auf das Verhalten von Komponenten, Komponentenoperationen, System oder bestimmten Systemaspekten im Bezug auf eine bestimmte Situation gezogen werden (s. Anforderung A2-A5). Dies liegt vornehmlich daran, dass die Tests vorgegeben sind und auch die Testkriterien festgelegt sind. Die festgelegten

Testkriterien gehen nicht auf den vorliegenden Kontext ein, so dass generell eine Bewertung der Performance schwierig erscheint. Eingebaute Tests beziehen sich immer auf eine bestimmte Komponente (s. Anforderung A6). Eingebaute Tests können problemlos für Blackbox-Komponenten durchgeführt werden, hier liegt eine Stärke des Verfahrens (s. Anforderung A7). Diese Tests laufen zudem automatisch ab (s. Anforderung A8) und erfordern keine weiteren Eingaben (s. Anforderung A9). Ein Ergebnisdokument umfasst lediglich die Testergebnisse (s. Anforderung A10 und A13), das zwar eine klare Entscheidung beinhaltet, aber nicht zur untersuchten Fehlerwirkung (s. Anforderung A11). Die Darstellung ist auf eine Übersicht der Testergebnisse beschränkt und weitere Detaillierung der Ergebnisse erfolgt nicht (s. Anforderung A12). Zusammenfassend ergibt sich damit die folgende Bewertung:

Anforderung	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
eingebaute Tests [9, 26]	o	-	-	-	-	+	+	+	+	+	o	o	-

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

3.3.4 Gängige Performance-Visualisierungen

In den Unterabschnitten 3.3.1 bis 3.3.3 wurden verschiedene Darstellungsarten eingeführt, die die verschiedenen Ansätze verwenden, um ihre Ergebnisse darzustellen. Dabei wurden die Darstellungen schon im Hinblick auf Übersichtlichkeit (s. Anforderung A12 aus Abschnitt 3.2) und ihren Inhalt bewertet. Inhaltlich wurde geprüft, ob hier eine konkrete Empfehlung dargestellt wird (s. Anforderung A11) und ob auch zusätzliche relevante Metriken dargestellt werden (s. Anforderung A13). Die Bewertung dieser Anforderungen in den Unterabschnitten 3.3.1 bis 3.3.3 lässt schon erkennen, dass es hier Darstellungsarten von unterschiedlicher Qualität gibt. Zudem werden in den Darstellungen verschiedene Techniken immer wieder angewandt. Daher werden im Folgenden die eingeführten Darstellungsarten zusammen mit weiteren verwandten Darstellungsarten noch einmal genauer unter die Lupe genommen. Zunächst werden die bisher erarbeiteten Anforderungen untersucht und daraus werden Visualisierungsanforderungen abgeleitet, die sich direkt auf die Darstellungsarten beziehen. So können die Darstellungsarten losgelöst von den Analyseverfahren verglichen und bewertet werden. Schließlich kann der Bereich der Visualisierungen auf dieser Grundlage auch bei der Ableitung verbleibender Anforderungen für den zu erarbeitenden Ansatz berücksichtigt werden.

Visualisierungsanforderungen

Zunächst betrachten wir die Anforderungen, die festlegen, welche Entitäten ein Performance-Blame-Analysis-Ansatz bewerten können soll. Besonders sollen Performance-Blame-Analysis-Ansätze Komponenten (s. Anforderung A2) und Komponentenoperationen (s. Anforderung A4) evaluieren. Somit sollten diese Elemente auch in einer Darstellung auftauchen. Damit ergeben sich die ersten zwei Visualisierungsanforderungen:

Anforderung V1:
Die Darstellung stellt Komponenten dar.

Anforderung V2:
Die Darstellung stellt Komponentenoperationen dar.

Die in Anforderung A3 geforderte Zuordnung des beobachteten Performance-Fehlers zu den Systementwicklern kann nur durch eine gute Entscheidungsgrundlage gewährleistet werden. Auf dieser Grundlage kann eine gute Darstellung der Komponenteninteraktion und der Komponentenfehler dem Systemarchitekten zusätzlich helfen. Die Anforderungen V1 und V2 fordern bereits, dass Komponenten und Komponentenoperationen dargestellt werden sollen. Die nötige Ausgestaltung der Visualisierung wird im Folgenden im Rahmen der weiteren Anforderung diskutiert.

Anforderung A5 fordert, dass die Komponentenverwendung, also z. B. die Komposition und Komponentenverteilung, untersucht werden kann. Während mit den Komponenten und Komponentenoperationen die Komposition schon recht vollständig überwacht wird, müssen für die Komponentenverteilung zusätzlich noch die Hardwareknoten und deren Ressourcen dargestellt werden. Es ergeben sich also die folgenden Visualisierungsanforderungen:

Anforderung V3:
Die Darstellung stellt Hardwareknoten dar.

Anforderung V4:
Die Darstellung stellt die Ressourcen der Hardwareknoten dar.

Die Anforderung A6 bezieht sich wieder auf Komponenten und Komponentenoperationen, die schon in Anforderungen V1 und V2 abgedeckt wurden. Die Anforderungen A7 und A9 sind für die Visualisierung irrelevant. Außerdem deckt Anforderung A8 die automatische Erstellung der Darstellungen bereits ab, so dass auch diese Anforderung hier ausgelassen werden

kann. Anforderung A10 fordert, dass die Ergebnisse des Performance-Blame-Analysis-Ansatzes in einem Dokument zusammengefasst werden. Wenn also eine Darstellung vorliegt, ist die Anforderung schon erfüllt. Demgegenüber fordert Anforderung A11, dass der Ansatz eine Entscheidung bezüglich der Performance-Blame-Analysis liefert und dass diese auch dargestellt wird. Diese Prüfung wird hier nochmals auf die einzelne Darstellungsart angewendet:

Anforderung V5:
Die Darstellung stellt eine klare Entscheidung bezüglich der Performance-Blame-Analysis dar.

Anforderung A12 fordert, dass die Ergebnisse des jeweiligen Ansatzes übersichtlich präsentiert werden. Dazu gehört, dass die relevanten Informationen, wie schon gefordert, im Diagramm enthalten sind. Diese Informationen sollen so dargestellt sein, dass das Diagramm im Hinblick auf die Performance-Blame-Analysis leicht zu interpretieren ist. Dazu sollte die Darstellung den Vergleich zwischen den dargestellten Entitäten durch eine Interpretationshilfe möglichst einfach gestalten. Idealerweise sollte die Interpretationshilfe im Einklang mit der Performance-Blame-Analysis sein. Wenn dies gewährleistet ist, sollen die Darstellungen zusätzlich so gestaltet sein, dass möglichst viele dieser Ergebnisse auf wenig Platz dargestellt werden. Um zugleich eine detailreiche, leicht verständliche und auch platzsparende Darstellung zu realisieren, soll es möglich sein, dynamisch Darstellungselemente ein- und auszublenden. Daraus ergeben sich die Anforderungen:

Anforderung V6:
Die Darstellung ist im Hinblick auf die Performance-Blame-Analysis leicht zu interpretieren. Das heißt, dass die Darstellung eine Interpretationshilfe für den Systemarchitekten bietet, die den Vergleich der dargestellten Entitäten möglichst einfach gestaltet. Diese Interpretationshilfe ist zudem idealerweise auch relevant für die Performance-Blame-Analysis.

Anforderung V7:
Die Darstellung ist platzsparend und präsentiert viele Ergebnisse auf einen Blick.

Anforderung V8:
Die Darstellung ist dynamisch und es können Darstellungselemente ein- und ausgeblendet werden.

Zuletzt besagt Anforderung A13, dass die Darstellung auch weiterführende Metriken beinhalten soll. An dieser Stelle soll daher bewertet werden, wie viele Informationstypen in einem Diagramm dargestellt werden können:

Anforderung V9:
Die Darstellung soll möglichst viele relevante Informationen darstellen können. Die unterschiedlichen Informationstypen werden gezählt.

Mit diesen neun Visualisierungsanforderungen werden nun im Folgenden die Visualisierungen der bisher vorgestellten verwandten Arbeiten näher untersucht. Bei der Anwendung der Visualisierungsanforderungen werden die Visualisierungen losgelöst vom Gesamtansatz evaluiert. Dabei ist natürlich die Datengrundlage auch wichtig, aber die Visualisierung soll die vorhandenen Daten eben auch so darstellen, dass sie für den Systemarchitekten leicht greifbar und nachvollziehbar sind. Zusätzlich werden auch exemplarisch einige Darstellungen aus dem Bereich der Performance-Profiler evaluiert.

Vorstellung und Bewertung der Visualisierungen

Zunächst sollen nun Darstellungen, wie sie im Rahmen der Performance-Profiler üblich sind untersucht werden. Performance-Profiler sind nach Kreft und Langer [43] Werkzeuge, die dazu dienen „eine Anwendung zu untersuchen und darin Schwachstellen zu bestimmen“. Dabei verfolgt man zumeist das Ziel bestimmte Performance-Schwachstellen wie „Performance-Bottlenecks“ oder „Memory-Leaks“ zu finden. Profiling wird während der Entwicklung und nicht während des Produktivbetriebs vorgenommen, weshalb hier ein höherer Aufwand bei der Beobachtung der Daten in Kauf genommen wird. Kreft und Langer haben drei Arten von Profilern identifiziert, die häufig in einem einzelnen Tool zusammengefasst sind. Zunächst gibt es „Time Profiler“, die die Zeit messen, die das beobachtete System in jeder Methode verbringt, so dass Methoden mit sehr hohem Verbrauch identifiziert werden können. Hier klingt schon an, dass Profiler meist nicht mit komponentenbasierten Systemen arbeiten, sondern üblicherweise mit prozeduralen oder objektorientierten Systemen. Die Profiler können bei komponentenbasierten Systemen angewendet werden, arbeiten aber auf einer anderen Abstraktionsebene, da alle Methoden- oder Prozeduraufrufe erfasst werden. Zweitens gibt es „Space Profiler“, die erfassen, welche Methode wie viel Speicher anfordert. Zuletzt gibt es sogenannte „Thread Profiler“, welche den Lebenszyklus von Threads beobachten. Dabei erfassen sie auch, wenn Threads aufgrund

einer Sperre warten, um so auf ineffiziente Zeitausnutzung und Verklemmungen hinzuweisen. Da sich diese Arbeit auf die Analyse von Antwortzeiten fokussiert, werden im Folgenden die Darstellungen der „Time Profiler“ beschrieben.

Der bekannte Open-Source-Profiler JFluid (auch als Netbeans Profiler bekannt) verwendet eine Tabellendarstellung, die auf dem sogenannten Aufrufkontextbaum (vgl. Abschnitt 2.1) beruht [19]. Die Visualisierung besteht im Wesentlichen aus einer Methodentabelle, in der der Aufrufkontextbaum dargestellt wird. Ein Beispiel dafür zeigt Abbildung 3.4. Jede Tabellenzeile steht für einen Methodenaufruf im Aufrufkontextbaum. Die von dieser Methode aufgerufenen Methoden werden eingerückt in den Tabellenzeilen darunter dargestellt. Jede Zeile beinhaltet zusätzlich zum Methodennamen auch die Aufrufhäufigkeit und die Summe der Antwortzeiten. Die speziellen Tabellenzeilen mit dem Pseudo-Methodennamen „Self time“ geben die Nettoantwortzeiten der Methode an. Der kommerzielle Profiler JProfiler [20] verwendet eine ähnliche Darstellung. JProfiler verwendet jedoch eine klassische Graphdarstellung mit Knoten und Kanten, um den Aufrufkontextbaum darzustellen (s. Abbildung 3.5). Dabei beinhaltet jeder Knoten dieselben Informationen wie eine Tabellenzeile beim JFluid-Profiler, also den Methodennamen, die Aufrufhäufigkeit, die Nettoantwortzeiten und die Summe der Antwortzeiten.

Diese Profiler-Darstellungen sind nicht auf komponentenbasierte Systeme spezialisiert, daher werden Komponenten (s. Anforderung V1) überhaupt nicht und Komponentenoperationen wie alle anderen Methoden dargestellt (s. Anforderung V1). In den Darstellungen finden sich auch keine Hardwareknoten und Ressourcen (s. Anforderungen V3 und V4). Die Darstellung enthält keine Entscheidung (s. Anforderung V5), aber die Einträge sind nach der relativen Antwortzeit verglichen mit dem Elternelement eingefärbt. Je näher die Antwortzeit des jeweiligen Elements der des Elternelements kommt, desto länger ist der rote Balken (JFluid) bzw. desto roter und dunkler ist die Einfärbung des Knotens (JProfiler). Das macht den Vergleich der Methoden im Bezug auf die Antwortzeit in der jeweiligen Darstellung recht leicht. Zudem kann der Nutzer der Spur der höchsten Antwortzeiten folgen und so Ursachen finden (s. Anforderung V6). Dies ist im Hinblick auf die Performance-Blame-Analysis, aber nicht zwingend eine nützliche Klassifizierung. Die Darstellung ist im Fall von JFluid einigermaßen platzsparend (s. Anforderung V7). Im Fall von JProfiler ist sie überhaupt nicht platzsparend. Dies relativiert sich jedoch durch die Möglichkeit die Bäume aus- und einzuklappen (s. Anforderung V8). Die Visualisierungen zeigen 4 Informationen, nämlich die Aufrufhierarchie,

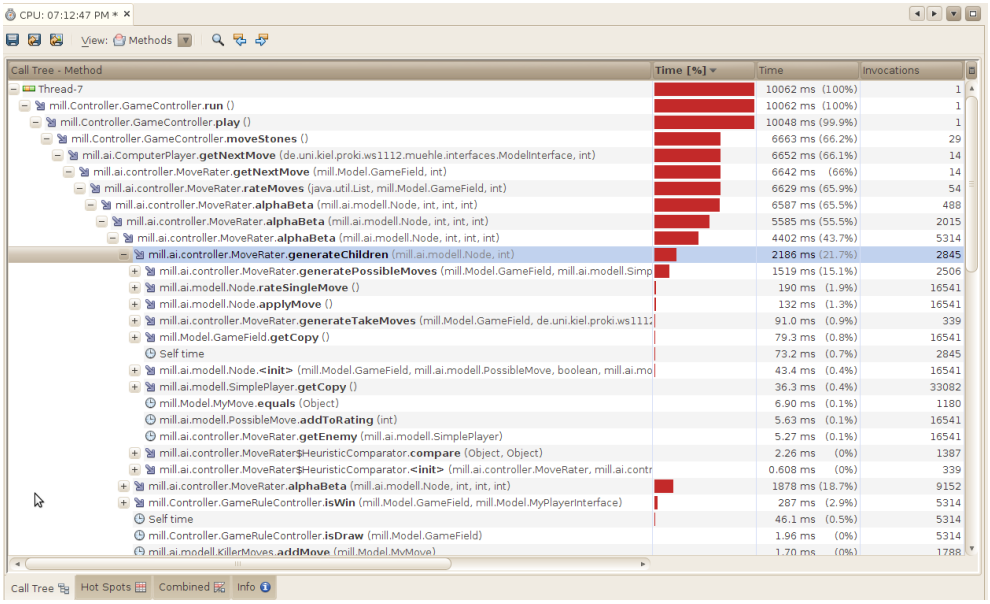

Abbildung 3.4: Beispiel für die Darstellung des Profilers JFluid (bekannter als Netbeans Profiler).[2]

die Nettoantwortzeiten, die Summe der Antwortzeiten und die Färbung nach dem Antwortzeitanteil. Zusammenfassend ergibt sich die Bewertung:

Anforderung	V1	V2	V3	V4	V5	V6	V7	V8	V9
Profiler [19, 20]	-	o	-	-	o	o	o	+	4

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Der Ansatz RanCorr [48] bietet eine erweiterte Darstellung von Aufrufgraphen (vgl. Abschnitt 2.1). Dabei wird, wie bei JProfiler, auf eine Graphdarstellung gesetzt. Ein Beispiel ist in Abbildung 3.6 zu sehen. Bei dieser Darstellung werden die Knoten für die Komponentenoperationen wiederum in Rechtecken zusammengefasst. Diese stehen für das jeweilige Komponentenobjekt. Die Komponentenrechtecke werden nochmals von Rechtecken zusammengefasst, die für den ausführenden Knoten stehen. Bei RanCorr tragen die Kanten die Aufrufhäufigkeit und in den Komponentenoperationsknoten wird

[2] Die Abbildung wurde ohne Änderungen von der Webseite http://software-talk.org/blog/de/2012/10/wann-und-wie-java-performance-steigern/ (abgerufen am 13.03.2014) übernommen. Sie wurde dort am 15.10.2012 von Tim Coen unter Lizenz CC BY-NC 3.0 (http://creativecommons.org/licenses/by-nc/3.0/) veröffentlicht.

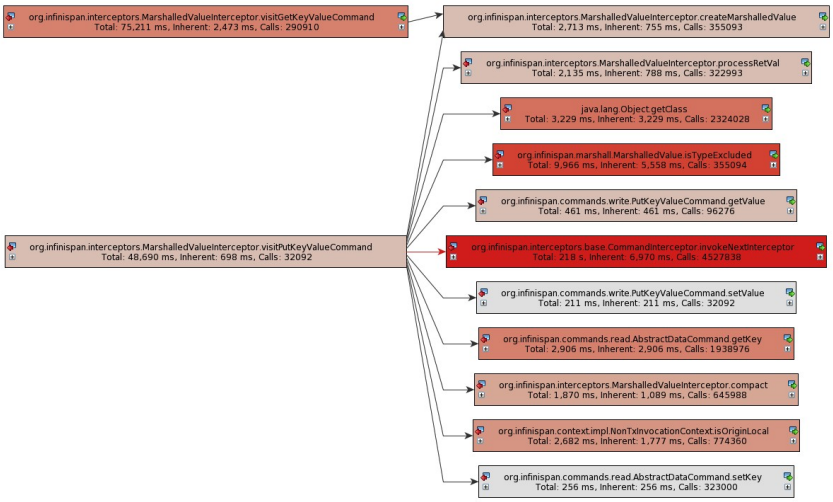

Abbildung 3.5: Beispiel für die Darstellung des Profilers JProfiler.[3]

das Ergebnis der Abweichungsformel (vgl. Unterabschnitt 3.3.2), die Fehlerwahrscheinlichkeit und ein Verteilungsdiagramm der Abweichungswerte angeführt. Generell sind die Rechtecke analog zur Fehlerwahrscheinlichkeit mittels einer Farbskala von Grün (0%) bis Rot (100%) eingefärbt.

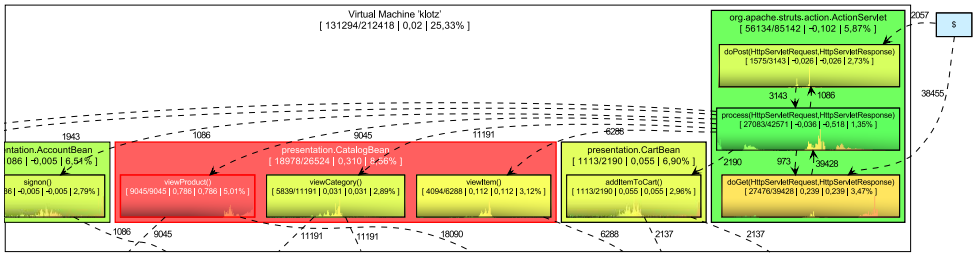

Abbildung 3.6: Beispiel für die Darstellung des RanCorr-Ansatzes von Marwede et al. [48].

Die Visualisierung von RanCorr stellt Komponentenoperationen, Komponenten und Hardwareknoten dar (s. Anforderung V1, V2 und V3). Er bildet jedoch keine Ressourcen ab (s. Anforderung V4). In der Darstellung ist keine direkte Entscheidung gegeben, wohl aber eine visuelle Repräsentation einer Fehlerwahrscheinlichkeit (s. Anforderung V5). Die farbliche Kennzeichnung

[3] Abbildung von der Webseite https://issues.jboss.org/browse/ISPN-1267 (abgerufen am 13.03.2014)

der Elemente ermöglicht einen einfachen Vergleich der Elemente auf allen drei Abstraktionsebenen. Dies gibt dem Systemarchitekten eine gute Interpretationshilfe, die sich dank der dargestellten Komponenten und Komponentenoperationen vergleichsweise leicht auf die Performance-Blame-Analysis übertragen lässt (s. Anforderung V6). Wenn alle Details aus allen Abstraktionsebenen eingeblendet sind, benötigt diese Darstellung allerdings auch sehr viel Platz (s. Anforderung V7). Dies wird jedoch teils dadurch wieder ausgeglichen, dass die drei Abstraktionsstufen dazu genutzt werden können, um Informationen ein- und auszublenden (s. Anforderung V8). Die Darstellung zeigt Aufrufhäufigkeit, Abweichungswert, Fehlerwahrscheinlichkeit inklusive Einfärbung für jede Abstraktionsstufe und Verteilungsdiagramm der Abweichungswerte je Komponentenoperation. Damit ergeben sich 7 Informationen (s. Anforderung V9). Insgesamt ergibt sich die folgende Bewertung:

Anforderung	V1	V2	V3	V4	V5	V6	V7	V8	V9
RanCorr [48]	+	+	+	-	o	o/+	o	+	7

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Der Ansatz Magpie [1], der Ansatz von Sambasivan et al. [63] sowie der Ansatz von Reynolds et al. [59] verwenden eine sogenannte Gleisplanvisualisierung.[4] Bei dieser Visualisierung wird die Interaktion zwischen verschiedenen Elementen dargestellt. Dabei erhält jedes Element ein Gleis (auch Lebenslinie genannt). Die Gleise beginnen auf unterschiedlichen Punkten der Y-Achse und verlaufen parallel zur X-Achse. Auf der X-Achse wird der Zeitverlauf aufgetragen. Es gibt nun Markierungen auf den Gleisen und Verbindungen zwischen Ihnen. An der Länge der Markierungen und Verbindungen über der X-Achse lässt sich ablesen, wie lange eine Verarbeitung durch das Element dauert bzw. wie viel Zeit eine Kommunikation zwischen den Elementen benötigt. Diese Darstellung eignet sich auch für die Darstellung von paralleler Verarbeitung. Dabei überlappen sich die Markierungen verschiedener Gleise über einer Spanne der X-Achse.

Im Einzelnen werden die Gleisplanvisualisierungen von jedem Ansatz sehr unterschiedlich genutzt. Bei Magpie [1] werden die Ressourcen und Entitäten aus dem Monitoring wiedergegeben (s. Abbildung 3.7), z. B. Threads, CPUs und Webserver-Sessions. Je nach Realisierung des Monitorings können unterschiedliche Ressourcen und Entitäten in der Darstellung auftauchen. Dabei stellen die Markierungen auf den Gleisen dar, wie lange diese Ressource bzw.

[4] Die Gleisplanvisualisierung kann auch als Spielart des Gantt-Diagramms [37] gesehen werden.

Entität aktiv war. Die Markierungen werden vertikal durch die Darstellung von Ereignissen verbunden. Bei Sambasivan et al. [63] werden Hardwareknoten und Nachrichten zwischen ihnen dargestellt (s. Abbildung 3.8). Bei Reynolds et al. [59] werden einzelne Aufgaben bzw. Operationsbearbeitungen als Markierungen auf Gleisen dargestellt und die Farbe der Markierungen steht für den Hardwareknoten auf dem die Aufgabe ausgeführt wird (s. Abbildung 3.9). Die Verbindungen zwischen den Aufgaben sind die Nachrichten, die zwischen Ihnen ausgetauscht werden.

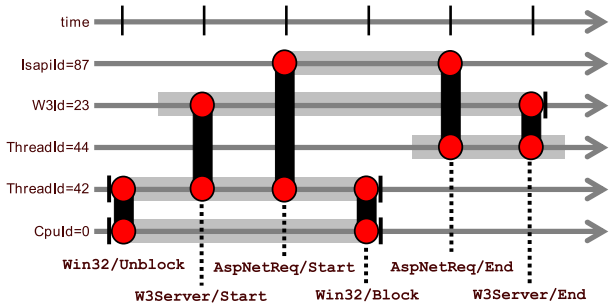

Abbildung 3.7: Beispiel für die Gleisplanvisualisierung des Magpie-Ansatzes von Barham et al. [1].

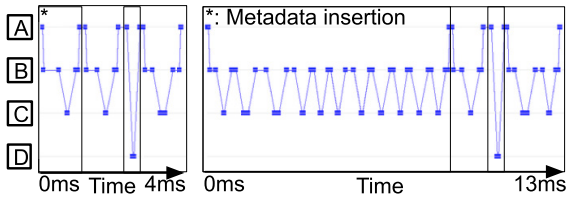

Abbildung 3.8: Beispiel für die Gleisplanvisualisierung des Ansatzes von Sambasivan et al. [63].

Damit ergibt sich eine ähnliche aber im Detail unterschiedliche Bewertung der Gleisplanvisualisierung in den verschiedenen Ansätzen. Keine der Visualisierungen stellt Komponenten dar (s. Anforderung V1). Nur bei Magpie wäre dies möglich, falls die Komponenten im Rahmen des Monitorings identifiziert werden. Die Komponentenoperationen werden lediglich bei Pip von Reynolds et al. im Rahmen der Aufgaben erkannt (s. Anforderung V2). Bei Magpie ist dies wiederum vom Monitoring abhängig und beim Ansatz von Sambasivan et al. ist dies nicht der Fall. Dagegen werden die Hardwareknoten bei allen Ansätzen dargestellt (s. Anforderung V3). Jedoch ist es bei

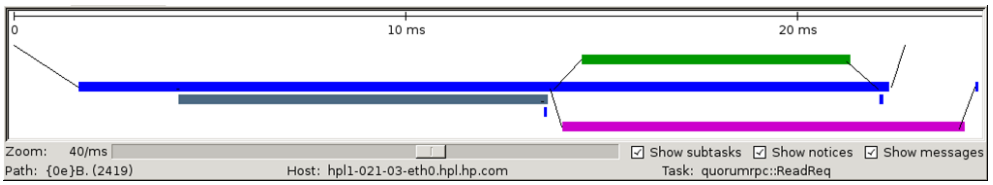

Abbildung 3.9: Beispiel für die Gleisplanvisualisierung des Ansatzes Pip von Reynolds et al. [59].[5]

Magpie erforderlich, das Monitoring so zu gestalten, dass die Hardwareknoten als Container der Ressourcen mit in der Darstellung auftauchen. Die Ressourcen werden bei Magpie, im Unterschied zu den anderen Darstellungen, standardmäßig dargestellt (s. Anforderung V4). Dies wird durch das in Magpie integrierte Monitoring des Betriebssystems sichergestellt. Die vorgestellten Gleisplanvisualisierungen stellen keine Entscheidung bezüglich der Performance-Blame-Analysis dar (s. Anforderung V5). Gleisplanvisualisierungen machen den Vergleich der dargestellten Entitäten nicht besonders einfach. Die für den Vergleich relevanten Zeiträume, also die Ausführung der verschiedenen Komponentenoperationen, werden über den gesamten Verlauf der Zeitachse auf verschiedenen Gleisen verstreut dargestellt werden. Zudem muss der Zusammenhang mit der Performance-Blame-Analysis über die Berechnungsdauern und die Kommunikation zwischen den Entitäten erst durch den Systemarchitekten ergründet werden. Somit ist die Darstellung bezüglich der Performance-Blame-Analysis nicht leicht zu interpretieren (s. Anforderung V6). Die Darstellung ist meist nicht sehr platzsparend, da sich die X-Achse, also die Zeitachse, meist viel Platz in Anspruch nimmt (s. Anforderung V7). Bei Sambasivan et al. ist hingegen ein Beispiel zu sehen, bei dem eine solche Darstellung auf das Nötigste komprimiert wurde. Der Effekt ist, dass die Darstellung zwar übersichtlich ist, aber nur wenige Informationen zu sehen sind. Bei keiner der Gleisplanvisualisierungen ist es möglich, Informationen dynamisch ein- und auszublenden (s. Anforderung V8). Die Darstellung bei Sambasivan et al. stellt lediglich Hardwareknoten, die Nachrichten zwischen Ihnen und die Berechnungen auf den Knoten dar (s. Anforderung V9). Die Gleisplanvisualisierung in Magpie stellt verschiedene Ressourcen und Entitäten, verknüpfende Ereignisse sowie die Aktivitätsperiode der Ressourcen und Entitäten dar. Hier kann es je nach Monitoring-Daten beliebig viele unterschiedliche Ressourcen und Entitäten geben, um jedoch eine abschließende Bewertung zu finden, wird die Begriffe Ressource und Entität als je eine Information gewertet. Die Gleisplanvisualisierung von Reynolds et al.

[5] Abbildung aus den Vortragsfolien zum USENIX-Paper [59] übernommen (s. Webseite http://issg.cs.duke.edu/pip/ (besucht am 13.03.2014)).

stellt Aufgaben, Nachrichten und Hardwareknoten dar. Damit ergeben sich für die drei Ansätze die folgenden Bewertungen:

Anforderung	V1	V2	V3	V4	V5	V6	V7	V8	V9
Magpie [1]	o	o	+	+	-	-	-	-	4
Sambasivan et al. [63]	-	-	+	-	-	-	+	-	3
Reynolds et al. [63]	-	+	+	-	-	-	-	-	3

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Der „j2eeprof“-Ansatz von Kłaczewski und Wytrębowicz [39] stellt eine weitere Aufrufbaum-Visualisierung vor. In dieser Darstellung wird jeder Methodenaufruf durch eine Box dargestellt (s. Abbildung 3.10). Wenn Methode `A()` Methode `B()` aufruft, wird die Box für `B()` über die Box für `A()` gesetzt. Die Boxbreite repräsentiert die Antwortzeit der jeweiligen Methode. Das heißt, dass die oben unbedeckte Fläche einer Box der Zeit entspricht, die in der jeweiligen Methode, für die Box steht, verbracht wird. Diese Darstellung ist deutlich kompakter als die bisher vorgestellten Darstellungen. Diese Darstellung kann sehr viele Methoden mit ihrer Antwortzeit innerhalb eines Diagramms und somit auf einen Blick darstellen. Allerdings hat dies auch den Nachteil, dass nicht für jede Methode alle Angaben auch schriftlich in den Boxen angegeben werden können.

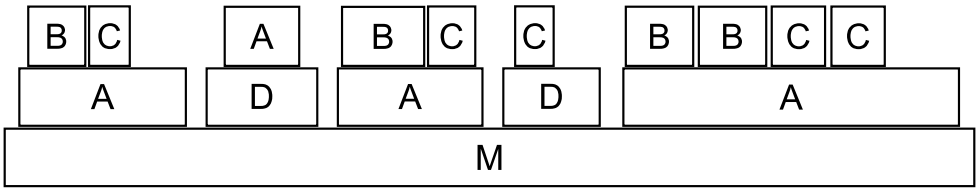

Abbildung 3.10: Beispiel für die Darstellung des j2eeprof-Ansatzes von Kłaczewski und Wytrębowicz [39].

Solch eine Visualisierung wird auch von den Flame-Graphen von Brendan Gregg [28, 29] realisiert. Analog zu j2eeprof wird hier eine kompakte Aufrufkontextbaum-Visualisierung realisiert. Bei Gregg wird auf der Grundlage einer Sampling-Datenreihe gearbeitet. Dabei umfasst jeder Datensatz, also jedes Sampling, eine Momentaufnahme der Stacks der derzeit ausgeführten Methoden. Bei den Flame-Graphen wird jede Methode wiederum durch eine Box realisiert, deren Breite für die Anzahl der Samplings im Testfall steht, bei denen die Methode ausgeführt wurde. Dabei soll dieser Wert die Gesamtausführungszeit der Methode annähern. Die Anordnung der Boxen im Diagramm

verdeutlicht auch hier die Aufrufhierarchie. Dabei gilt, dass die Farben der Boxen sowie ihre horizontale Abfolge keinerlei Bedeutung haben. Im Unterschied zu j2eeprof liegen für Flame-Graphen Beispiele, wie in Abbildung 3.11 zu sehen, vor und es existiert ebenso ein Skript, um Flame-Graphen zu erstellen. Die Beispiel-Flame-Graphen sind interaktive SVG-Grafiken. Da die Breite der Boxen aufgrund einer geringen Sampling-Zahl sehr knapp ausfallen kann, sind Methodenname und CPU-Zeit häufig nicht zu lesen. Dies wird bei Flame-Graphen so umgangen, dass der Methodenname, die Sampling-Zahl und der relative Anteil an der Gesamtzahl der Samplings neben dem „Function“-Text erscheinen, sobald sich der Mauszeiger über der Box befindet.

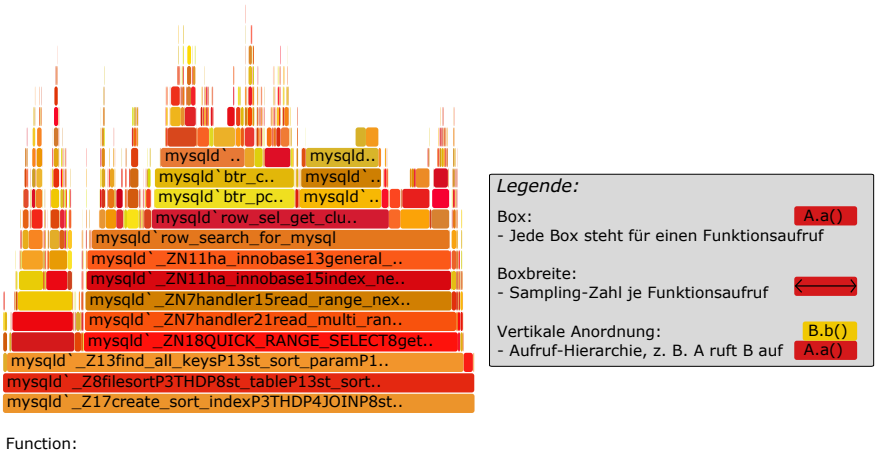

Abbildung 3.11: Der Beispiel-Flame-Graph (vorgestellt von Gregg [28, 29]) stellt den Aufrufbaum mit der Gesamt-CPU-Zeit dar; durch eine Legende erweitert.

Die Visualisierungen von j2eeprof und Flame-Graphen sind sehr ähnlich, so dass sich eine ähnliche Bewertung ergibt. Beide Ansätze zeigen keine Komponenten, aber doch Komponentenoperationen unter den anderen Methoden bzw. Funktionen (s. Anforderungen V1 und V2). Ebenso werden weder Hardwareknoten noch Ressourcen angezeigt (s. Anforderungen V3 und V4). Eine Entscheidung im Rahmen der Performance-Blame-Analysis ist ebenso nicht teil der Visualisierung (s. Anforderung V5). Bei dieser Darstellung kann der Systemarchitekt leicht visuell vergleichen, welche Methoden oder Funktionen die höchste Antwortzeit bzw. Sampling-Zahl hat. Dies muss aber vom Systemarchitekten noch auf die Performance-Blame-Analysis bezogen werden (s. Anforderung V6). Die Darstellung ist sehr kompakt, was dem Systemarchitekten einen sehr guten Überblick über alle beteiligten Methoden bzw. Funktionen verschafft (s. Anforderung V7). Dabei können die aus

Platzgründen ausgelassenen Informationen bei Flame-Graphen dynamisch angezeigt werden (s. Anforderung V8). Dies ist bei j2eeprof nicht der Fall. Die Darstellung in j2eeprof stellt die Methoden, die Aufrufhierarchie und die Antwortzeit dar (s. Anforderung V9). Flame-Graphen stellen die Funktionen, die Aufrufhierarchie, die Sampling-Zahl und den relativen Anteil an allen Samplings dar. Für j2eeprof bzw. Flame-Graphen ergeben sich insgesamt die folgenden Bewertungen:

Anforderung	V1	V2	V3	V4	V5	V6	V7	V8	V9
j2eeprof [39]	-	o	-	-	-	o	+	-	3
Flame-Graphen [28, 29]	-	o	-	-	-	o	+	+	4

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Der Ansatz von Jiang et al. [38] verwendet zur Gegenüberstellung von Verteilungsdiagrammen ein sogenanntes Bean-Plot. Dabei werden zwei Verteilungsdiagramme nebeneinander dargestellt, die sich eine X-Achse teilen (s. Abbildung 3.12). Bei Jiang et al. verläuft die X-Achse vertikal und je eine Y-Achse verläuft vom Nullpunkt der X-Achse horizontal nach links und nach rechts. Nun wird in jedes der beiden Koordinatensysteme eine Verteilungsfunktion eingetragen. Dadurch, dass die Diagramme sich nun quasi gespiegelt gegenüberstehen, kann man leicht vergleichen, welche Verteilung wo ihr Maximum hat oder wo mehr Ausreißer zu finden sind.

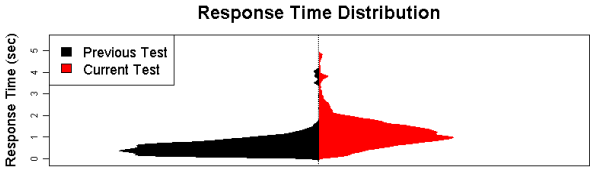

Abbildung 3.12: Beispiel für die Bean-Plot-Visualisierung des Ansatzes von Jiang et al. [38].

Bean-Plots bieten keinen Überblick über mehrere Operationen, sondern stellen Details zu Datenreihen je einer bestimmten Operation dar. Der Bean-Plot stellt somit keine Komponenten, Hardware-Knoten und Ressourcen sondern lediglich eine Komponentenoperation dar (s. Anforderungen V1 bis V4). Es wird ebenso keine Entscheidung dargestellt (s. Anforderung V5). Dabei ist die Interpretation relativ einfach, da hier ein einfacher Vergleich von zwei Datenreihen ermöglicht wird (s. Anforderung V6). Allerdings muss der Systemarchitekt den Vergleich noch auf die Performance-Blame-Analysis beziehen.

Die Darstellung ist dabei kompakt (s. Anforderung V7), aber statisch (s. Anforderung V8). Als Informationen werden nur die beiden Operationsdatenreihen dargestellt (s. Anforderung V9). Somit ergibt sich die Bewertung:

Anforderung	V1	V2	V3	V4	V5	V6	V7	V8	V9
Bean-Plot [38]	-	+	-	-	-	o	+	-	2

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Eine zum Bean-Plot ähnliche Darstellung kann mithilfe von Box-Whisker-Plots konstruiert werden. Bei Box-Whisker-Plots wird die Verteilung einer Operationsdatenreihe durch die Darstellung von Lageparametern und Ausreißern abstrahiert. Dabei werden je Datenreihe das Minimum, das 25%-Quartil, der Median, das 75%-Quartil und das Maximum dargestellt. Zusätzlich werden auch noch zwei Punkte zur Abgrenzung von Ausreißern angezeigt.[6] Dabei können die Begrenzungswerte mit dem Minimum bzw. Maximum zusammenfallen. Dann gibt es an der entsprechenden Stelle keine Ausreißer. Ein Beispiel für ein Box-Whisker-Plot ist in Abbildung 3.13 zu sehen. Um einen Vergleich der Datenreihen zu ermöglichen, werden in der Abbildung zwei Operationsdatenreihen an einer gemeinsamen Antwortzeit-Y-Achse (ms) dargestellt. Die Box in der Mitte stellt die Quartile dar und die Linien darüber bzw. darunter die Abgrenzung zu den Ausreißern. Darüber bzw. darunter werden mit Kreisen die Ausreißer jeweils einzeln dargestellt.

Wie schon der Bean-Plot ist der Box-Whisker-Plot [18] eine Möglichkeit zwei Operationsdatenreihen zu vergleichen. Somit liegt hier ein Bezug zu einer Komponentenoperation vor (s. Anforderung V2). Komponenten, Hardwareknoten und Ressourcen werden nicht dargestellt (s. Anforderungen V1, V3 und V4). Hier wird keine Entscheidung bezüglich der Performance-Blame-Analysis dargestellt (s. Anforderung V5). Der Vergleich der Datenreihen anhand der Darstellung ist einfach und intuitiv. Aber die Interpretation im Bezug auf die Performance-Blame-Analysis verbleibt beim Systemarchitekten (s. Anforderung V6). Die Darstellung ist zudem kompakt (s. Anforderung V7), aber statisch (s. Anforderung V8). Der Box-Whisker-Plot stellt die zwei Operationsdatenreihen jeweils mit dem Minimum, dem Maximum, den Quartilen

[6] Der obere und untere Abgrenzungspunkt o bzw. u für die Datenreihe D werden wie folgt berechnet [18, 55]:
$o = \max(\{x \in D | x \leq \text{75\%-Quartil} + 1,5 * (\text{75\%-Quartil} - \text{25\%-Quartil})\})$
$u = \min(\{x \in D | x \geq \text{25\%-Quartil} - 1,5 * (\text{75\%-Quartil} - \text{25\%-Quartil})\})$
Diese Werte sind auf höchstens das Maximum und mindestens das Minimum begrenzt.

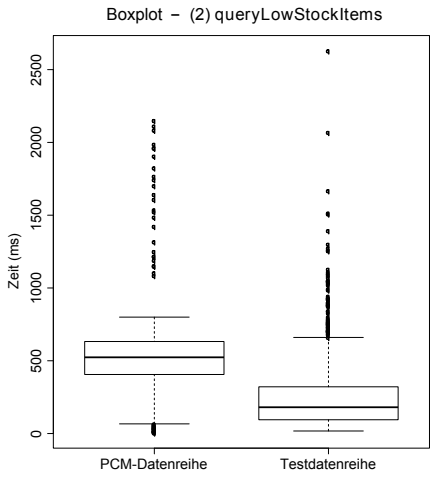

Abbildung 3.13: Zeigt ein Beispiel für ein Box-Whisker-Plot. Die Boxen stellen die Quartile dar und stehen jeweils für 50% der Werte einer Datenreihe.

und den Ausreißern dar (s. Anforderung V9). Somit ergibt sich die folgende Bewertung:

Anforderung	V1	V2	V3	V4	V5	V6	V7	V8	V9
Box-Whisker-Plot [18]	-	+	-	-	-	o	+	-	8

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Der Bereich der Softwarekartografie beschreibt, wie die Techniken der herkömmlichen Kartografie, wie sie z. B. für Stadtkarten verwendet werden, auf Software übertragen werden können. Krogmann et al. [45] haben gezeigt, dass Softwarekartografie auch zur Darstellung von Performance-Metriken geeignet ist. Sie haben sich dabei hauptsächlich mit der Präsentation von Ressourcenverbräuchen im Kontext komponentenbasierter Systeme befasst. Insbesondere haben sie verschiedene Sichten erstellt, die Daten aus verschiedenen Szenarien durch sogenannte Overlays visualisieren. Die Overlays werden über eine Basisvisualisierung gelegt und lassen sich an- und abschalten. So können die Daten, die in Overlays visualisiert werden, schnell verglichen werden.

Auch wenn Krogmann et al. zeigen, dass die Softwarekartografie interessante Möglichkeiten bietet und sich auch gut für Performance-Daten eignet, lassen sich die Ergebnisse nicht direkt auf die anderen Performance-Metriken

übertragen. Krogmann et al. fokussieren sich auf die Darstellung von Ressourcenverbräuchen und nicht auf die Darstellung von Antwortzeiten. Allerdings können die Konzepte, wie z. B. Overlays, die hier gezeigten Visualisierungen zusätzlich aufwerten.

3.4 Vergleichende Wertung

Zuvor wurden die verwandten Ansätze zur Performance-Analyse von Ablaufverfolgungsprotokollen einzeln vorgestellt und bewertet (s. Unterabschnitte 3.3.1 bis 3.3.3). Im Unterabschnitt 3.4.1 sollen nun Gemeinsamkeiten und Unterschiede herausgearbeitet werden, indem alle Anforderungen durchgegangen werden. Ähnlich werden in Unterabschnitt 3.4.2 auch die Visualisierungen (s. Unterabschnitt 3.3.4) anhand der Visualisierungsanforderungen miteinander verglichen. Zuletzt werden in Unterabschnitt 3.4.3 aus den Anforderungen und dem Stand der Technik die verbleibenden Anforderungen abgeleitet, die zu einer weiteren Verbesserung der Performance-Blame-Analysis führen.

3.4.1 Vergleich der Analyse-Ansätze

In den Unterabschnitten 3.3.1 bis 3.3.3 wurden die einzelnen Ansätze vorgestellt, mit denen eine Performance-Blame-Analysis möglich ist. Dabei wurden auch die einzelnen in Abschnitt 3.2 hergeleiteten Anforderungen bei jedem Ansatz bewertet. Tabelle 3.1 fasst nun alle Bewertungen bezüglich der Anforderungen zusammen. Aus der Übersicht lässt sich ersehen, dass kein bisheriger Ansatz alle Anforderungen erfüllt. Allerdings wird fast jede Anforderung durch wenigstens einen Ansatz erfüllt. Im Folgenden wird jede Anforderung durchgegangen und es wird darauf eingegangen, wie die Ansätze im Vergleich untereinander zu bewerten sind.

Anforderung A1 (Testfallbezug)

Die erste Anforderung verlangt, dass sich die Performance-Blame-Analysis auf einen bestimmten Testfall bezieht. Diese Anforderung ist elementar und kann von den meisten Ansätzen erfüllt werden, da hier die Ablaufverfolgungsprotokolle von einem oder mehreren Testläufen eines Testfalls untersucht werden. Nur der Ansatz von Groenda versucht, eine allgemeingültige Aussage zu treffen. Daher untersucht dieser Ansatz keinen bestimmten Testfall. Des

Tabelle 3.1: Bewertung der verwandten Arbeiten bezüglich der Anforderungen

Ansatz \ Anf.	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
Srinivas et al. [70]	+	o	-	o	-	+	o	+	o	+	-	-	-
„Magpie“ [1]	+	o	-	o	o	o	+	+	o	+	-	o	+
Rutar et al. [62]	+	-	-	o	-	-	-	+	+	+	-	-	-
„RanCorr“ [48]	+	o/+	o	o/+	-	+	+	+	-	+	o	+	+
Jiang et al. [38]	+	-	-	-	-	-	+	+	-	+	o	+	+
Sambasivan et al. [63]	+	-	-	-	-	-	+	+	-	+	o	+/o	-
Reynolds et al. [59]	+	o/+	+	+	o	-	+	+	o/-	+	+	o	+
Groenda [31, 30]	-	o	-	o	-	+	-	+	+	+	+	+/o	-
eingebaute Tests [9, 26]	o	-	-	-	-	+	+	+	+	+	o	-	-

Bewertungsskala:

\+ Anforderung ist erfüllt

o Anforderung ist mit Einschränkungen erfüllt

\- Anforderung ist nicht erfüllt

Weiteren werden bei eingebauten Tests von den Komponentenentwicklern vorgegebene Testfälle durchgespielt, so dass es nicht möglich ist, gezielt einen bestimmten Testfall zu untersuchen.

Anforderung A2 (Bewertung der Komponenten)

Die zweite Anforderung besagt, dass die untersuchten Ansätze die Komponenten im Hinblick auf die untersuchte Fehlerwirkung bewerten können, so dass sich der Fehler den richtigen Komponentenentwicklern zuordnen lässt. Eine direkte Bewertung von Komponenten erfolgt lediglich beim Ansatz RanCorr von Marwede et al. [48]. Bei diesem Ansatz wird durch Aggregation der Werte der Komponentenoperationen auf die Komponenten eine Komponentenbewertung erstellt. In der Bewertung werden auch Performance-Verbesserungen als berichtenswerte Änderung registriert, was im Rahmen der Performance-Blame-Analysis nicht von Interesse ist. In den anderen Ansätzen muss die Bewertung für die Komponentenoperationen nach Anforderung A4 manuell auf die Komponenten übertragen werden. Die Bewertung von Komponentenoperationen fußt üblicherweise direkt auf den Beobachtungen, während die Bewertung von Komponenten auf daraus berechneten Indikatoren aufbaut. Dies ist der Fall, da Komponentenoperationen, im Unterschied zu Komponenten, direkt während der Ausführung beobachtet werden können. Aus Sicht der Implementierung sind Komponenten häufig nicht mehr als eine Gruppe von Klassen (bei objektorientierter Implementierung), auf die über Schnittstellen zugegriffen wird. Während der Ausführung kann so bestenfalls die Instanziierung der Klassen und vor allem der Aufruf der Methoden, also auch der Aufruf der Komponentenoperationen, beobachtet werden. So können aus den Beobachtungen nur die Bewertung der Komponentenoperationen direkt abgeleitet werden. Falls keine bessere Heuristik vorliegt, kann der Systemarchitekt eine Komponente beschuldigen, sobald eine ihrer Komponentenoperationen beschuldigt wird. Mit dieser Heuristik kann der Systemarchitekt die Ergebnisse eines Ansatzes leicht manuell von der Operations- auf die Komponentenebene übertragen, wenn der Ansatz die Komponentenoperationen konkret beschuldigt, wie es bei Pip von Reynolds et al. [59] möglich ist. Viele Ansätze treffen aber keine klare Entscheidung sondern errechnen lediglich einen Indikator, der einen bestimmten Aspekt für beispielsweise Methoden, Komponentenoperationen oder Variablen charakterisiert. Dieser Indikator lässt sich meist nicht trivial auf eine Komponente übertragen. Dies ist insbesondere beim Ansatz von Rutar et al. [62] der Fall, bei dem die Blame-Werte auf Variablenebene errechnet werden.

Anforderung A3 (Bewertung des Systems)

Anforderung A3 verlangt, dass der Ansatz entscheiden kann, ob der Fehler ausschließlich im Bereich der Systementwickler liegt. Damit muss auch ausgeschlossen werden, dass der Fehler von einzelnen Komponenten herrührt. Dieser Ausschluss ist aber nur möglich, wenn die Bewertung der Komponenten bzw. Komponentenoperationen nicht über einen unsicheren Indikator erfolgt. Diese unsicheren Indikatoren erfassen entweder nicht alle Komponentenoperationen (Jiang et al. [38] und Sambasivan et al. [63]) oder aber liefern nur einen Hinweis, dass die Komponentenoperation als Beschuldigte infrage kommt (Srinivas und Srinivasan [70], Barham et al. [1] und Rutar et al. [62]). Eine Systembewertung ist zudem nur dann möglich, wenn das System bei der Analyse auch betrachtet wird. Bei Groenda [31, 30] und bei eingebauten Tests [9, 26] werden die Komponenten jedoch isoliert betrachtet. Demgegenüber kann bei Reynolds et al. [59] und RanCorr [48] die Zuordnung zu den Systementwicklern gelingen und gleichzeitig ausgeschlossen werden, dass sich die Komponenten fehlerhaft verhalten haben. Bei RanCorr muss allerdings erwähnt werden, dass hier generell nach Abweichungen beschuldigt wird, so dass hier möglicherweise Komponenten beschuldigt werden, die sich im Vergleich zur Referenz verbessert haben.

Anforderung A4 (Bewertung der Komponentenoperationen)

Die Anforderung A4 besagt, dass die Ansätze die Komponentenoperationen im zu untersuchenden Testfall einzeln bewerten können sollen. Wie schon erläutert, können Komponentenoperationen direkt bei der Ausführung beobachtet werden und deshalb stellt dies meist die Grundlage der jeweiligen Ansätze dar. So lässt sich auch erklären, dass die meisten Ansätze diese Anforderung in gewisser Weise erfüllen. Dagegen sind die Ansätze von Jiang et al. [38] und Sambasivan et al. [63] auf die Analyse von Ereignisketten spezialisiert und erfassen somit nicht alle Komponentenoperationen. Zudem bewerten eingebaute Tests [9, 26] zwar die Komponentenoperationen, aber die von den Komponentenentwicklern festgelegten Kriterien sind für die Performance-Blame-Analysis ungeeignet. Die Anforderung wird von den Ansätzen, die sich auf nur ein Ablaufverfolgungsprotokoll stützen, zumindest teilweise erfüllt. Sie bewerten zwar die Komponentenoperationen, geben aber jeweils nur einen Hinweis auf fehlerhafte Komponentenoperationen. Dies liegt darin begründet, dass diese Ansätze keinerlei erwartete Performance berücksichtigen. Demgegenüber liefert der Ansatz von Groenda [31, 30] eine gute Bewertung der Komponentenoperationen, bezieht sich aber nicht auf den

zu untersuchenden Testfall. RanCorr [48] kann die Komponentenoperationen bewerten, ist dabei aber auf ein fehlerfreies Referenzablaufverfolgungsprotokoll angewiesen. Der Ansatz Pip von Reynolds et al. [59] ist schließlich der einzige Ansatz, der die Komponentenoperationen aufgrund der für Pip spezifizierten Anforderungen zufriedenstellend bewertet.

Anforderung A5 (Bewertung der Komponentenverwendung)

Die Anforderung A5 fordert, dass die Ansätze die Komponentenverwendung und damit verschiedene Systemaspekte direkt untersuchen. Diese Anforderung wird von keinem Ansatz erfüllt, da die meisten Ansätze diesen Aspekt ignorieren. Lediglich Magpie [1] bietet zusätzlich zu den Daten über Komponenten auch Daten zu Betriebssystemaspekten, die eine solche Analyse ermöglichen können. Zusätzlich erlaubt es der Ansatz von Reynolds et al. [59] Erwartungen an knotenüberspannende Pfade zu erstellen, wodurch Aussagen bezüglich der Komponentenverteilung getroffen werden können.

Anforderung A6 (Komponentenbegriff)

Die Anforderung A6 besagt, dass die Ansätze nicht nur komponentenbasierte Systeme untersuchen können, sondern auch selbst Komponenten und Komponentenoperationen kennen. Magpie [1] kennt zwar möglicherweise Komponenten, da hier flexibel festgelegt werden kann, welche Daten erhoben werden. Bei der Analyse werden diese Daten aber nur rudimentär berücksichtigt. Die anderen Ansätze, die Komponenten kennen, verwenden dies auch für die Analyse. Insbesondere RanCorr [48] nutzt dies konsequent, da hier abweichendes Verhalten auch auf die Komponenten übertragen wird.

Anforderung A7 (Komponentenquelltext)

Anforderung A7 fordert, dass die Ansätze auch Komponenten ohne den vorliegenden Quelltext analysieren können. Das ist immer dann der Fall, wenn der Quelltext nicht in die Analyse mit einbezogen wird, was nur bei wenigen Ansätzen der Fall ist. Bei Srinivas und Srinivasan [70] wird zwar nicht direkt der Quelltext einbezogen, aber es muss möglich sein, einen Profiler auf die Komponente anzuwenden. Eine direkte Analyse des Quelltexts erfolgt insbesondere bei Rutar et al. [62], die eine statische Analyse für die Blame-Bewertung nutzen. Aber auch Groenda [31, 30] nutzt den Quelltext, da er die PCM-Diagramme damit abgleicht.

Anforderung A8 (Automatisierung)

Die Anforderung A8 besagt, dass die Schritte Analyse und Präsentation der Performance-Blame-Analysis automatisiert sein sollen. Alle untersuchten Ansätze können dies erfüllen. Es kann jedoch eine manuelle Übertragung der Ergebnisse auf die Performance-Blame-Analysis nötig sein. Beispielsweise müssen die Ergebnisse für verschiedene Ansätze noch manuell auf die Komponenten übertragen werden.

Anforderung A9 (Eingaben)

Anforderung A9 fordert, dass die Eingaben für die Ansätze mit wenig Aufwand zu ermitteln sind und sie wiederverwendet werden können. Die Ansätze, die nur auf einem Ablaufverfolgungsprotokoll aufsetzen, benötigen meist nur wenige Eingaben und diese lassen sich teils gut wiederverwenden. Die Ansätze, die zwei Ablaufverfolgungsprotokolle vergleichen, brauchen ein Referenzablaufverfolgungsprotokoll als Eingabe. Wenn es aus einem vorherigen Testlauf vorliegt, muss sichergestellt sein, dass dieser Testlauf keine Performance-Fehler beinhaltet hat und dass das Protokoll kompatibel zu einem aktuellen Testlauf ist. Das heißt, die beobachteten Anfragepfade sollten idealerweise identisch sein. Wenn das Referenzprotokoll nicht vorliegt, muss es aufwendig konstruiert werden. Dies ist aufwendiger als die Spezifikationen für den Ansatz Pip von Reynolds et al. [59] zu erstellen. Der Ansatz unterstützt den Systemarchitekten dabei, indem es eine Spezifikationssprache bietet, mit der vergleichsweise einfach Pfade und die Performance-Erwartungen daran formuliert werden können. Der Systemarchitekt muss dabei darauf achten, dass diese Spezifikation vollständig ist. Der Ansatz von Groenda [31, 30] benötigt die Komponentenimplementierung sowie die PCM-Spezifikation als Eingaben. Beides wird idealerweise von den Komponentenentwicklern geliefert. Bei eingebauten Tests sind keine Eingaben nötig, da hier nur die von den Komponentenentwicklern vorgegebenen Fälle durchgespielt werden.

Anforderung A10 (Ergebnisdokument)

Alle Ansätze liefern Ergebnisse, die laut Anforderung A10 als Ergebnisdokument gelten können. Der Inhalt der Ergebnisse wird im Folgenden durch die Anforderungen A11 bis A13 bewertet.

Anforderung A11 (Blame-Entscheidung)

Die Anforderung A11 verlangt, dass der Ansatz klar entscheidet, ob und welche Komponenten zur untersuchten Fehlerwirkung beitragen. Die Ansätze, die nur ein Ablaufverfolgungsprotokoll analysieren, können prinzipbedingt nur eine Rangliste mit Auffälligkeiten ausgeben, bei der die Sortierung nicht notwendigerweise den Beitrag der Auffälligkeit zur Fehlerwirkung widerspiegelt. Bei Ansätzen, die zwei Ablaufverfolgungsprotokolle vergleichen, können Abweichungen festgestellt werden und es kann auch bewertet werden, ob eine Verschlechterung gegenüber dem Referenzablaufverfolgungsprotokoll eingetreten ist. Diese Bewertung ist allerdings nur valide, wenn im Referenzablaufverfolgungsprotokoll keine Performance-Fehler zu finden sind. Eine zuverlässige Bewertung ist bei direkter Spezifikation der erwarteten Performance möglich, wie es beispielsweise Pip (Reynolds et al. [59]) zeigt. Dabei muss die Performance-Spezifikation auch hier korrekt und vollständig sein, allerdings lässt sich das bei einer dafür geeigneten Spezifikationssprache leichter überblicken als bei einem Ablaufverfolgungsprotokoll. Außerdem ist der Bezug zum untersuchten Testfall eine Grundvoraussetzung für eine valide Performance-Blame-Analysis. Das ist beim Ansatz von Groenda [31, 30] und bei eingebauten Tests [9, 26] nicht der Fall.

Anforderung A12 (Übersichtlichkeit)

Die Anforderung A12 besagt, dass Ergebnisse übersichtlich dargestellt werden sollen. Dabei ist es entscheidend, dass der Systemarchitekt zunächst einen Überblick gewinnen kann und trotzdem alle nötigen Details einsehen kann. Idealerweise werden die Details, auf Wunsch, durch weitere relevante Zusatzinformationen ergänzt (s. Anforderung A13). Die Ansätze von Srinivas und Srinivasan [70] und Rutar et al. [62] bieten zwar eine Rangliste zur Übersicht, aber keinerlei weiteren Detailinformationen. Magpie von Barham et al. [1] bietet neben der Cluster-Rangliste auch Gleisplanvisualisierungen inklusive der jeweiligen Ressourcenverbräuche an. Jedoch sind weder die Cluster-Rangliste noch die Gleisplanvisualisierung sonderlich übersichtlich, da die Darstellungen sehr viele Details beinhalten. Die Ansätze von Marwede et al. [48] sowie Jiang et al. [38] bieten eine gute Übersicht, bei der Details in weiteren Stufen zuschaltbar sind. Ähnlich sieht es auch bei Sambasivan et al. [63] aus, doch ist hier neben einer Übersicht über die Abweichungen zum Referenzablaufverfolgungsprotokoll nur die übersichtliche Vergleichsansicht, die einzelnen Schritte in Anfragenpfaden aus zwei Testläufen vergleicht, verfügbar. Bei Reynolds et al. [59] fehlt eine Übersicht über

Pfade, Komponenten oder Komponentenoperationen, dafür kann Pip aber mit einer grafischen Aufrufbaumdarstellung aufwarten. Zusätzlich können Aufrufpfade in einer Gleisplanvisualisierung dargestellt werden, zudem können weitere Diagramme zu einzelnen Komponentenoperationen angezeigt werden. Vorbildlich ist die Visualisierung der Prüfentscheidungen im Ansatz von Groenda [31, 30], bei der die Entscheidungen in den Diagrammen des Kontrakts eingebettet sind. So schafft Groenda gleichzeitig eine Übersicht über die Komponenten und eine Detailansicht der Komponentenoperationen im Kontext des Performance-Modells im Rahmen der PCM-Diagramme. Bei eingebauten Tests [9, 26] wird lediglich mit dem Testprotokoll eine Übersicht über bestandene und nicht bestandene Testfälle ausgegeben.

Anforderung A13 (zusätzliche Performance-Metriken)

Die Anforderung A13 verlangt, dass neben den Hauptanalyse-Metriken noch zusätzliche entscheidungsrelevante Metriken zur Verfügung stehen, um die Analyse besser nachvollziehbar zu machen. Die analysierten Ansätze sind häufig stark fokussiert und verzichten auf diese zusätzlichen Informationen. Jedoch lässt sich bei Magpie von Barham et al. [1] auswählen, welche Entitäten beobachtet werden sollen. Zusätzlich stehen immer auch die Beobachtungen aus dem Betriebssystem sowie die Ressourcenverbräuche zur Verfügung. Auch bei Marwede et al. [48] stehen Informationen wie Aufrufhäufigkeit und Verteilung der Vergleicherwerte zur Verfügung. Bei Jiang et al. [38] sind es die zusätzlichen Ansichten über die Verteilung der Antwortzeiten. Bei Reynolds et al. [59] werden neben der Antwortzeitverteilung beispielsweise auch die Antwortzeiten über einer Zeitachse angezeigt.

3.4.2 Vergleich der Visualisierungen

Im Folgenden werden nun die Visualisierungen, die in Unterabschnitt 3.3.4 vorgestellt wurden nochmals vergleichend im Bezug auf die Visualisierungsanforderungen bewertet. Zunächst fasst Tabelle 3.2 die Ergebnisse der vorherigen Bewertung zusammen. Dabei fällt auf, dass die Visualisierungen teils sehr unterschiedliche Aspekte darstellen. Außer bei der Gleisplanvisualisierung von Sambasivan et al. werden überall Komponentenoperationen dargestellt (s. Anforderung V2). Komponenten werden nur bei RanCorr zusätzlich dargestellt, aber auch bei Magpie, wenn das Monitoring es hergibt (s. Anforderung V1). Hardwareknoten werden von RanCorr und den drei Gleisplanvisualisierungen dargestellt (s. Anforderung V3). Ressourcen werden hingegen lediglich bei Magpie dargestellt (s. Anforderung V4).

Tabelle 3.2: Übersicht über Visualisierungen und ihre Bewertung bezüglich der Visualisierungsanforderungen

Anforderung	V1	V2	V3	V4	V5	V6	V7	V8	V9
Profiler [19, 20]	-	o	-	-	o	o	o	+	4
RanCorr [48]	+	+	+	-	o	o/+	o	+	7
Magpie [1]	o	o	+	+	-	-	-	-	4
Sambasivan et al. [63]	-	-	+	-	-	-	+	-	3
Reynolds et al. [63]	-	+	+	-	-	-	-	-	3
j2eeprof [39]	-	o	-	-	-	o	+	-	3
Flame-Graphen [28, 29]	-	o	-	-	-	o	+	+	4
Bean-Plot [38]	-	+	-	-	-	o	+	-	2
Box-Whisker-Plot [18]	-	+	-	-	-	o	+	-	8

Bewertungsskala:

+	Anforderung ist erfüllt
o	Anforderung ist mit Einschränkungen erfüllt
-	Anforderung ist nicht erfüllt

Die Visualisierungsanforderungen V1 bis V4 geben Auskunft darüber, was in den jeweiligen Darstellungen zu sehen ist. Wenn man die einzelnen Visualisierungen vergleicht, fällt aber auf, dass sie unterschiedliche Zwecke haben. Um die Darstellungen zu klassifizieren, soll daher auch erwähnt werden, ob alle Anfragen an Operationen, die Bearbeitung einer bestimmten Systemoperation oder lediglich die Anfragen an eine Operation dargestellt werden. Die Profiler-Darstellungen sowie RanCorr, j2eeprof und Flame-Graphen geben alle Anfragen an Operationen eines Testlaufs in einer Darstellung wider. Dagegen stellen die Gleisplanvisualisierungen lediglich die Bearbeitung einer Systemoperation dar. Zuletzt detaillieren Bean-Plots und Box-Whisker-Plots die Antwortzeitverteilung der Anfragen an eine bestimmte Operation. Jede dieser Klassen erfüllt einen bestimmten Zweck, so lässt sich mit einer Übersicht über alle Anfragen ein guter Überblick über das Verhalten aller Operationen erhalten. Die isolierte Überblicksdarstellung der Bearbeitung bestimmter Systemoperationen lässt eine Beurteilung des Verhaltens der beteiligten Operationen in diesem speziellen Kontext zu. Die detaillierte Darstellung einer Operationsdatenreihe ergänzt die bisher diskutierten kontextbezogenen Überblicksdarstellungen durch die detaillierte Darstellung des jeweiligen Operationsverhaltens.

Nachdem nun festgestellt wurde, was die einzelnen Darstellungen zeigen und wozu die jeweiligen Darstellungen genutzt werden, muss nun die Qualität der

Darstellungen beleuchtet werden. Keine der Darstellungen zeigt eine richtige Entscheidung (s. Anforderung V5). Wenngleich insbesondere die Darstellung von RanCorr wegen der Bewertung aller drei dargestellten Abstraktionsebenen leicht zu interpretieren ist (s. Anforderung V6), müssen auch dort die Ergebnisse manuell auf die Performance-Blame-Analysis übertragen werden. Anforderung V7 besagt, dass Darstellungen platzsparend sein sollen und dies können viele der untersuchten Darstellungen für sich in Anspruch nehmen. Bei den Übersichten über alle Anfragen trifft dies lediglich auf j2eeprof und Flame-Graphen zu. Allerdings ermöglichen es die Profiler-Darstellungen sowie RanCorr Informationen ein- und auszublenden (s. Anforderung V8), um so an Übersichtlichkeit zu gewinnen. Bei den Gleisplanvisualisierungen ist nur die Darstellung von Sambasivan et al. platzsparend und hier wird es durch wenige Informationstypen in der Darstellung erkauft. Bei den Detaildarstellungen können beide als kompakt gelten. Schließlich bietet die Anzahl der dargestellten Informationen nach Anforderung V9 nur wenig Unterscheidungspotenzial. Lediglich die Darstellung von RanCorr sticht heraus, da hier mehrere Abstraktionsebenen dargestellt werden. Auch der Box-Whisker-Plot stellt mit den unterschiedlichen Punktwerten und der Berechnung der Ausreißer außergewöhnlich viele unterschiedliche Informationen dar.

3.4.3 Verbleibende Anforderungen

In der vergleichenden Wertung der Analyse-Ansätze (s. Unterabschnitt 3.4.1) wird deutlich, dass es einige Schlüsselkonzepte gibt, die für eine gute Wertung verantwortlich sind. Dazu gehört zunächst, dass ein Ansatz nach Anforderung A1 den zu untersuchenden Testfall betrachten muss. Wenn dies nicht der Fall ist, so sind die Schlussfolgerungen der Analyse nach den Anforderungen A2-A5 nur begrenzt aussagekräftig. Die folgende Anforderung ist die Grundlage für einen guten Performance-Blame-Analysis-Ansatz:

> *Anforderung Z1:*
> Die Analyse nach den Anforderungen A2-A5 erfolgt nach Anforderung A1 im Hinblick auf den zu untersuchenden Testfall.

Zudem hat sich erwiesen, dass prinzipbedingt nur eine Analyse, die die Testdaten mit einer Spezifikation vergleicht, zuverlässige Ergebnisse liefert. Die Bewertung anhand eines Ablaufverfolgungsprotokolls kann nur Auffälligkeiten entdecken, die nicht unbedingt relevant für die Performance-Blame-Analysis sind. Wenn ein Referenzablaufverfolgungsprotokoll als Spezifikation genutzt wird, ist dies mit der besonderen Schwierigkeit verbunden, dass ein Durchlauf der Software mit allen möglicherweise existierenden Fehlern als

Referenz gekennzeichnet wird. Aufgrund der großen Datenmenge eines Referenzablaufverfolgungsprotokolls kann der Systemarchitekt dabei nur schwer überschauen, ob die Daten wirklich als Performance-Erwartungswerte geeignet sind. Dies ist einfacher, wenn geeignete Spezifikationswerkzeuge, wie zum Beispiel eine Spezifikationssprache, zum Einsatz kommen, um die erwartete Performance zu spezifizieren. Ein geeignete Spezifikationssprache erleichtert es dem Systemarchitekten nach Anforderung A9 zudem, die Eingaben für den Performance-Blame-Analysis-Ansatz zu erstellen. Daraus ergibt sich die folgende Anforderung:

Anforderung Z2:
Für die Analyse nach den Anforderungen A2-A5 wird eine geeignete Spezifikation für die erwartete Performance verwendet. Dabei stehen dem Systemarchitekten Spezifikationswerkzeuge, wie zum Beispiel eine Spezifikationssprache zur Verfügung, die die spezifizierte erwartete Performance verständlich erfasst und nach Anforderung A9 mit möglichst geringem Aufwand erstellt werden kann.

Bei der Analyse hat sich herausgestellt, dass es für die Analyse der Komponenten und Komponentenoperationen von Vorteil ist, wenn der Performance-Blame-Analysis-Ansatz mithilfe eines Komponentenbegriffs nach Anforderung A6 agiert und so Komponenten und Komponentenoperationen direkt analysieren kann. Von diesem Umstand sind nicht nur die Analyseanforderungen A2 bis A5 betroffen. Ein Komponentenbegriff kann auch dazu genutzt werden, dass sich die für den Ansatz erforderlichen Eingaben bei der Analyse eines komponentenbasierten Systems einfacher erfassen lassen (Anforderung A9) und dass die Blame-Entscheidungen nach Anforderung 11 direkt auf Komponentenebene gefällt werden können. Zudem können Komponenten und Komponentenoperationen verwendet werden, um eine übersichtliche Visualisierung zu erreichen (s. Anforderung 12).

Anforderung Z3:
Die Analyse nach den Anforderungen A2-A5 ist gemäß Anforderung A6 auf komponentenbasierte Systeme spezialisiert.

Zusätzlich sollte das Ergebnisdokument aus Anforderung A10 nicht nur, wie in Anforderung A11 gefordert, eine klare Blame-Entscheidung enthalten sondern auch nach Anforderung A12 übersichtlich gestaltet sein. Dies hat zwei Aspekte. Zum einen heißt das, dass die Visualisierung für die Performance-Blame-Analysis relevante Interpretationshilfen nach Anforderung V6 liefern soll, wobei die Visualisierung gleichzeitig kompakt gestaltet ist (Anforderung V7). Zum anderen hat sich in der vergleichenden Wertung der Visuali-

sierungen (s. Unterabschnitt 3.4.2) herausgestellt, dass es Visualisierungen für Übersichten und Detailansichten gibt. Um eine übersichtliche Visualisierung zu erreichen, müssen beide Visualisierungsarten in einen Performance-Blame-Analysis-Ansatz integriert werden. Daraus ergeben sich die folgenden verbleibenden Anforderungen an die Visualisierung:

Anforderung Z4:
Die Visualisierung soll eine klare Entscheidung bezüglich der Performance-Blame-Analysis gemäß Anforderung V5 (analog Anforderung A11) enthalten. Die angebotenen Interpretationshilfen sollen analog dazu direkt auf die Performance-Blame-Analysis ausgerichtet sein (Anforderung V6). Gleichzeitig soll die Darstellung nach Anforderung V7 kompakt sein.

Anforderung Z5:
Die Visualisierungen des Performance-Blame-Analysis-Ansatzes bieten eine Übersicht und auch Detailansichten.

Während jede verbleibende Anforderung für sich genommen lediglich den Stand der Entwicklung widerspiegelt, entsteht aus der Kombination der verbleibenden Anforderungen eine neue Qualität der Peformance-Blame-Analysis. Insbesondere die Kombination eines Ansatzes der konsequent einen Komponentenbegriff zur Analyse nutzt und gleichzeitig eine geeignete Spezifikationssprache zum Ausarbeiten der erwarteten Performance zur Verfügung stellt, ermöglicht eine Verbesserung. Kombiniert man dies mit einer Visualisierung, die die getroffenen Blame-Entscheidungen auch direkt übersichtlich darstellt, so erhält man einen im Vergleich zu den verwandten Arbeiten deutlich verbesserten Peformance-Blame-Analysis-Ansatz.

Zusammenfassend kann man sagen, dass diese Arbeit eine komponentenbasierte Analysemethode für Testfallbeobachtungen eines komponentenbasierten Systems mit einer übersichtlichen Darstellung von Blame-Entscheidungen im Ergebnis liefern soll. Die Darstellung soll dabei eine kompakte Übersicht mit Interpretationshilfen und Entscheidungen zur Performance-Blame-Analysis bereitstellen, die sinnvoll mit Detaildarstellungen kombiniert wird.

Kapitel 4

Der PBlaman-Prozess

In diesem Kapitel wird der PBlaman-Prozess (s. Abbildung 4.1) vorgestellt. Die Erläuterungen stützen sich dabei auf die bisherige Arbeit zu PBlaman [11, 12, 13]. Zunächst werden die Palladio-basierten Testfälle (s. Abschnitt 4.1) erläutert, die im Rahmen der Systementwicklung bei den Tests eingesetzt werden sollen. Die Testfälle ermöglichen es, Performance-Testfälle mithilfe des PCMs zu formulieren, die auch als Simulationsszenarien genutzt werden können. Bei der späteren Performance-Blame-Analysis steht so für jeden Testfall immer auch ein äquivalentes Simulationsszenario zur Verfügung (s. Abschnitt 4.1).

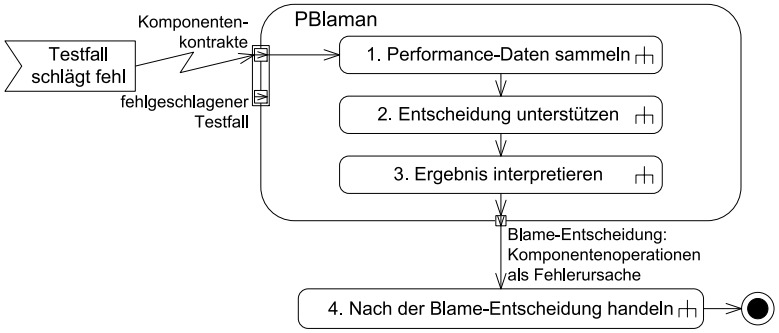

Abbildung 4.1: Das UML-Aktivitätendiagramm zeigt die Hauptaktivitäten des PBlaman-Ansatzes

Danach werden in diesem Kapitel die Hauptaktivitäten von PBlaman (s. Abbildung 4.1) näher erläutert. In Abschnitt 4.2 wird beschrieben, wie die Antwortzeitdatenreihen aus dem Test und der Performance-Vorhersage gesammelt werden. Diese Daten werden im Rahmen der Entscheidungsunterstützung bewertet (s. Abschnitt 4.3). Dazu werden die beiden Antwortzeitdatenreihen mit geeigneten Entscheidungskriterien verglichen. Die Werte der Entscheidungskriterien und das Vergleichsergebnis werden dann visualisiert. Aus den numerischen Ergebnissen und den Visualisierungen ergeben sich die Entscheidungsunterstützungsartefakte. Der Systemarchitekt interpretiert diese

Artefakte (s. Abschnitt 4.4) und entscheidet, welche Komponentenoperationen beschuldigt werden zum untersuchten Fehler beizutragen. Schließlich werden in Abschnitt 4.5 die bei PBlaman verwendeten Werkzeuge beschrieben. Dabei wird darauf eingegangen, wer die Werkzeuge verwendet und bei welchen Prozessschritten sie zum Einsatz kommen.

4.1 Palladio-basierte Testfälle

Eine wesentliche Annahme des PBlaman-Ansatzes ist es, dass der untersuchte Testfall und die dazu gehörende PCM-Instanz äquivalent sind. Das heißt, sie sollen beide die unter Performance-Gesichtspunkten selbe Eingabelast an ein System mit derselben Komposition und derselben Komponentenverteilung betrachten. Denn nur dann kann man davon ausgehen, dass die beobachteten Antwortzeitwerte aus dem Test und der Simulation der PCM-Modelle vergleichbar sind. Um diese Äquivalenz konstruktiv sicherzustellen, wird im Folgenden die Testfallnotation „Palladio-basierte Testfälle“ [13, 12, 11] vorgestellt. Im Rahmen dieser Testfallnotation werden Performance-Testfälle schon mithilfe des PCMs spezifiziert. Somit können die Testfälle leicht auch mittels der Palladio-Workbench simuliert werden.

Die Testfallnotation soll während der Systementwicklung von den Testern genutzt werden, um Performance-Testfälle zu entwerfen. Dabei wird die Testfallnotation Palladio-basierte Testfälle zunächst nur genutzt, um den Testfallentwurf ohne konkrete Daten zu erstellen. Die Testfallnotation bindet PCM-Diagramme in die Standard-Testfallnotation IEEE 829 [34] ein. Die Einzelheiten der Testfallnotation in Anlehnung an den Standard werden in Unterabschnitt 4.1.1 erläutert. Um den Testfall zu konkretisieren, soll der Testfallentwurf zunächst auf Grundlage der PCM-Diagramme mittels eines Quelltextgenerators in ein ausführbares Testskript überführt werden. Dann spezifiziert der Systemarchitekt die Testdaten sowie andere noch fehlende Aspekte. Fischer hat im Rahmen seiner Bachelorarbeit [25] einen Generator für JUnit-Testskripte konzipiert und prototypisch implementiert. In Unterabschnitt 4.1.2 wird erläutert, welche Aspekte eines Testskripts generiert werden und wie die Elemente der PCM-Diagramme aus dem Testfall auf ein JUnit-Testskript abgebildet werden.

4.1.1 Aufbau Palladio-basierter Testfälle

Der Aufbau der Palladio-basierten Testfälle richtet sich nach der Standard-Testfallnotation IEEE 829 [34] (vgl. Unterabschnitt 2.4.2). Bei diesen Testfällen

werden einige Teile der Testfallspezifikation und der Testfallablaufspezifikation eines IEEE 829-Testfalls mit PCM-Diagrammen spezifiziert. Dies betrifft vor allem die Eingaben, die Schritte des Testfallablaufs sowie Teile der Testumgebung.

Die meisten Informationen, die laut IEEE 829 für einen Testfall benötigt werden, kommen auch in einer PCM-Instanz vor. Das Verwendungsmodell (vgl. Unterabschnitt 2.3.5) modelliert beispielsweise, wie ein System von den Benutzern verwendet wird. Damit ist es gut geeignet, die Eingaben im Rahmen der Testfallspezifikation anzugeben (vgl. Punkt 2.3 in Tabelle 4.1). Zudem kann das Verwendungsmodell auch die einzelnen Schritte der Benutzerinteraktion darstellen, wie sie in der Testfallablaufspezifikation spezifiziert werden. Das Verteilungsmodell und das Ressourcenmodell des PCMs können hingegen für die Modellierung der Testumgebung genutzt werden. Dazu wird das Verteilungsmodell in die Testfallspezifikation (vgl. Punkt 2.5 in Tabelle 4.1) eingebunden. Das Ressourcenmodell wird durch das Verteilungsmodell referenziert. Daher müssen die genannten PCM-Diagramme aus einer zusammenhängenden PCM-Instanz stammen. Diese PCM-Instanz muss ebenfalls im Testfall referenziert werden, beispielsweise durch das Nennen der zugehörigen Dateien unter Punkt 2.6. in der Testfallspezifikation (s. Tabelle 4.1).

Tabelle 4.1: Palladio-basierte Testfälle als Abwandlung des Standard-Testfallnotation IEEE 829 [34] (vgl. Tabelle 2.1)

2	**Details (once per test case)**
2.1	Test case identifier
2.2	Objective
2.3	Inputs ⇒*PCM-Verwendungsmodell*
2.4	Outcome(s)
2.5	Environmental needs ⇒*PCM-Verteilungsmodell*
2.6	Special procedural requirements ⇒*Referenz auf vollständige PCM-Instanz*
2.7	Inter-case dependencies

Es können jedoch nicht alle für einen IEEE 829-Testfall nötigen Elemente mithilfe des PCMs spezifiziert werden. Dies betrifft vor allem die Vorbedingungen des Testfalls sowie die erwarteten Ausgaben. Diese beiden Bereiche können nicht mithilfe des PCM ausgedrückt werden. Derzeit ist es nicht vorgesehen, die Infrastruktur für die Testumgebung außerhalb des zu testenden Systems

in einem Palladio-basierten Testfall zu erfassen. Dies könnte mittels einer zweiten PCM-Instanz realisiert werden, die die Testumgebung modelliert. Zudem können informelle Teile, wie die Ziele eines Tests, auch nicht durch das PCM ausgedrückt werden. Die informellen Teile enthalten Hinweise für die Tester, die allerdings auch nicht unbedingt formalisiert werden müssen.

4.1.2 JUnit-Testskript-Generator

Der JUnit-Testskript-Generator ist dazu in der Lage, ein JUnit-Testskript aus der unterliegenden PCM-Instanz eines Palladio-basierten Testfalls zu generieren. Dies trägt dazu bei, den Testfallentwurf in einen konkreten ausführbaren Testfall zu überführen. Das generierte Testskript kann mithilfe des JUnit-Frameworks [3] ausgeführt werden, nachdem es von den Testern vervollständigt wurde. Das Testskript enthält den Testtreiber, der das zu testende System von außen mit der spezifizierten Anfragelast ansteuert. Um das JUnit-Testskript zu generieren, kann eine prototypische Implementierung des Testskriptgenerators [25] genutzt werden. Die primäre Grundlage des Generators ist die Übersetzung des Verwendungsmodells, in dem die Abfolge und Häufigkeit der Anfragen an das System spezifiziert sind. Zusätzlich werden Informationen aus dem Komponenten-Repository betrachtet, die über die Systemschnittstellen Auskunft geben. Im Folgenden wird erläutert, wie die PCM-Modelle, insbesondere das Verwendungsmodell, auf das JUnit-Testskript abgebildet werden.

Die Zuordnung der relevanten Elemente des Verwendungsmodells zu den Elementen des JUnit-Testskripts ist in Abbildung 4.2 zu sehen. Auf der linken Seite ist der Auszug des PCM-Metamodells dargestellt. Dabei besteht ein Verwendungsmodell (bzw. `UsageModel`) zunächst aus verschiedenen Szenarien, den `UsageScenario`-Elementen, die jeweils einen bestimmten Verwendungskontrollfluss, das `ScenarioBehaviour`, beinhalten. Ein `ScenarioBehaviour` reiht verschiedene Aktionen, die `AbstractUserAction`-Elemente, in einen Kontrollfluss ein. Eine Aktion kann ein Systemaufruf, eine Wartezeit, eine bedingte Anweisung oder eine Schleife sein. Dabei können bedingte Anweisungen und Schleifen wiederum einen oder mehrere Kontrollflüsse (also `ScenarioBehaviour`) umfassen. Das `Workload`-Element eines Szenarios steuert dabei die Wiederholungsrate und die Nebenläufigkeit des Szenarios. Schließlich werden Instanzen von stochastischen Ausdrücken, den `PcmRandomVariable`-Elementen, an verschiedenen Stellen des Verwendungsmodells zum Einsatz gebracht.

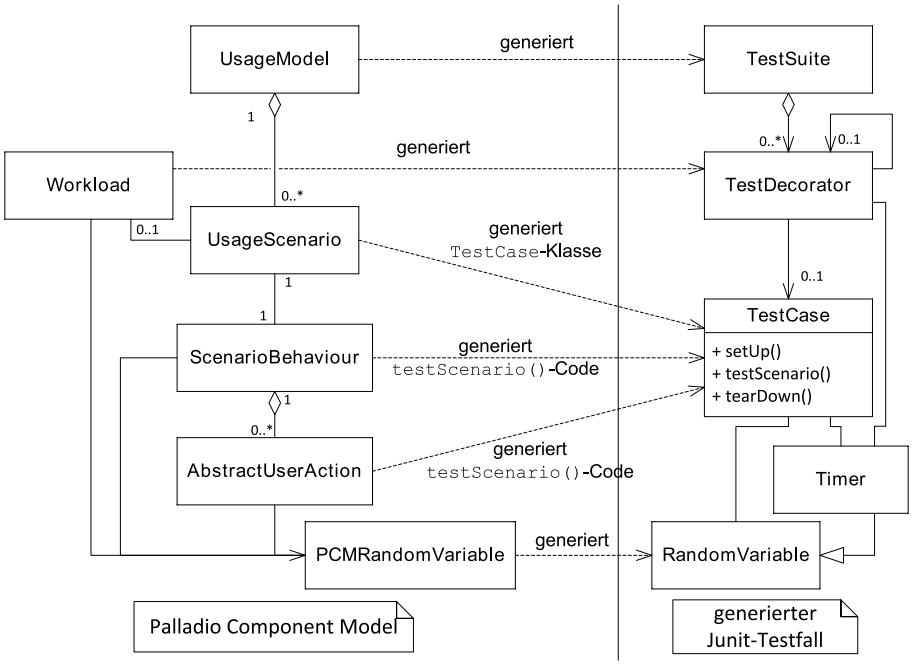

Abbildung 4.2: Zuordnung PCM-Metamodell auf der linken Seite zum generierten JUnit-Testskript auf der rechten Seite (nach Fischer [25]).

Abbildung 4.2 zeigt auf der rechten Seite den Aufbau eines JUnit-Performance-Tests. Das oberste Element ist die Klasse `TestSuite`, die die beteiligten Verwendungskontrollflüsse initialisiert und ausführt. Die `TestSuite` korrespondiert zum Verwendungsmodell (`UsageModel`) aus dem PCM. Ein Verwendungskontrollfluss wird von einem `TestCase`-Element repräsentiert und damit korrespondiert dieses Element zum `UsageScenario`-Element. Dabei kann jedes `TestCase`-Element von einem `TestDecorator`-Element gekapselt werden, das dafür verantwortlich ist, den Verwendungskontrollfluss in einer Schleife oder parallel auszuführen. `TestDecorator` realisieren damit das `Workload`-Element. Die Schritte des Testfalls werden in der Methode `testScenario()` des `TestCase`-Elements ausgeführt. Der Kontrollfluss dieser Methode korrespondiert dabei zu den Aktionen des `UsageScenario`-Elements. Auf PCM-Seite sind die verschiedenen Aktionen dabei als Unterklassen von `AbstractUserAction` modelliert. Auf JUnit-Seite werden Schleifen und bedingte Anweisungen in Standard-Java-Syntax generiert. Ein Systemaufruf wird als Java-Schnittstellenaufruf generiert. Das Initialisieren der Schnittstelle innerhalb des Testskripts muss der Tester manuell vornehmen. Die Deklaration der Variable, die die Schnittstelle im Testskript repräsentiert, wird jedoch generiert. Der Name der Systemschnittstelle muss auf PCM-Seite aus dem Komponenten-Repository übernommen werden, da der Schnittstellenname nicht direkt als Attribut im `EntryLevelSystemCall`-Element des Verwendungsmodells spezifiziert ist. Abbildung 4.3 zeigt die Verbindung zwischen dem `EntryLevelSystemCall`-Element im Verwendungsmodell und dem `ProvidedInterface`-Element aus dem Komponenten-Repository. Der Schnittstellenname aus dem Komponenten-Repository korrespondiert häufig mit dem Namen der Implementierung. Falls dies nicht der Fall ist, kann der Implementierungsname durch eine entsprechende Angabe in einem `Note`-Element im Komponenten-Repository-Diagramm hinzugefügt werden. Die Angabe im `Note`-Element wird dabei dem Schnittstellennamen im Komponenten-Repository vorgezogen. Diese Art von PCM-Erweiterung ließe sich in Zukunft alternativ auch mit sogenannten „Profiles“ [42] realisieren.

Die vorgestellte Generierung des JUnit-Testskripts hat klar definierte Grenzen. Zunächst werden weder die Testdaten noch die erwarteten Ergebnisse mithilfe des PCMs modelliert, so dass diese nicht in der Generierung berücksichtigt werden. Somit müssen sowohl die Testdaten als auch die Überprüfung der erwarteten Ergebnisse vom Tester beigesteuert werden. Ähnlich ist es mit dem Herstellen der Vorbedingung des Testfalls. Auch dies wird nicht im PCM spezifiziert und wird deshalb nicht bei der Generierung des JUnit-Testskripts (in der dafür vorgesehenen Methode `setup()`) berücksichtigt.

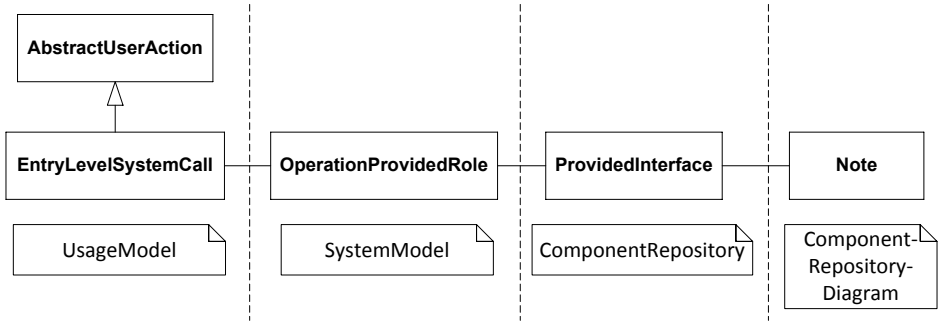

Abbildung 4.3: Die Verbindung zwischen dem `EntryLevelSystemCall` im Verwendungsmodell und der zugehörigen Schnittstellenspezfikation, also dem `ProvidedInterface`-Element im Komponenten-Repository

Die Vorbedingung ließe sich aber auch durch eine initiale Folge von Operationsaufrufen automatisch herstellen. Diese Technik, bei der eine bestimmte Reihenfolge von Operationsaufrufen aufgrund der mittels „visuellen Kontrakten“ angegebenen Vor- und Nachbedingungen der Operationen berechnet wird, wurde von Güldali [27, 21] vorgestellt. Schließlich zieht es die Wahl des JUnit-Frameworks nach sich, dass sich die generierten Testskripte insbesondere dazu eignen mit Java implementierte Systeme zu testen. Allerdings existieren in einer Vielzahl von anderen Sprachen zu JUnit äquivalente Testwerkzeuge, auf die sich das Konzept nahezu eins zu eins übertragen lassen sollte. Auch die Portierung auf ein komplett anderes Testwerkzeug sollte sich mithilfe der exemplarischen Interpretation des PCMs für Performance-Testfälle sowie der Beispielarchitektur für Testskripte leicht bewerkstelligen lassen.

Das in der Palladio-Workbench enthaltene ProtoCom [5, 4, 47] leistet ähnliches wie der hier beschriebene JUnit-Testskript-Generator. ProtoCom generiert einen Performance-Prototyp aufgrund einer gegebenen PCM-Instanz. Dies kann dazu genutzt werden, um genauere Performance-Vorhersagen als mit der Simulation zu erhalten. Der generierte Performance-Prototyp enthält auch einen Lastgenerator, der vergleichbar zu dem vorgestellten JUnit-Testskript ist. Viele Bereiche der beiden Lastgeneratoren verhalten sich identisch, da die Interpretation des PCM natürlich korrekt sein muss. Das betrifft zum Beispiel die Abbildung der Workload-Elemente und der stochastischen Ausdrücke. Allerdings zeigt sich bei den stochastischen Ausdrücken auch, dass der JUnit-Testskript-Generator nicht alle Bereich des PCM abdeckt, da es sich hierbei lediglich um eine prototypische Implementierung handelt,

werden „IntPMF“-Ausdrücke (Wahrscheinlichkeitsfunktion für ganzzahlige Werte) nicht berücksichtigt. Die Unterschiede liegen überdies auch in der jeweiligen Zielrichtung begründet. Der von ProtoCom generierte Prototyp ist in sich geschlossen, so dass im Lastgenerator ein funktionstüchtiger Aufruf des generierten Prototyps enthalten ist. Dabei lässt sich dieser Zugriff nur für das echte System weiterverwenden, wenn die jeweils gewählte Zugriffstechnologie, also RMI bzw. HTTP [40], auch für die spätere Umsetzung eingesetzt wird. Beim JUnit-Testskript-Generator wird ein Aufruf immer als Aufruf einer Java-Methode auf dem angegebenen Interface-Namen realisiert. Welche Technologie letztlich für den Zugriff genutzt wird, ist unerheblich. Allerdings muss der Tester hier auch den Zugriff auf das System eigens implementieren. Zudem geht ProtoCom davon aus, dass ein Test weder eine Vorbereitung, also das Setzen einer Vorbedingung, noch ein nachträgliches Aufräumen benötigt. Während der JUnit-Testskript-Generator weder Quelltext für die Vorbereitung und Aufräumen, also die `setup`- bzw. `tearDown`-Methode, generieren kann, so sind diese Schritte doch wenigstens vorgesehen. Schließlich werden Eingabedaten in ProtoCom innerhalb der Abarbeitung des `UsageScenarios` mittels der Interpretation eines stochastischen Ausdrucks generiert. Der JUnit-Testskript-Generator hat die Generierung von Eingabewerten in eine eigene Klasse ausgelagert, so dass der Tester hier einfach eingreifen kann, um den gewünschten Testdatengenerator bzw. die gewünschte Testdaten-Datenbank anzuschließen.

4.2 Performance-Daten sammeln

In den Unterabschnitten 4.2.1 und 4.2.2 wird beschrieben, wie die Antwortzeitdatenreihen aus dem Performance-Test und der Performance-Vorhersage gesammelt werden. Dieser Prozess ist in Abbildung 4.4 dargestellt. Im oberen Teil ist zu sehen, dass ein Performance-Testfall (ggf. mit Abweichungen) wiederholt wird. Unten sieht man, dass bei der Performance-Vorhersage mithilfe des PCMs ein zum Testfall äquivalentes Szenario simuliert wird, um die Antwortzeitdaten für die Performance-Vorhersage zu erhalten. Die Antwortzeitdaten aus beiden Quellen werden zunächst in das geforderte Format gebracht und dann im Rahmen der weiteren Analyse verglichen (s. Abschnitt 4.3).

4.2.1 Messungen aus dem Performance-Test sammeln

Wie in Abschnitt 3.1 erwähnt wurde, beginnt die Performance-Blame-Analysis mit einem fehlgeschlagenen Testfall. Dieser Testfall wird daraufhin im Rah-

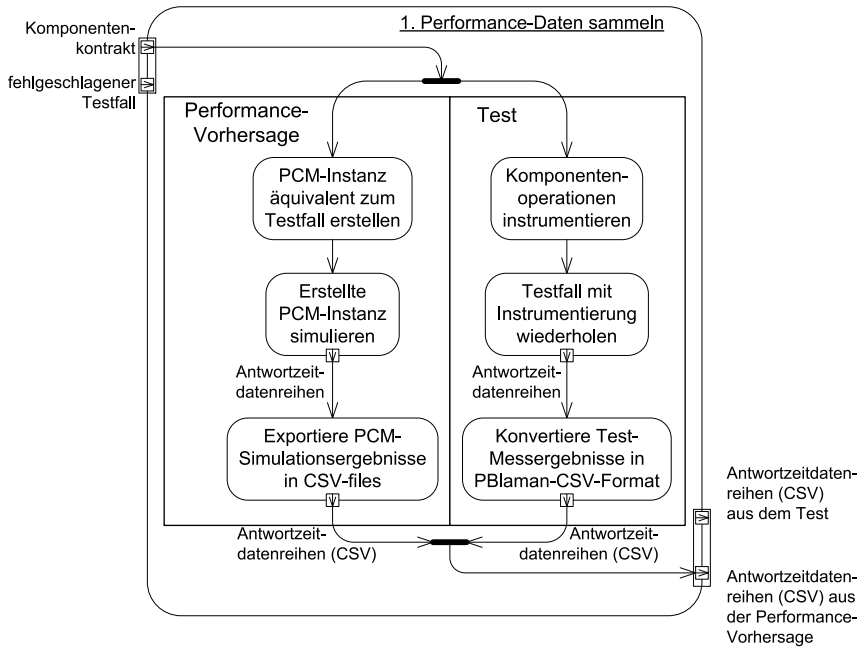

Abbildung 4.4: Der Subprozess zum Sammeln der Antwortzeitdatenreihen. Der interne Datenfluss zeigt die Entstehung der Antwortzeitdatenreihen je Operation während der Testfalldurchführung und der PCM-Simulation.

men des PBlaman-Prozesses wiederholt ausgeführt. Dazu instrumentiert der Systemarchitekt zunächst das komponentenbasierte System (s. Abbildung 4.4), so dass er in der Lage ist, zu jeder Komponentenoperation eine Antwortzeitdatenreihe zu messen. Dann führt der Systemarchitekt den Testfall, wie angesprochen, mit der Instrumentierung erneut aus. Die Instrumentierung muss dabei die richtigen Daten messen. Dies sind (vgl. Abbildung 2.3) der Zeitpunkt der Anfrage, die Antwortzeit, der Stacktrace auf Operationsebene und die Identifikation des Komponentenobjekts. Falls die Instrumentierung die Daten nicht in der geforderten Syntax liefert, sorgt ein Konverter dafür, das von PBlaman geforderte Datenformat mit CSV-Daten (die genaue Definition folgt in Abschnitt 4.5) herzustellen. Der Konverter muss einmal passend für das Messdatenformat der Instrumentierung implementiert werden. Solange sich das Messdatenformat der Instrumentierung nicht ändert, kann der Konverter für diesen und andere Testfälle wiederverwendet werden.

Für das Anwendungsbeispiel aus Abschnitt 1.2 muss der Systemarchitekt eine Antwortzeitdatenreihe mit den erwähnten Daten für jeden Operationsaufruf aus Abbildung 1.3 messen. Um beispielsweise die Nachricht 4 („Hole Infos über Supermarktkette“) zu erfassen, muss die Antwortzeit der Operation `queryEnterpriseById()` in der Komponente `EnterpriseQuery` gemessen werden. Falls alle erforderlichen Antwortzeitdatenreihen schon vorliegen, weil sie z. B. schon beim initialen Ausführen des Testfalls gemessen wurden, kann der Systemarchitekt diese Daten verwenden und das Sammeln der Testdaten überspringen. Die gemessenen Antwortzeitdatenreihen werden im Rahmen der Entscheidungsunterstützung (s. Abschnitt 4.3) weiterverarbeitet.

4.2.2 Messungen aus der Performance-Vorhersage sammeln

Bei PBlaman werden erwartete Antwortzeiten mithilfe der Performance-Vorhersage aus den Komponentenkontrakten hergeleitet. Die Performance-Vorhersage dient als Testorakel. Um die erwarteten Antwortzeitdatenreihen aus der Performance-Vorhersage zu sammeln, muss eine vollständige PCM-Instanz simuliert werden. Der Systemarchitekt konstruiert eine vollständige PCM-Instanz, die den Komponentenkontext genauso wie die Komponenten und das System explizit modelliert (vgl. Abschnitt 2.3), von Grund auf neu oder bearbeitet eine schon bestehende PCM-Instanz (s. Abbildung 4.4). Diese PCM-Instanz muss auch äquivalent zum untersuchten Testfall sein (vgl. Abschnitt 4.1). Die Simulation dieser PCM-Instanz resultiert in einer Antwortzeitdatenreihe für jede am Testfall teilnehmende Komponentenoperation. Die Palladio-Workbench sammelt während der Simulation automatisch die

erforderlichen Daten (s. Unterabschnitt 2.3.6). Die Datenreihen müssen im Anschluss an die Simulation zur weiteren Analyse noch jeweils manuell in eine CSV-Datei exportiert werden. Dies erfordert in der Palladio-Workbench jeweils nur wenige Klicks. Für das Anwendungsbeispiel aus Abschnitt 1.2 müsste eine Antwortzeitdatenreihe für jeden Aufruf einer Komponentenoperation in Abbildung 1.3 vorliegen, genau wie beim Sammeln der Testdaten. Falls die benötigten Daten aus der Aktivität QoS-Analyse schon vorliegen, kann der Systemarchitekt diese Daten verwenden und das Sammeln der Performance-Vorhersage-Daten überspringen.

4.3 Entscheidung unterstützen

Die Entscheidungsunterstützung erhält die Antwortzeitdatenreihen im CSV-Format aus dem Test und der Performance-Vorhersage als Eingabe. Sie resultiert in Artefakten, die dem Systemarchitekten dabei helfen, zu entscheiden, welche Komponentenoperationen für den untersuchten Fehler zu beschuldigen sind. Im Folgenden wird dargestellt, welche Entscheidungskriterien für eine automatisierte Entscheidungsunterstützung geeignet sind. Unterabschnitt 4.3.1 evaluiert statistische Kriterien, die entscheiden, ob eine Antwortzeitdatenreihe aus dem Test größere Werte hat als die zugehörige Datenreihe aus der Performance-Vorhersage. In Unterabschnitt 4.3.2 werden die geeignetsten Visualisierungen aus Unterabschnitt 3.3.4 für die ausgewählten Kriterien ermittelt. Zuletzt wird der Subprozess vorgestellt (s. Unterabschnitt 4.3.3), der die ausgewählten statischen Kriterien und die dazu passenden Visualisierungen berechnet und in entsprechenden Ergebnisartefakten präsentiert.

4.3.1 Automatisiert auswertbare Entscheidungskriterien

PBlaman entscheidet, ob eine Komponentenoperation beschuldigt werden muss, indem bewertet wird, ob die Antwortzeitdatenreihe aus dem Test die insgesamt größeren Werte beinhaltet. Um dies zu entscheiden, werden in diesem Abschnitt automatisiert auswertbare Entscheidungskriterien vorgestellt, die diesen Sachverhalt im Rahmen einer Heuristik bewerten können. Dazu muss zunächst geklärt werden, welche Entscheidungskriterien überhaupt automatisiert entscheiden können, welche Komponentenoperationen beschuldigt werden sollen. Zudem muss untersucht werden, welches dieser Kriterien am robustesten ist, also in den meisten Fällen eine korrekte Einschätzung liefert. Aus diesen beiden Punkten lassen sich folgende Anforderungen an die Entscheidungskriterien formulieren:

1. Das Entscheidungskriterium muss sich auf Antwortzeitdatenreihen anwenden lassen.
2. Das Entscheidungskriterium muss automatisiert entscheiden können, ob eine Datenreihe höhere Werte beinhaltet als eine andere.
3. Das Entscheidungskriterium trifft so oft wie möglich korrekte Entscheidungen.

Während sich die beiden ersten Anforderungen direkt aus der Definition des Entscheidungskriteriums ableiten lassen, sollte die dritte Anforderung empirisch überprüft werden. Im Folgenden werden nur Kriterien betrachtet, bei denen die ersten beiden Anforderungen erfüllt sind. Bei diesen Kriterien wird untersucht, wie gut die dritte Anforderung erfüllt ist. Dazu werden 27 Datenreihenpaare[1] aus unserem initialen Experiment mit dem CoCoME (s. Abschnitt 5.1) ausgewertet.[2] Für jedes dieser Paare wurde zunächst manuell untersucht, ob die Testdatenreihe größere oder kleinere Werte aufweist als die Datenreihe aus der Performance-Vorhersage. Diese Analyse wurde aufgrund von Box-Whisker-Plots, wie in Abbildung 3.13, und Histogrammen, wie in Abbildung 4.5, vorgenommen. Jedes Entscheidungskriterium wurde mit dieser manuellen Entscheidung abgeglichen. Je häufiger das Entscheidungskriterium mit den manuellen Entscheidungen übereinstimmt, desto besser eignet es sich als Heuristik für die Entscheidungsunterstützung. Das Ergebnis dieses Auswahlverfahrens ist also eine Heuristik, die für CoCoME gut funktioniert und sich hoffentlich auch auf ähnliche Anwendungen generalisieren lässt.

Zunächst wurden Punktschätzer von Lageparametern als Heuristik untersucht. Punktschätzer von Lageparametern eignen sich als Heuristik, da sie Datenreihen mit einem einzelnen Wert charakterisieren. Die beiden Lageparameter für die Testdatenreihe und die Datenreihe aus der Performance-Vorhersage werden verglichen, um festzustellen, welche der Datenreihen die größeren Werte aufweist. Die bekanntesten Lageparameter [67] sind Minimum, Maximum, Mittelwert (arithmetisches Mittel) und die Quartile. Die Quartile umfassen das 25%-Quartil, den Median und das 75%-Quartil. Es wurden die Punktschätzer aller angeführten Lageparameter mit den ausgewählten

[1] In vorherigen Arbeiten [12, 13] war von lediglich 20 Datenreihenpaaren die Rede. Dort wurde nur ein Teil der insgesamt 27 Datenreihenpaare bewertet, da teils derselbe Operationsaufruf doppelt bzw. nahezu identische Datenreihen für dieselbe Operation in verschiedenen Aufrufkontexten vorkamen. Im Rückblick war die Auswahl jedoch unsystematisch, so dass hier alle 27 Datenreihenpaare mit einbezogen werden. Am Ergebnis der Analyse ändert das jedoch nichts.

[2] Die Daten zur Auswahl der Entscheidungsheuristik finden sich auf der Begleitwebseite der Dissertation:
http://homepages.uni-paderborn.de/bruesie/dissertation/index_de.html.

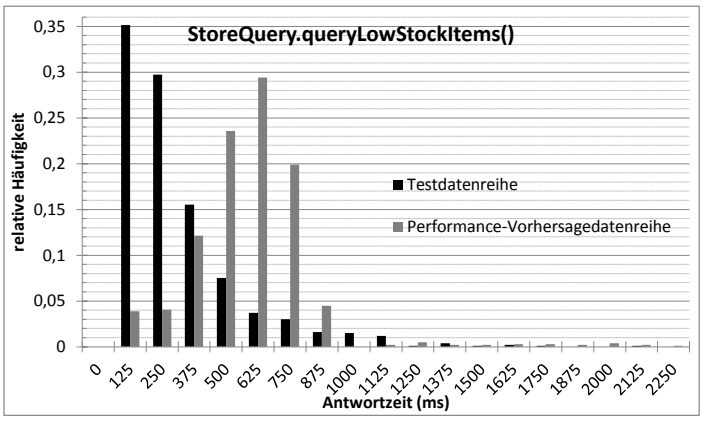

Abbildung 4.5: Bildet ein Histogramm ab, dessen Klassen jeweils 125 ms breit sind.

Datenreihenpaaren getestet. Das Ergebnis ist in Tabelle 4.2 dargestellt. Das Minimum und das Maximum haben dabei am schlechtesten abgeschnitten. Das Minimum wich 13-mal und das Maximum viermal von den Referenzentscheidungen ab. Der Mittelwert verhielt sich mit gerade einmal zwei falschen Entscheidungen besser. Eine falsche Entscheidung wurde dabei von einer großen Zahl von Ausreißern nahe dem Maximum verursacht. Solche Fälle können von den Quartilen besser gehandhabt werden. Dementsprechend wichen die Quartile, also 25%-Quartile, Median und 75%-Quartil, jeweils nur einmal von den Referenzentscheidungen ab.

Tabelle 4.2: Die Tabelle zeigt das Abschneiden der Entscheidungskriterien beim Bewerten der 27 Datenreihenpaare aus dem initialen CoCoME-Test. Die Tabelle zeigt in den Spalten korrekte bzw. inkorrekte Entscheidungen sowie wenn ein Kriterium fälschlicherweise unentschieden war.

Kriterium	korrekt	unentschieden	inkorrekt
arithmetisches Mittel	25	0	2
Maximum	23	0	4
75% Quartil	26	0	1
Median	26	0	1
25% Quartil	26	0	1
Minimum	14	0	13
KS-Test	25	2	0

Einige der Lageparameterpunktschätzer eignen sich schon gut als Entscheidungsheuristik. Der Bereich des statistischen Testens ist aber ebenso vielversprechend. Besonders der Kolmogoroff-Smirnow-Test (KS-Test) [66] eignet sich für die Analyse. Das liegt daran, dass je eine Variante des KS-Tests entscheiden kann, ob eine Datenreihe größere bzw. niedrigere Werte als eine andere hat. Zudem ist der KS-Test unabhängig von der Verteilung der Antwortzeiten. Dafür sollte aber die Stichprobe eine Mindestanzahl von Messwerten aufweisen, um ein zuverlässiges Ergebnis zu erhalten. Im Rahmen dieser Arbeit wird angenommen, dass die zu untersuchenden Testfälle lang genug laufen, so dass dieses Problem keine Rolle spielt. Andere statistische Tests setzen eine bestimmte Werteverteilung voraus. Z. B. nimmt der Zweistichproben-t-Test [18] an, dass die Daten annähernd normalverteilt sind. Eine ähnliche Argumentation findet sich auch bei Sambasivan et al. [63], die auch den KS-Test verwenden, um Datenreihen miteinander zu vergleichen.

Für dieses Auswahlverfahren wurde die Implementierung der KS-Test-Varianten aus dem Statistiksystem „R“ [55] verwendet. Damit werden zwei Nullhypothesen überprüft: 1.) Die Testdatenreihe hat *niedrigere* Werte als die Datenreihe aus der Performance-Vorhersage. 2.) Die Testdatenreihe hat *höhere* Werte als die Datenreihe aus der Performance-Vorhersage. In klaren Fällen kann genau eine dieser beiden Hypothesen zurückgewiesen werden. Für sehr ähnliche Datenreihen zeigen häufig beide Tests das gleiche Ergebnis, so dass das KS-Test-Kriterium unentschieden ist. Damit unterscheidet dieses Kriterium drei Fälle, nämlich entweder hat die Testdatenreihe niedrigere bzw. höhere Werte als die Datenreihe aus der Performance-Vorhersage oder aber der KS-Test ist unentschieden. Dies ist ein Vorteil gegenüber den Lageparameterpunktschätzern. Das hat sich auch im Auswahlverfahren widergespiegelt, bei dem der KS-Test keine Entscheidungen getroffen hat, die den Referenzentscheidungen entgegenstehen (vgl. Tabelle 4.2). Allerdings war der KS-Test für zwei Datenreihenpaare unentschieden, bei denen eine Entscheidung möglich gewesen wäre. In diesen Fällen wurden beide Nullhypothesen widerlegt. So schneidet der KS-Test knapp besser ab, als die Quartile.

Insgesamt sind die Quartile, also das 25%-Quartil, der Median und das 75%-Quartil, zuverlässige Kriterien, die sich als Entscheidungsheuristik eignen. Knapp übertroffen werden diese Kriterien vom KS-Test, der in unserem Auswahlverfahren besser abgeschnitten hat, da er auch auf knappe Fälle hinweisen kann, die manuell untersucht werden sollten. Somit wird der KS-Test als primäre und die Quartile als sekundäre Entscheidungsheuristik verwendet. Im nächsten Abschnitt 4.3.2 wird untersucht, wie diese Entscheidungskriterien visualisiert werden können.

4.3.2 Entscheidungskriterien visualisieren

Wie schon in Unterabschnitt 3.4 dargestellt, verfolgen die Visualisierungen verschiedene Ziele. Zum einen gibt es Übersichten, die alle Operationsaufrufe beziehungsweise die Operationsaufrufe zu einem Systemaufruf darstellen. Zwischen der Darstellung aller Operationsaufrufe und der Darstellung der Operationsaufrufe zu einem Systemaufruf besteht konzeptionell kein Unterschied. In beiden Fällen wird eine Aufrufhierarchie in einer Übersicht dargestellt, lediglich mit einem jeweils unterschiedlichen Startpunkt. Zum anderen gibt es Detaildarstellungen, die zwei Operationsdatenreihen vergleichend darstellen. Dabei kann eine Detaildarstellung die Übersicht nicht ersetzen und umgekehrt. Daher wird hier eine Darstellung für beide Darstellungsarten gesucht. In diesen Darstellungen sollen die in Unterabschnitt 4.3.2 ausgewählten Kriterien, also der KS-Test und die Quartile, visualisiert werden.

Das primäre Entscheidungskriterium ist der KS-Test (vgl. Unterabschnitt 4.3.2). Der Vergleich der Datenreihen mithilfe des KS-Tests unterscheidet drei Entscheidungen: 1.) die Testdatenreihe hat niedrigere Werte als die Datenreihe aus der Performance-Vorhersage; 2.) die Testdatenreihe hat höhere Werte als die Datenreihe aus der Performance-Vorhersage; und 3.) der Test ist unentschieden, da die Datenreihen sehr ähnlich sind. Diese drei Ergebniskategorien stehen auch für drei Schweregrade bei der Beschuldigung von Komponentenoperationen. Somit bietet sich hier eine Visualisierung mit Ampelfarben an. Dabei stehen grün und rot dafür, dass die Testdatenreihe die niedrigeren bzw. höheren Werte aufweist. Bei Gelb ist der KS-Test unentschieden. Dieses Farbschema lässt sich leicht in andere Visualisierungen integrieren. Wenn das Farbschema in eine Visualisierung integriert wird, enthält die Visualisierung eine Entscheidung, die gleichzeitig als Interpretationshilfe für die Performance-Blame-Analysis dient. Mit dem Farbschema erfüllt eine Darstellung somit die Visualisierungsanforderungen V5 und V6.

Für die Übersichtsdarstellung wird in PBlaman ein modifizierter Flame-Graph [28, 29] eingesetzt. Der Flame-Graph schneidet wegen seiner Übersichtlichkeit besser ab, als die Profiler-Darstellungen [19, 20] und j2eeprof [39] (vgl. Unterabschnitt 3.4). Ähnliches gilt für die Gleisplanvisualisierung in Magpie [1] sowie die Gleisplanvisualisierungen von Sambasivan et al. [63] und von Reynolds et al. [63]. Die Gleisplanvisualisierungen fokussieren, dabei auch eher auf die Darstellung von Hardwareknoten und Ressourcen als auf die Darstellung von Komponenten und Komponentenoperationen. Da sich diese Arbeit auf die Betrachtung der Antwortzeiten von Komponentenoperationen

beschränkt, ist die Darstellung von Komponenten und Komponentenoperationen wichtiger als die Darstellung von Hardwareknoten und Ressourcen. Zuletzt ließe sich auch eine zu RanCorr [48] ähnliche Darstellung wählen. Die Darstellung ist mit der Darstellung verschiedener Abstraktionsebenen sehr ausgereift, kann aber bei der Übersichtlichkeit nicht mit dem Flame-Graphen mithalten. Zudem hat Gregg dargestellt [29], wie der Flame-Graph für Messergebnisse aus verschiedenen Prozessen erweitert werden kann. Diese Technik lässt sich nutzen, um Komponenten und Hardwareknoten darzustellen. Wenn man dieses Verbesserungspotenzial berücksichtigt, ist der Flame-Graph am geeignetsten.

In PBlaman wird ein modifizierter Flame-Graph, der sogenannte Blame-Graph, als Übersichtsdarstellung genutzt. Da Farben bei den Flame-Graphen keinerlei Bedeutung zugeordnet ist, lassen sich anstelle der Flame-Graph-Farben die zuvor besprochenen Ampelfarben für die KS-Test-Ergebnisse verwenden. Ein Beispiel für einen Blame-Graphen ist in Abbildung 4.6 zu sehen. Der Blame-Graph stellt einen Aufrufkontextbaum dar und zeigt einen Operationsaufruf in jeder Box. Die Boxbreite richtet sich dabei nach dem Median der Antwortzeitdatenreihe aus dem Test (im Folgenden kurz Medianantwortzeit), da sich der Median während des Kriterienauswahlverfahrens (vgl. Unterabschnitt 4.3.1) als aussagekräftig erwiesen hat und auch da der Median eine gute Abschätzung für eine übliche Antwortzeit der jeweiligen Operation liefert. Zudem wird jede Box nach dem Ergebnis des KS-Tests für die beiden zugehörigen Datenreihen eingefärbt. Zu beschuldigende Operationen werden mit roten Trapezen dargestellt, bei denen die Basis unten ist. Im Folgenden werden solche Trapeze „Fußtrapez“ genannt. Operationen, bei denen eine Beschuldigung nicht infrage kommt, werden durch grüne Trapeze, bei denen die Basis oben ist, dargestellt. Solche Trapeze werden in dieser Arbeit „Kopftrapeze“ genannt. Wenn der KS-Test kein eindeutiges Ergebnis ausgibt, werden die Operationen als gelbes Hexagon abgebildet. Die Darstellung mit den Trapezen und Hexagonen verbessert die Lesbarkeit. Zum einen lassen sich die Elemente auch ohne die Farben erkennen, zum anderen wird so besser sichtbar, wo ein Element aufhört und ein Angrenzendes beginnt. Um die Formen der Trapeze und Hexagons einzuhalten, wird jedes Element mit einer minimalen Breite dargestellt. Dies kann die Darstellung verfälschen, da die Elemente mit minimaler Breite überproportional breit dargestellt sein können. Es wird im Rahmen dieser Arbeit jedoch davon ausgegangen, dass die Anzahl der Elemente mit minimaler Breite in einer Zeile überschaubar ist, so dass die Darstellung nur wenig verfälscht wird und die Interpretation der Darstellung nicht beeinflusst. Eine minimale Breite ist auch bei einer Darstellung, die ausschließlich Rechtecke nutzt, wie den Flame-Graphen,

nötig, da sonst einige Elemente überhaupt nicht sichtbar wären. Der Blame-Graph ermöglicht es, mithilfe der geometrischen bzw. farblichen Darstellung direkt die beschuldigten Operationen ausfindig zu machen. Diese Operationen können im nächsten Schritt priorisiert nach Aufrufhierarchie, also der vertikalen Boxenanordnung, bzw. der Medianantwortzeit, also der Boxbreite, näher untersucht werden.

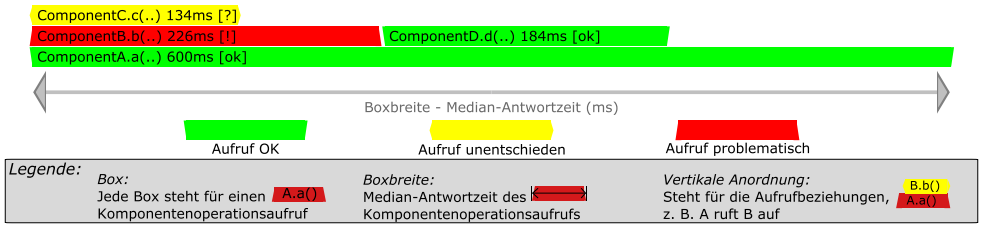

Abbildung 4.6: Beispiel-Blame-Graph: modifizierter Flame-Graph, der Median und KS-Test-Resultat darstellt

Mit dem Blame-Graphen verfügt PBlaman über eine Übersichtsdarstellung, die die KS-Test-Ergebnisse hinreichend gut visualisiert. Für die Detaildarstellung bietet sich der Box-Whisker-Plot (vgl. Abbildung 3.13) an, wie es in Unterabschnitt 3.3.4 vorgestellt wird. Diese Darstellung eignet sich besonders, da hier die Quartile, das sekundäre Entscheidungskriterium (vgl. Unterabschnitt 4.3.2), aber auch das Minimum und das Maximum dargestellt werden. Der Box-Whisker-Plot stellt die Datenreihe einer Komponentenoperation aus dem Test der zugehörigen Datenreihe aus der Performance-Vorhersage gegenüber. Wenn im Box-Whisker-Plot die Box einer Datenreihe vollständig über der Box der anderen Datenreihe liegt, so beinhaltet diese Datenreihe wenigstens gleich große und wahrscheinlich sogar die höheren Werte.[3] Diese Darstellung der beiden Datenreihen in einem Box-Whisker-Plot wird in den sogenannten *Performance-Report* übernommen. Der Performance-Report enthält zusätzlich zu den Box-Whisker-Plots die genauen numerischen Ergebnisse je Operationsaufruf. Dies soll dabei helfen, die Darstellungen nachvollziehbar zu machen (Anforderung A13). Diese numerischen Ergebnisse umfassen

[3] Im Folgenden wird angenommen, dass das 25%-Quartil aus einer Datenreihe wenigstens gleich dem 75%-Quartil aus der anderen Datenreihe ist und jede Datenreihe wenigstens 25 Messwerte beinhaltet. Daraus folgt, dass eine KS-Test-Variante die Datenlage automatisch so bewertet, dass die Datenreihe mit der höher gelegenen Box auch insgesamt die höheren Werte beinhaltet. Da die Werte außerhalb der Boxen beliebig verteilt sein können, lässt sich das Ergebnis der anderen KS-Test-Variante nicht auf ähnliche Weise eingrenzen. Also würde der KS-Test bei dieser Datenlage wenigstens unentschieden sein oder sogar entscheiden, dass die Datenreihe mit der höher gelegenen Box die insgesamt größeren Werte beinhaltet.

alle Lageparameter, die im Auswahlverfahren (s. Unterabschnitt 4.3.1) berücksichtigt wurden, sowie die KS-Test-Ergebnisse. Somit sind eine Übersichts- und eine Detaildarstellung verfügbar, die die KS-Test-Ergebnisse und die Quartile visualisieren. Der Systemarchitekt nutzt den Blame-Graphen und den Performance-Report, um im Schritt Ergebnisinterpretation (s. Unterabschnitt 4.4) zu einer Blame-Entscheidung zu kommen.

4.3.3 Vorgehen zur Entscheidungsunterstützung

Es wird nun der Subprozess zur Entscheidungsunterstützung vorgestellt, der die zuvor ausgewählte Entscheidungsheuristik berechnet und die entwickelte Visualisierung produziert. Wie in Abbildung 4.7 zu sehen ist, werden die meisten Tätigkeiten automatisch durchgeführt. Lediglich die Aktionen auf der linken Seite werden vom Systemarchitekten ausgeführt. In der Realisierung der automatischen Aufgaben kam ein Python-Skript zum Einsatz, dass das Statistiksystem R [55] verwendet (s. Abschnitt 4.5).

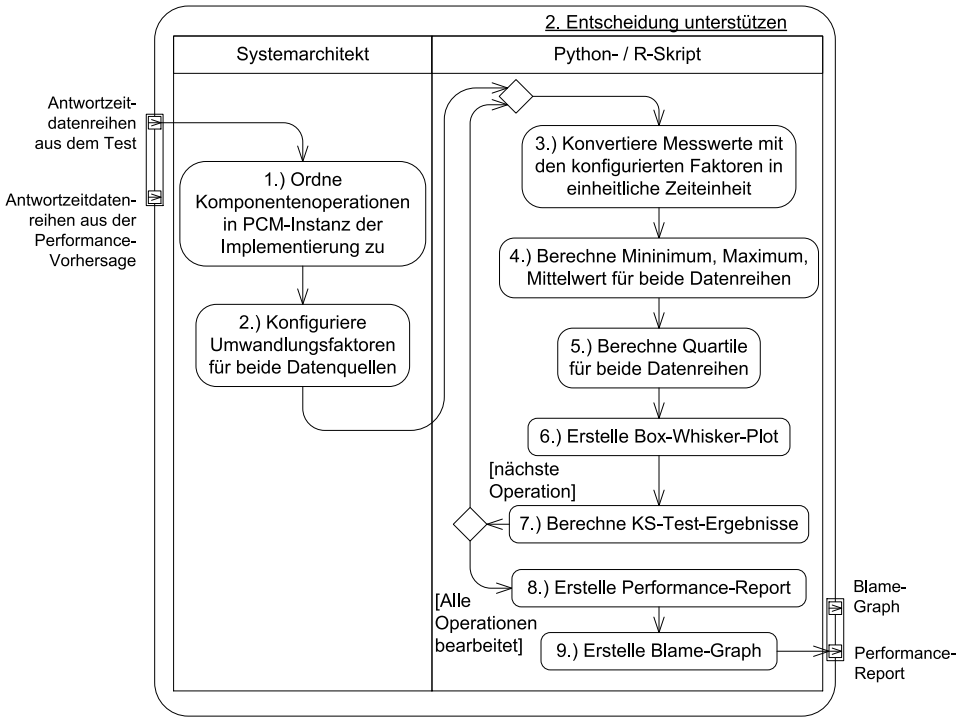

Abbildung 4.7: Der Subprozess in dem die Artefakte zur Entscheidungsunterstützung im Rahmen von PBlaman erstellt werden.

Im Rahmen der manuellen Vorbereitung muss der Systemarchitekt zunächst die Komponentenoperationen den PCM-Modellelementen zuordnen (Schritt 1 in Abbildung 4.7). Dabei ist die Zuordnung abhängig von den Aufrufbeziehungen. Es wird hier also jedes Element im Aufrufkontextbaum zu einem PCM-Modellelement zugeordnet. In Schritt 2 muss der Systemarchitekt Umrechnungsfaktoren für die gemessenen Zeitspannen in den Testdatenreihen und in den Datenreihen aus der Performance-Vorhersage angeben. Dies ist erforderlich, um die Zeitangaben aus beiden Quellen auf eine gemeinsame Zeiteinheit (zum Beispiel Millisekunden) zu normalisieren.

Bei der Zuordnung werden die Antwortzeitdatenreihen aus dem Test denen aus der PCM-Simulation zugeordnet. Wie schon erwähnt ist diese Zuordnung abhängig von den Aufrufbeziehungen, da PBlaman eine Datenreihe für jedes Vorkommen einer Operation im Aufrufkontextbaum des Szenarios betrachtet. Jeder Knoten des Aufrufkontextbaums unterscheidet einen bestimmten Operationsaufruf eines bestimmten Komponentenobjekts. Bei den Testdaten lassen sich anhand der Identifikation des Komponentenobjekts und der Stacktraces leicht entsprechende Datenreihen bilden (vgl. Unterabschnitt 4.2.1). Dies ist bei den Daten aus der PCM-Simulation standardmäßig nicht möglich. Die Daten aus der PCM-Simulation stammen aus der Palladio-Workbench. Die Palladio-Workbench erfasst eine Datenreihe für jeden der in den Service-Effect-Specifications (SEFFs) modellierten Aufrufe (vgl. Unterabschnitt 2.3.7). Ein Aufrufobjekt legt dabei lediglich fest, dass aus einer Komponentenoperation eine bestimmte Operation aus einer bestimmten Schnittstelle aufgerufen wird. Zusätzlich kann über das Verteilungsdiagramm und das Systemdiagramm noch ermittelt werden, welches Komponentenobjekt welches andere Komponentenobjekt aufruft. Mit diesen Informationen kann die Zuordnung nur bei einer Aufrufbeziehung erfolgen, die einzigartig im Aufrufkontextbaum ist. Am Aufrufkontextbaum illustriert, wird hier ein Aufrufpaar (aus der Simulation) einem vollständigen Aufrufpfad (auf Testseite) zugeordnet. Auf dieser Grundlage lassen sich einige Operationsaufrufe zuordnen aber nicht alle. Dies ist keine konzeptionelle Schwäche, sondern spiegelt die Situation aufgrund der aktuellen PCM-Workbench-Implementierung wider. Allerdings ist die PCM-Workbench, genauer gesagt das `SensorFramework`, mit dem die Messdaten gesammelt werden, erweiterbar. Diese Lücke ließe sich dadurch schließen, dass das `SensorFramework` mit einem Messfühler erweitert wird, der bei jedem Operationsaufruf den Stacktrace inklusive der aufgerufenen Komponentenobjekte sowie dessen Antwortzeit ermittelt. Diese Daten ließen sich wie gehabt ausgeben und müssten lediglich vorverarbeitet werden, so dass für jeden unterschiedlichen Stacktrace eine Datei entsteht, die entsprechend zugeordnet werden kann.

Nachdem der Systemarchitekt die Konfiguration für das Skript festgelegt hat, werden die Entscheidungsheuristiken berechnet und visualisiert. Dabei werden zunächst die Werte in der aktuell untersuchten Datenreihe mithilfe der vom Systemarchitekten angegebenen Faktoren konvertiert (Schritt 3 in Abbildung 4.7). Danach werden die Werte für die Lageparameter berechnet (Schritte 4 und 5). Auf dieser Grundlage wir der Box-Whisker-Plot erstellt (Schritt 6). Nun werden noch die beiden KS-Tests ausgeführt (Schritt 7). Die Schritte 5 bis 9 werden jeweils für jedes Datenreihenpaar wiederholt. Im Anschluss wird der Performance-Report zusammengestellt (Schritt 8). Dabei werden für jedes Datenreihenpaar zwei Seiten in den Performance-Report eingefügt, die Eine enthält den Box-Whisker-Plot und die Andere die numerischen Ergebnisse der Lageparameter und des KS-Tests. Zuletzt werden die KS-Test-Ergebnisse und die Medianantwortzeiten zu einem Blame-Graphen verarbeitet (Schritt 9). Der Systemarchitekt kann die beiden hier erstellten Artefakte, also den Blame-Graphen und den Performance-Report, im nächsten Schritt interpretieren und untersuchen, um schließlich zu seiner Entscheidung zu kommen.

4.4 Ergebnis interpretieren

Der Subprozess zum Interpretieren der Ergebnisse basiert auf den Entscheidungsunterstützungsartefakten Blame-Graph und Performance-Report. Dieser Subprozess ist in Abbildung 4.8 zu sehen. Der Blame-Graph stellt dem Systemarchitekten drei verschiedene Kriterien zur Verfügung anhand derer er entscheiden kann, welche Komponentenoperationen zum untersuchten Fehler beitragen (Schritt 1). Zum einen ist das die Blame-Heuristik, die mithilfe des KS-Tests berechnet wurde und im Blame-Graphen durch die Farbcodierung und die Form der Boxen dargestellt wird. Zum anderen sind das die Aufrufhierarchie und die Medianantwortzeit. Im zweiten Schritt schaut der Systemarchitekt in den Performance-Report. Dort findet er den Box-Whisker-Plot für jede Komponentenoperation sowie die numerischen Ergebnisse für die Lageparameter und den KS-Test. Mithilfe dieser detaillierten Ergebnisse kann er die Ergebnisse aus dem Blame-Graphen prüfen und zu einer Einschätzung gelangen. Sollte der Systemarchitekt noch unsicher sein, so sollte er aufgrund der beobachteten Daten eigene Analysen vornehmen (Schritt 3). Aufgrund dieser drei Schritte entscheidet der Systemarchitekt für jede Komponentenoperation, ob sie zu beschuldigen ist oder nicht (Schritt 4).

Die getroffene Entscheidung sollte dann, wie in Abbildung 4.9 dargestellt, umgesetzt werden (vgl. dazu auch Abschnitt 3.1). Wenn es beschuldigte Kom-

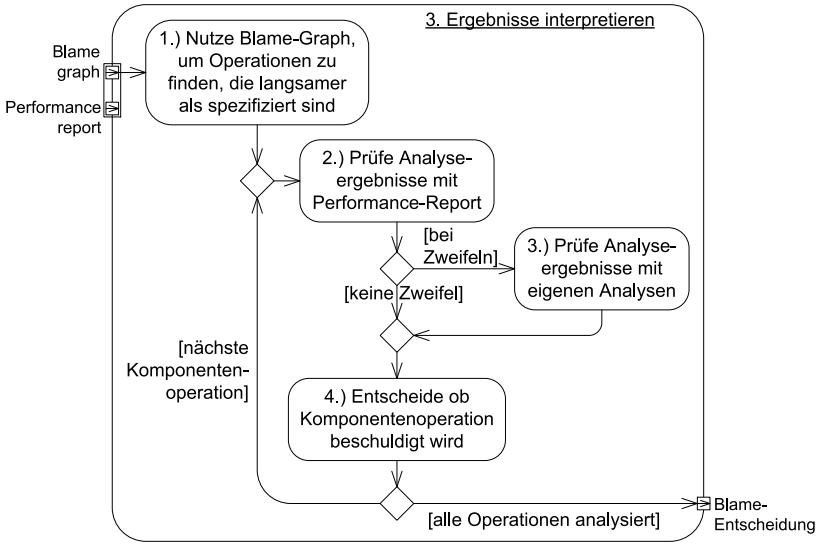

Abbildung 4.8: Der PBlaman-Subprozess, in dem der Systemarchitekt die Entscheidungsartefakte interpretiert und die Blame-Entscheidung fällt

ponentenoperationen gibt, sollte der Systemarchitekt zunächst mit den jeweiligen Komponentenentwicklern in Kontakt treten und sie auffordern den Fehler zu beseitigen. Da der PBlaman-Ansatz bewertet, wie gut die Implementierung einer Komponente zu ihrer Spezifikation passt, kann eine Änderung in jedem der beiden Artefakte dazu führen, dass die Artefakte wieder zusammenpassen. Sollte es keine beschuldigten Komponentenoperationen geben, so sollen der Systemarchitekt und der System-Deployer eine Lösung des Problems finden, indem sie beispielsweise die Komposition, die Komponentenkonfiguration oder die Komponentenverteilung anpassen (vgl. Tabelle 4.3).

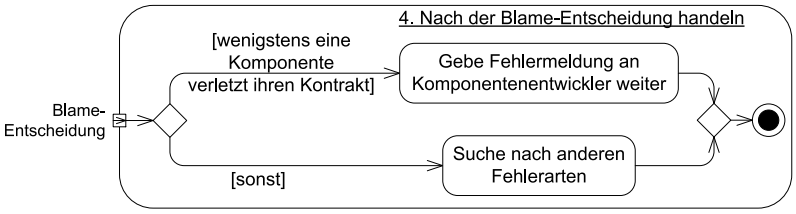

Abbildung 4.9: Der empfohlene Subprozess zur Umsetzung der Blame-Entscheidung

Die genannten Empfehlungen zur Entscheidungsumsetzung können nur eine Leitlinie darstellen. Zum einen setzt die empfohlene Umsetzung voraus, dass die Komponentenentwickler bekannt sind und dazu verpflichtet sind, den gemeldeten Fehler zu beseitigen. Zum anderen hat der Systemarchitekt noch andere Handlungsoptionen. Tabelle 4.3 zeigt einen Maßnahmenkatalog, der einen Überblick über die Handlungsoptionen des Systemarchitekten gibt. Dabei werden jeweils auch die Voraussetzungen für die jeweilige Maßnahme dargestellt.

Tabelle 4.3: Die Tabelle zeigt die Maßnahmen und ihre Voraussetzungen, mit denen der Systemarchitekt auf die Ergebnisse der Performance-Blame-Analysis reagieren kann.

Maßnahme	Voraussetzung für Maßnahme
Komponentenentwickler kontaktieren und eine Fehlerkorrektur verlangen	Komponentenentwickler sind bekannt und verpflichtet zur Fehlerkorrektur
Komponenten ersetzen	-
Komponentenverteilung anpassen	-
Komponentenkonfiguration anpassen	-
Systemarchitektur ändern	-
Leistungsfähigere Hardware kaufen	-
Den Fehler selbst in der Komponente entfernen	Open-Source-Komponente
Die Anforderung des Systems anpassen	Einverständnis der Kunden und Entscheider

4.5 PBlaman Werkzeugkette

In diesem Abschnitt wird erläutert, welche Werkzeuge entlang des dargestellten Prozesses (s. Abschnitte 4.2 bis 4.4) eingesetzt werden. Zudem wird dargestellt, wer die Werkzeuge nutzt und welche Werkzeuge speziell für PBlaman implementiert wurden. Außerdem werden die Datenformate für die Messdaten genau erläutert, die später als Eingabe für das Analyseskript (R/Python-Skript) dienen.

Zunächst wird die PCM-Workbench eingesetzt, um die PCM-Diagramme zu erstellen. Im Rahmen der komponentenbasierten Softwareentwicklung

wird die PCM-Workbench von verschiedenen Benutzern verwendet (s. Abschnitt 2.3). Bei der Performance-Blame-Analysis erstellt der Systemarchitekt die Modelle des zu untersuchenden Systems für die zu untersuchenden Testfälle. Dabei setzt er das System und den Testfall, nach ihrer jeweiligen Spezifikation, in einer PCM-Instanz um. Zusätzlich kann die PCM-Workbench auch von den Testern dazu verwendet werden, um Palladio-basierte Testfälle (s. Abschnitt 4.1) zu erstellen. Diese Testfälle können dann mittels eines Quelltext-Generators [25] in JUnit-Testskripte umgewandelt werden. Mithilfe der PCM-Workbench werden auch die zu den untersuchten Testfällen äquivalenten PCM-Instanzen simuliert und die jeweiligen Ergebnisse werden als CSV-Datei exportiert. Die CSV-Datei enthält dabei zwei Spalten: 1.) Der Zeitpunkt des Operationsaufrufs; und 2.) die Antwortzeit des Aufrufs. Der Zeitpunkt wird dabei relativ zum Simulationsstart angegeben.

Beim Sammeln der Testdaten werden zwei Werkzeuge benötigt. Zum einen wird eine Instrumentierung benötigt, die die Antwortzeit jeder Komponentenoperation erfassen kann. Die Instrumentierung wird von den Testern erstellt und im Rahmen der Testausführung für die Performance-Blame-Analysis verwendet. Die Instrumentierung kann auch für die regulären Performance-Tests verwendet werden und muss nicht zwingend exklusiv für die Performance-Blame-Analysis erstellt werden. Zum anderen wird ein Konverter benötigt, der die von der Instrumentierung erzeugten Messdaten in das von PBlaman geforderte Format übersetzt. Der Systemarchitekt muss den Konverter für PBlaman erstellen. Mithilfe des Konverters ist PBlaman von einer bestimmten Monitoring-Lösung unabhängig. Zudem braucht sich die Instrumentierung nicht darum zu kümmern direkt dieses Format zu erstellen und der Overhead für die Umwandlung kann auf die Offline-Analyse übertragen werden.

Das von PBlaman für Testdaten geforderte Datenformat umfasst zwei CSV-Dateien. Die erste CSV-Datei enthält die eigentlichen Testdaten. In dieser Datei gibt es die folgenden Spalten:

1. Der Schlüssel, der die Operation bezeichnet
2. Die Thread-ID
3. Der Aufrufzeitpunkt
4. Die Antwortzeit
5. Die Referenz auf den Stacktrace

Der Stacktrace ist nicht in der Testdatendatei enthalten und wird dort lediglich referenziert. Die Referenz wird in einer zweiten CSV-Datei aufgelöst. Diese

Datei enthält drei Spalten: 1) Die Stacktrace-Referenz; 2) der Stacktrace und 3.) der Pfad zur dazu passenden Datei mit den Daten aus der PCM-Simulation. Der Vorteil bei der Verwendung von zwei Dateien ist die Kompression der Datendatei mithilfe der Stacktrace-Referenz. Die Stacktrace-Referenz nummeriert die Stacktraces und ordnet ihnen einen kurzen Beschreibungstext zu, so dass diese Referenzen auch besser lesbar sind als der vollständige Stacktrace. Außerdem lässt sich die zweite Datei, die die Referenzen wieder den Stacktraces zuweist, ebenfalls für die Zuordnung der Simulationsdaten verwenden. Diese Zuordnung wird in die dritte Spalte der zweiten Datei eingetragen.

Für die Performance-Blame-Analysis wird das schon erwähnte R/Python-Skript verwendet. Das Skript wurde im Rahmen der Entwicklung von PBlaman erstellt. Es erhält die Messdaten aus dem Test und der Performance-Vorhersage, eine Zuordnung der Komponentenoperationen in Performance-Vorhersage und Test sowie die Umwandlungsfaktoren (s. Unterabschnitt 4.3.3) als Eingabe. Python parst die vom Systemarchitekten gegebenen Kommandozeilenparameter und generiert ein R-Skript, das die Werte dieser Parameter enthält (vgl. Abbildung 4.10). Das R-Skript wird für die Analyse der Messdaten verwendet. Dabei berechnet es die Statistikdaten zu den Messdatenreihen und generiert den Performance-Report. Die Statistikdaten werden wiederum vom Python-Skript genutzt, um den Blame-Graphen zu erstellen. Schließlich gibt das Python-Skript dem Benutzer den Blame-Graphen und den Performance-Report zurück.

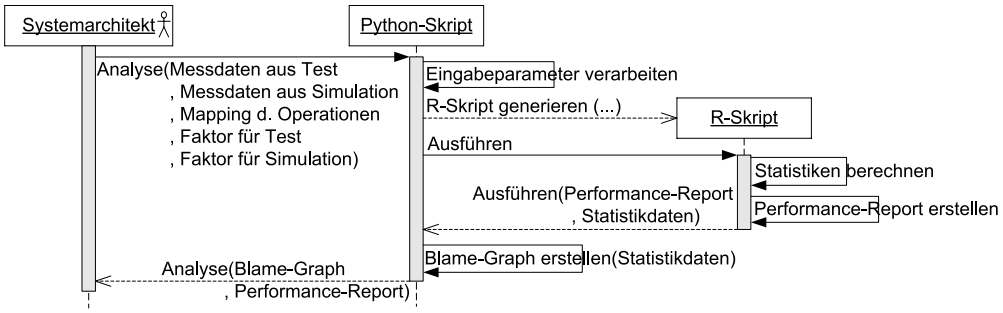

Abbildung 4.10: Abarbeitung des R/Python-Skripts mit besonderem Augenmerk auf die Aufgabenverteilung zwischen den R- und Python-Anteilen des Skripts

Kapitel 5

Evaluierung (und Realisierung)

In diesem Kapitel werden zwei Fallstudien vorgestellt, die die Anwendbarkeit von PBlaman zeigen. Dabei wird auch untersucht, ob die Artefakte Performance-Report und Blame-Graph (vgl. Unterabschnitt 4.3.2) konsistente Ergebnisse liefern. Die erste Fallstudie (s. Abschnitt 5.1) untersucht das CoCoME-System. Dieses System implementiert das Informationssystem einer Supermarktkette, das über eine Schichtarchitektur verfügt. Die zweite Fallstudie (s. Abschnitt 5.2) untersucht ein System, das aus Nachrichtentexten Aussagen über die zukünftige finanzielle Entwicklung von Unternehmen extrahiert. Dieses System verfügt über eine Pipes-and-Filters-Architektur. In der zweiten Fallstudie wird zudem die Qualität der Kategorisierung durch das primäre Entscheidungskriterium, also den KS-Test, evaluiert (s. Unterabschnitt 5.2.5). Der KS-Test kann unentschieden sein, anstatt eine Operation klar zu beschuldigen oder nicht zu beschuldigen. Daher wird untersucht, ob der Bereich in dem der KS-Test unentschieden ist, sinnvoll verkleinert werden kann. Schließlich fasst der Abschnitt 5.3 die Ergebnisse der Fallstudien zusammen.

5.1 Fallstudie: Common Component Modeling Example (CoCoME)

Dieser Abschnitt präsentiert die erste Fallstudie, die PBlaman auf die schon in Abschnitt 1.2 eingeführte Beispielanwendung CoCoME anwendet.[1] Dabei wird auch das dort verwendete Szenario als für die Performance-Blame-Analysis zu untersuchender Testfall betrachtet. Unterabschnitt 5.1.1 führt zunächst die beiden verwendeten Implementierungen ein. Die Unterabschnitte 5.1.2 bis 5.1.4 gehen darauf ein, wie die jeweiligen PBlaman-Prozessschritte (vgl. Kapitel 4) in der Fallstudie mit der ersten CoCoME-Implementierung

[1] Die Beispieldaten, die Analyseergebnisse und die Analyseskripte zu dieser Fallstudie finden sich auf der Begleitwebseite der Dissertation: http://homepages.uni-paderborn.de/bruesie/dissertation/index_de.html

eingesetzt werden. Dabei werden die Aufwände für den Systemarchitekten ebenso dargestellt, wie technische Details, wie zum Beispiel die Instrumentierung von CoCoME. Schließlich wird in Unterabschnitt 5.1.5 geschildert, welche zusätzlichen Erkenntnisse sich aus den Experimenten mit der zweiten CoCoME-Implementierung gewinnen lassen.

5.1.1 Getestete Implementierungen

CoCoME ist ein gutes Beispiel für ein Informationssystem mit einer Schichtarchitektur, das zudem mit Daten gefüttert wird, die mittels eingebetteter Systeme erhoben werden. Die CoCoME-Architektur besteht aus mehreren Subsystemen. Das Informationssystem, das im Folgenden für die Analyse genutzt wird, besteht aus insgesamt neun verschiedenen Komponenten. Diese Komponenten werden auf vier verschiedenen Hardwareknoten ausgeführt, dem Filial-Server, dem Filial-Client, dem Zentral-Server und dem Zentral-Client. Jeder Filial-Server führt insgesamt fünf Komponentenobjekte aus, während es beim Zentral-Server sechs sind. Die Clients nehmen nicht am untersuchten Testfall teil, daher wird hier nur auf die Server eingegangen.

CoCoME wurde ursprünglich mit einer Java-Referenzimplementierung veröffentlicht[2]. Um den in Abschnitt 1.2 eingeführten Testfall ausführen zu können, wurde die Referenzimplementierung zunächst funktionstüchtig gemacht. Diese funktionstüchtige Implementierung umfasst 6320 Zeilen Quelltext. Im Folgenden wird diese Implementierung als CoCoME-Referenzimplementierung bezeichnet. Neben dieser Implementierung hat das Karlsruhe Institute of Technology (KIT) die ursprüngliche Referenzimplementierung ebenfalls repariert und verbessert. Diese Implementierung stellt das KIT als CoCoME 2 zur Verfügung[3]. Das KIT wartet CoCoME 2 weiterhin, um es für den zukünftigen Gebrauch zu erhalten. CoCoME 2 enthält insgesamt 9788 Zeilen Quelltext.

5.1.2 Performance-Daten sammeln

Um die Daten aus dem Performance-Test zu sammeln, wird der zu untersuchende Testfall mit einer passenden Instrumentierung wiederholt ausgeführt (s. Abschnitt 4.2). In der Beispielanwendung wurden ein Lastgenerator und die Instrumentierung für die Messung der Antwortzeiten in die Testumgebung integriert. Die Instrumentierung injiziert dabei zur Laufzeit Bytecode in den

[2] s. http://cocome.org (zuletzt besucht am 24.05.2014)

[3] s. https://sdqweb.ipd.kit.edu/wiki/CoCoME2 und http://sourceforge.net/projects/cocome/ (beide zuletzt besucht am 24.05.2014)

CoCoME-Bytecode. Das Werkzeug BTrace[4] stellt sowohl den Messcode als auch die Bytecode-Injizierung in den CoCoME-Code zur Verfügung. Darüber hinaus bietet es auch die Möglichkeit, die zu protokollierenden Daten abzufragen und sie in eine Datei zu schreiben. Im Rahmen der Fallstudie wurde ein kurzes BTrace-Programm erstellt, das die nötige Antwortzeit-Messung ermöglicht. Der von BTrace injizierte Bytecode misst die Zeit, die zwischen dem Operationsaufruf und dem Zurückgeben des Ergebnisses aufseiten des ausführenden Hardwareknotens, vergeht. Diese selbst erstellte Lösung erfasst das PBlaman-CSV-Datenformat direkt (vgl. Abschnitt 4.5). Trotzdem ist auch hier ein Konverter nötig ist, da BTrace den Stacktrace in mehreren Zeilen ausgibt. Zudem wird der Stacktrace in eine zweite Datei ausgelagert und somit die Datendatei komprimiert.

Die Testumgebung für die Ausführung des Testfalls mit der CoCoME-Referenzimplementierung und CoCoME 2 ist sehr einfach gestaltet. Das CoCoME-System und der Lastgenerator wurden beide auf einem einzelnen PC ausgeführt. Der PC verfügt über eine Pentium-D-3-GHz-Einzelkern-CPU mit zwei Threads sowie 2 GB RAM und das Windows-7-Betriebssystem in der 64-bit-Version. Das CoCoME-System besteht aus drei Filial-Servern und einem Zentral-Server sowie der dazu gehörenden Infrastruktur, das heißt, der RMI-Registry, einem Active-MQ-JMS-Nachrichten-Server und einer Apache-Derby-Datenbank. In diesem Testfall werden keine Kassensysteme ausgeführt. Der Lastgenerator simuliert acht parallel auf das System zugreifende Benutzer. Jeder Benutzer richtet 125 Anfragen an denselben Filial-Server (insgesamt 1000 Anfragen), um ein gewisses Maß an Parallelverarbeitung zu erzeugen. Allerdings wurde das Maß der Parallelverarbeitung nicht gemessen. Dazu hätte man beispielsweise die Warteschlangen-Länge für die Benutzeranfragen messen müssen.

Im Rahmen der Performance-Vorhersage hat der Systemarchitekt zunächst die verfügbare CoCoME-PCM-Instanz (ausschnittsweise in [44]) angepasst, damit sie äquivalent zum untersuchten Testfall ist. Dann muss der Systemarchitekt die PCM-Simulation für diese PCM-Instanz starten. Während der Simulation sammelt die Palladio-Workbench die Antwortzeitdaten, die der Systemarchitekt anschließend manuell exportieren muss (s. Unterabschnitt 4.2.2).

5.1.3 Entscheidung unterstützen

Im Folgenden wird zunächst die Skript-Konfiguration, die vom Systemarchitekten vorgenommen werden muss, erläutert. Dann werden Generierung und

[4] s. https://kenai.com/projects/btrace (zuletzt besucht am 24.05.2014)

Beispiele für beide Entscheidungsunterstützungsartefakte, also Performance-Report und Blame-Graph, illustriert.

Skript-Konfiguration

Zunächst muss der Systemarchitekt die Konfiguration für das Python/R-Skript, das den Performance-Report und den Blame-Graphen generiert, festlegen. Der Systemarchitekt muss dazu die Komponentenoperationen in den Testdaten denen in der PCM-Instanz zuordnen (s. Unterabschnitt 4.3.3). Da die Zuordnung auf den Test-Stacktraces basiert, muss die Zuordnung für jedes System individuell erstellt werden. Bei der Zuordnung können aber generelle Informationen der Komponentenentwickler über die Zuordnung der Modellelemente zur Implementierung genutzt werden. Der Systemarchitekt ordnet dabei den im Test beobachteten Stacktraces die dazu passenden PCM-Datenreihen zu. Dazu trägt der Systemarchitekt in der CSV-Datei mit den vollständigen Stacktraces die jeweils zum Stacktrace passende aus Palladio exportierte Datei ein (vgl. Abschnitt 4.5). Tabelle 5.1 zeigt eine Beispielzuordnung, bei der in der linken Spalte der komprimierte Java-Stacktrace zu finden ist und in der rechten Spalte die CSV-Datei aus der PCM-Simulation.

Tabelle 5.1: Zuordnung vom komprimierten Java-Stacktrace zur CSV-Datei mit den Antwortzeitdaten für einen bestimmten Operationsaufruf aus der PCM-Simulation

Java-Stacktrace	**PCM-CSV-Datei**
(9) overall	bookSale0.csv
(8) otherStoreInterchange	orderProductsAvailableAtOtherStores0.csv
(7) markStock	markProductsUnavailableInStock1.csv
(6) solveOptimization	solveOptimization0.csv

Als Nächstes muss der Systemarchitekt die Antwortzeitwerte aus dem Test und der Performance-Vorhersage in eine gemeinsame Zeiteinheit konvertieren (s. Unterabschnitt 4.3.3). Dazu muss der Systemarchitekt eine Konfigurationsdatei anlegen, bei der in der ersten Zeile der Faktor für alle Datenreihen aus dem Test und in der zweiten Zeile der Faktor für alle Datenreihen aus der Performance-Vorhersage steht. In der Beispielanwendung werden die Testdatenreihen von Nanosekunden zu Millisekunden konvertiert und die Datenreihen aus der Performance-Vorhersage von Sekunden nach Millisekunden. Damit ergibt sich die Konfigurationsdatei mit dem Inhalt:

```
0.001*0.001
1000
```

Die bisher beschriebenen manuellen Schritte zur Skript-Konfiguration ließen sich in ca. einer Stunde für jede der CoCoME-Implementierungen erledigen. In Zukunft können auch das Exportieren der Messungen aus der Palladio-Workbench sowie die Berechnung der Konvertierungsfaktoren für die Antwortzeitendatenreihen auf Grundlage von Zeiteinheiten automatisiert werden. Dies gilt auch für das Erstellen der Zuordnung, vorausgesetzt, die Komponentenentwickler geben schon die Beziehung der Modellelemente zur Implementierung an. Diese Beziehung ist beispielsweise automatisch gegeben, wenn die Komponentenentwickler die PCM-Instanz mittels Reverse-Engineering nach dem SoMoX-Ansatz [7] erstellen.

Performance-Report

Während der Blame-Graph das primäre Entscheidungsunterstützungsartefakt ist, kann der Performance-Report eine Detaildarstellung beisteuern, die die Datenreihen aus dem Test und der Performance-Vorhersage gegenüberstellt. Der Performance-Report besteht aus einem Box-Whisker-Plot, das beide Datenreihen darstellt, sowie den Lageparametern (Maximum, Minimum, Quartile und arithmetisches Mittel) und den numerischen KS-Test-Ergebnissen (s. Unterabschnitt 4.3.2).

Das Python/R-Skript berechnet aufgrund der gegebenen Konfiguration die Punktschätzer der Lageparameter. Tabelle 5.2 zeigt die berechneten Werte für die Operation „(8) otherStoreInterchange“ (vgl. Nachricht 3 in Abbildung 1.3). Der Systemarchitekt kann die Werte der Lageparameter vergleichen, um sich einen Eindruck zu verschaffen, welche Datenreihe die größeren Werte hat. Die Werte helfen dabei, die Visualisierungen nachvollziehbarer zu machen. In Tabelle 5.2 sind alle Lageparameter der Test-Datenreihe größer als bei der Datenreihe aus der Performance-Vorhersage.

Zudem generiert das Python/R-Skript auch ein Box-Whisker-Plot für jede Operation. Abbildung 5.1 zeigt den Box-Whisker-Plot für die Operation „(8) otherStoreInterchange“ aus dem zu untersuchenden Testfall. In dem Box-Whisker-Plot werden die Antwortzeitdatenreihen aus dem Test und der Performance-Vorhersage gegenübergestellt. Um eine bessere Lesbarkeit zu erreichen, zeigt der Box-Whisker-Plot 1% der größten Messwerte nicht mit an. Das betrifft sowohl die Testdatenreihe als auch die Performance-Vorhersage-Datenreihe. In Abbildung 5.1 liegt die Box der Testdatenreihe über der Box

Tabelle 5.2: Numerische Werte der Lageparameter für Operation „(8) otherStoreInterchange“ aus dem Performance-Report

Lageparameter	PCM-Antwortzeit	Test-Antwortzeit
Minimum	163,1 ms	760,7 ms
25%-Quartil	1120,8 ms	2528,6 ms
Median	1506,2 ms	3279,6 ms
75%-Quartil	1952,4 ms	4012,1 ms
Maximum	3186,0 ms	24595,4 ms
Mittelwert	1500,3 ms	3248,7 ms

der Performance-Vorhersage-Datenreihe, was darauf hindeutet, dass die Testdatenreihe die größeren Werte beinhaltet.

Darüber hinaus beinhaltet der Performance-Report auch die numerischen KS-Test-Ergebnisse. Genauer gesagt weist der Report den Wahrscheinlichkeitswert, also den sogenannten p-Wert [18], für beide Varianten des KS-Tests aus. Der Wert gibt an wie wahrscheinlich es ist die gegebenen Daten zu erhalten, wenn man annimmt, dass die Nullhypothese gilt.

Blame-Graph

Das Python/R-Skript generiert neben dem Performance-Report auch den Blame-Graphen. Der Blame-Graph visualisiert mithilfe der KS-Test-Ergebnisse, welche Komponentenoperationen beschuldigt werden sollen (vgl. Unterabschnitt 4.3.2). Die KS-Test-Implementierung aus R berechnet aus den beiden Antwortzeitdatenreihen aus dem Test und der Performance-Vorhersage den p-Wert. Dabei wird die Hypothese der jeweiligen KS-Test-Ausführung akzeptiert, wenn der p-Wert 5% nicht übersteigt. Im Rahmen der zweiten Fallstudie (s. Unterabschnitt 5.2.5) wird diese initiale p-Wert-Schwelle von 5% evaluiert.

Der Blame-Graph visualisiert die KS-Test-Ergebnisse, die Aufrufhierarchie aus der Testfallausführung sowie die Medianantwortzeit für jede Komponentenoperation (vgl. Unterabschnitt 4.3.2). Für den untersuchten Testfall generiert das Python/R-Skript den in Abbildung 5.2 dargestellten Blame-Graphen. Die Operationsnamen in den Boxen des Blame-Graphen korrespondieren mit den Methodennamen aus der CoCoME-Referenzimplementierung. Dabei werden die Operationsnamen nicht als vollqualifizierte Namen dargestellt, da der

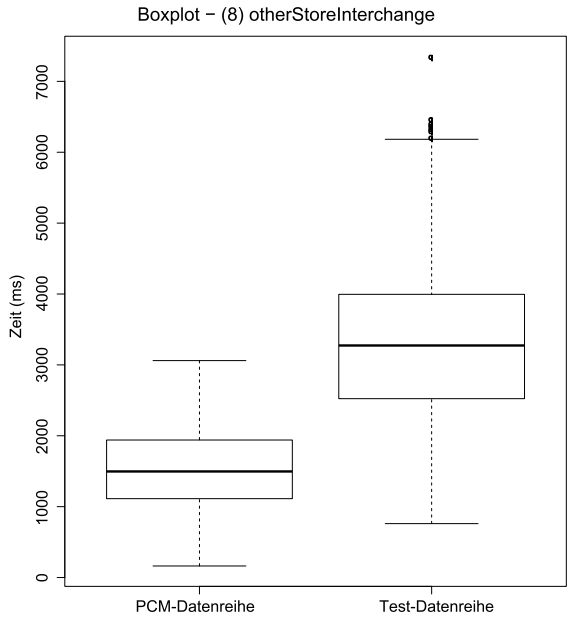

Abbildung 5.1: Box-Whisker-Plot für die Operation „(8) otherStoreInterchange“ aus dem Performance-Report

Platz in den Boxen knapp ist. Die Boxbreite hängt von der Medianantwortzeit ab und daher kann selbst ein kurzer Name schon abgeschnitten werden oder ist sogar gar nicht sichtbar. Der vollqualifizierte Methodenname wird deshalb zusammen mit der zugehörigen Medianantwortzeit, neben dem Text „Function“ dargestellt, wenn sich der Mauszeiger über der jeweiligen Box befindet. Der Blame-Graph ist als SVG-Grafik realisiert, die solche interaktiven Elemente unterstützt. In dem in Abbildung 5.2 dargestellten Blame-Graphen kann der Systemarchitekt erkennen, dass drei Komponentenoperationen beschuldigt werden. Die beschuldigten Komponentenoperationen werden als rote Fußtrapeze dargestellt. Die anderen Komponentenoperationen sind klar nicht zu beschuldigen und werden daher als grüne Kopftrapeze dargestellt. Wie schon der Performance-Report angedeutet hat, wird die `ProductDispatcher.orderProductsAvailableAtOtherStores(...)`-Java-Methode beschuldigt, die die Operation „(8) otherStoreInterchange“ implementiert. Die Ergebnisse im Blame-Graphen sind also konsistent mit denen im Performance-Report.

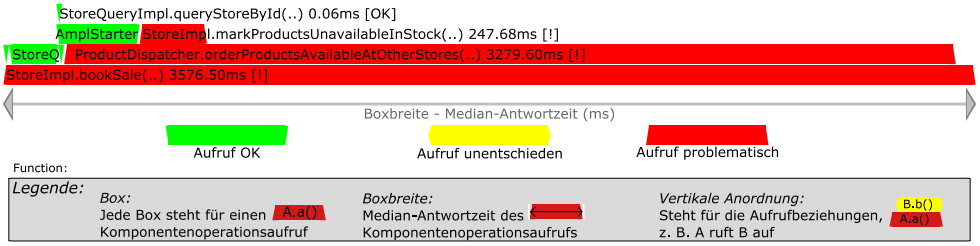

Abbildung 5.2: Der Blame-Graph für den zu untersuchenden Testfall in der Beispielanwendung

5.1.4 Ergebnis interpretieren

Der Systemarchitekt beginnt seine Analyse, indem er den Blame-Graphen untersucht. Dort identifiziert er die Operationen, die langsamer als erwartet sind. Im zu untersuchenden Testfall aus der Beispielanwendung sind dies die folgenden drei Operationen:

1. `StoreImpl.bookSale(...)`
2. `ProductDispatcher.orderProductsAvailableAtO-therStores(...)`
   // implementiert die Operation „(8) otherStoreInterchange“
3. `StoreImpl.markProductsUnavailableInStock(...)`

Der Systemarchitekt wählt nun die Reihenfolge, in der die beschuldigten Operationen weiter analysiert werden. Er kann die Operationen entweder nach Aufrufhierarchie (also nach der vertikalen Box-Anordnung) oder nach Medianantwortzeit (also nach der Box-Breite) priorisieren. Beispielsweise kann sich der Systemarchitekt dazu entscheiden, die dritte beschuldigte Operation zuerst zu analysieren, da diese Operation keine andere Operation aufruft. Dann untersucht der Systemarchitekt mithilfe des Performance-Reports, warum die Operationen beschuldigt wurden, wobei er nach der gewählten Reihenfolge vorgeht. Für die dritte beschuldigte Operation kann er beispielsweise sehen, dass die vorhergesagte Medianantwortzeit nur ein Zehntel der im Test gemessenen Antwortzeit beträgt, was die größte Abweichung darstellt. Außerdem kann die absolute Antwortzeitdifferenz von einer Sekunde für die zweite Operation nicht mit der Abweichung in den aufgerufenen Operationen, also der dritten Operation, erläutert werden. Diese Differenz findet sich auch so für die erste Operation. D. h., dass die erste Operation fälschlicherweise beschuldigt wurde, da die Antwortzeitabweichung auf die aufgerufene Operation, also

die zweite Operation, zurückzuführen ist. Dagegen hat sich der Verdacht bei der zweiten und dritten Operation klar bestätigt. Dieses Beispiel zeigt, dass der Systemarchitekt PBlaman auf die CoCoME-Referenzimplementierung anwenden lässt. Dabei erhält der Systemarchitekt die beschuldigten Komponentenoperationen aus dem Blame-Graphen, die er anhand der detaillierten Werte und Darstellungen im Performance-Report überprüfen kann.

5.1.5 Zusätzliche Erkenntnisse mit CoCoME 2

Während die Unterabschnitte 5.1.2 bis 5.1.4 sich auf die CoCoME-Referenzimplementierung beziehen, werden nun kurz die zusätzlichen Erkenntnisse aus der Fallstudie mit CoCoME 2 beschrieben. Dabei wurden die Erkenntnisse aus der Fallstudie mit der CoCoME-Referenzimplementierung größtenteils bestätigt. Der Systemarchitekt kann PBlaman genauso auf CoCoME 2 anwenden wie zuvor auf die CoCoME-Referenzimplementierung. PBlamans semiautomatische Entscheidungsunterstützung, also das Python/R-Skript, generiert den Blame-Graphen und den Performance-Report ohne großen Aufwand. Die aufwendigsten Schritte in der Fallstudie waren die Implementierung des Testtreibers und der Instrumentierung. Der Aufwand zur Erstellung und Analyse der Entscheidungsunterstützungsartefakte war im Vergleich mit diesen Schritten gering.

Der auffälligste Unterschied zwischen den beiden CoCoME-Fallstudien ist, dass der Blame-Graph Komponentenoperationen aufweist, bei denen der KS-Test unentschieden war. Der Performance-Report spiegelt dies wider. Er zeigt aber auch, dass für zwei der unentschiedenen Komponentenoperationen durchaus eine Entscheidung möglich ist. Dies unterstreicht, wie wichtig es ist, dass der Performance-Report zur Verfügung steht, um die Ergebnisse aus dem Blame-Graphen zu prüfen. Der Performance-Report ist immer dann mit dem Blame-Graphen konsistent, wenn der KS-Test auch eine Entscheidung trifft, also die Datenreihen sich genügend unterscheiden. Zudem zeigt der Blame-Graph für die Fallstudie zu CoCoME 2, dass die Instrumentierung nicht in der Lage ist, Anfragen zu verfolgen, die über Prozessgrenzen hinausgehen. Operationsanfragen an andere Prozesse werden im Blame-Graphen wie Operationen dargestellt, die direkt vom Testtreiber aufgerufen werden.

5.2 Fallstudie: Analyse unstrukturierter Texte

Abschnitt 5.1 hat gezeigt, wie PBlaman auf das bekannte Benchmark-System CoCoME angewendet wird. Im Folgenden wird dargestellt, wie PBlaman

auf ein weiteres komponentenbasiertes System angewendet werden kann.[5] Dieses System hat eine Pipes-and-Filters-Architektur [14]. Das System extrahiert strukturierte Informationen aus unstrukturierten Texten und basiert auf einem Standard-Framework aus dem Bereich der Analyse unstrukturierter Informationen. In Unterabschnitt 5.2.1 wird zunächst das System und das Framework eingeführt. Zudem wird der zu untersuchende Testfall für die Performance-Blame-Analysis erläutert. Dann wird in den Unterabschnitten 5.2.2 bis 5.2.4 wiederum illustriert, wie die einzelnen Schritte des PBlaman-Prozesses (s. Kapitel 4) auf das vorgestellte System angewendet werden. Über die Anwendbarkeit PBlamans hinaus wird auch gezeigt, dass Blame-Graph und Performance-Report, wie zuvor, konsistente Ergebnisse liefern. Zusätzlich wird für das primäre Entscheidungskriterium, den KS-Test, überprüft, wie groß die Spanne ist, in der der KS-Test unentschieden ist. Darauf aufbauend wird analysiert, ob diese Spanne sinnvoll verringert werden kann (s. Unterabschnitt 5.2.5). Schließlich werden in Unterabschnitt 5.2.6 noch die Ergebnisse der Fallstudie zusammengefasst.

5.2.1 System und getesteter Anwendungsfall

Im Folgenden wird ein Beispiel-System aus dem Bereich der Informationsextraktion untersucht. Der Begriff Informationsextraktion [64] umfasst Aufgaben aus dem Bereich der Verarbeitung natürlicher Sprache. Im Rahmen dieser Aufgaben werden strukturierte Informationen über Entitäten und Ereignisse, an denen die Entitäten beteiligt sind, aus unstrukturierten Texten extrahiert. Unter anderem wird die Informationsextraktion zur Verbesserung der Ergebnisse von Echtzeitsuchmaschinen eingesetzt, bei denen die Performance kritisch ist. Dabei kommt für Extraktion der geforderten Informationen aus den Eingabetexten eine geordnete Menge von Extraktionsalgorithmen zum Einsatz, die fest vorgegeben oder nach Anfrage dynamisch zusammengesetzt [77] werden kann. Das Folgende beschränkt sich dabei auf die Betrachtung einer einzelnen Extraktionsaufgabe, die mit einem festen Satz an Extraktionsalgorithmen bearbeitet wird.

Aufgrund von Abhängigkeiten der verschiedenen Extraktionsalgorithmen untereinander werden sie in einer Pipeline angeordnet, so dass die Ausgabe eines Algorithmus die Eingabe des nächsten Algorithmus in der Pipeline darstellt. Diese Pipes-and-Filters-Architektur ist gängig im Bereich der Informationsextraktion. Deshalb ist dies auch die Architektur, die vom Standard-Framework

[5] Die Beispieldaten, die Analyseergebnisse und die Analyseskripte zu dieser Fallstudie finden sich auf der Begleitwebseite der Dissertation: http://homepages.uni-paderborn.de/bruesie/dissertation/index_de.html.

Apache UIMA[6] („Unstructured Information Management Architecture“) unterstützt wird. Dieses Framework wird auch von der Beispielanwendung verwendet. Apache UIMA ist eine Open-Source-Implementierung des UIMA-Standards für Softwaresysteme, die unstrukturierte Texte und Informationen analysieren [23]. Die UIMA wurde von der OASIS für die semantische Suche und Inhaltsanalyse standardisiert [54]. Apache UIMA erlaubt es dem Systementwickler verschiedene sogenannte „primitve analysis engines“ (PAE), also Extraktionsalgorithmen, zu einer „aggregate analysis engine“ (AAE), bzw. Pipeline, zu komponieren. Die Pipeline erhält die zu analysierenden Texte aus einem sogenannten „collection reader“, der die Eingabetexte nacheinander aus einer Sammlung von Texten liest. Die PAEs sind also einfache Komponenten und die AAEs bzw. Pipelines komponierte Komponenten. AAEs können also wiederum in andere AAEs eingebunden werden.

Ein System, das auf dem Apache-UIMA-Framework basiert, besteht üblicherweise aus einer AAE, die jeden Eingabetext einer Sammlung nacheinander analysiert. Für jeden Eingabetext ruft das Framework dann automatisch die `process`-Operation jeder im AAE enthaltenen PAE oder AAE auf. Eine PAE verarbeitet den Eingabetext sowie die schon erkannten Informationen über den Text (z. B. Annotationen über die Sätze im Text), um neue Informationen über den Text zu extrahieren (z. B. Annotationen über Token in den Sätzen). Das Framework speichert alle Informationen über den Eingabetext in einem einzigen Objekt, der sogenannten „common analysis structure“ (CAS), das von einer PAE zur nächsten weitergereicht wird. Damit das Framework den Kontroll- und Datenfluss automatisch steuern kann, werden PAEs und AAEs mit einer Metadatenbeschreibung versehen. Die Metadatenbeschreibung spezifiziert die Eingabe- und Ausgabeinformationen im CAS auf Attributebene. Bei AAEs kommt noch die Reihenfolge der aufzurufenden PAEs bzw. AAEs hinzu.

In dieser Fallstudie wird ein System betrachtet, dass aus Eingabetexten Vorhersagen über die finanzielle Entwicklung von Unternehmen über die Zeit erkennt. Diese Aussagen werden in einer Sammlung von insgesamt 1128 deutschsprachigen Nachrichtenartikeln gesucht, dem sogenannten „Revenue Corpus“ [76]. Wachsmuth und Stein [78] haben ein System vorgestellt, das die gesuchten Aussagen mithilfe der in Tabelle 5.3 genannten Algorithmen extrahiert. Für jeden Algorithmus in Tabelle 5.3 ist auch die erwartete Performance als mittlere Antwortzeit je Satz spezifiziert. Darauf aufbauend wird in Abbildung 5.3 die Abarbeitung der Textsammlung durch ein UIMA-basiertes System dargestellt. In Abbildung 5.3 wird dabei jede PAE als Komponentenobjekt mo-

[6] s. http://uima.apache.org (zuletzt besucht am 15.09.2014)

delliert. Da PBlaman nicht mit komponierten Komponenten umgehen kann, wird die AAE ebenfalls als Komponentenobjekt modelliert, das für die Koordinationsfunktion des Frameworks steht. Die Extraktionsalgorithmen werden von dem Komponentenobjekt der Klasse `AggregateAnalysisEngine_Impl` zu einer Pipeline komponiert und in der dargestellten Reihenfolge aufgerufen. Ein Operationsaufruf wird von einem Pfeil mit ausgefüllter Spitze dargestellt und gestrichelte Pfeile modellieren die Rückgabe aus der Operation. Falls die Rückgabepfeile nicht nummeriert sind, erfolgt die Rückgabe direkt nach dem Aufruf der Operation. Zunächst holt der Testtreiber ein CAS-Objekt vom `AggregateAnalysisEngine_Impl`-Komponentenobjekt. Das `CollectionReader_ImplBase`-Komponentenobjekt kontrolliert die Textsammlung und lädt einen Text und speichert ihn im CAS-Objekt. Im dritten Schritt lässt der Testtreiber den Text durch die AAE verarbeiten. Die AAE ruft dazu der Reihe nach die `process`-Operation jeder PAE auf (Schritte 4 bis 10). Nach Schritt 10 enthält das CAS-Objekt die gefundenen Informationen und wird an den Testtreiber zurückgegeben (Schritt 11).

Im Folgenden wird ein Performance-Testfall untersucht, dessen Testkriterium eine Schwelle für die mittlere Antwortzeit der `process`-Operation des AAE, also die Zeit zwischen dem Beginn von Schritt 3 und dem Ende von Schritt 11 (s. Abbildung 5.3), überprüft. Der Testfall untersucht die durchschnittliche Antwortzeit bei der Untersuchung der einzelnen Texte, denn die in Abbildung 5.3 dargestellten Schritte werden wiederholt für jeden Text in der Sammlung aufgerufen. Die beschriebene Antwortzeitschwelle ist bei Durchführung des Testfalls nicht eingehalten worden, so dass im Folgenden die Performance-Blame-Analysis für diesen Fehler beschrieben wird.

Der Anwendungsfall, der in Abbildung 5.3 dargestellt ist, repräsentiert eine Extraktionsaufgabe typischer Größenordnung. Die Komponenten `TreeTagger`[7] und `StanfordNER`[8] sind von Dritten beigesteuert, wohingegen die andere Komponenten von Wachsmuth entwickelt wurden [76], der geholfen hat die Fallstudie durchzuführen. Die Komponenten von Wachsmuth umfassen 3957 Zeilen Quelltext. Dies umfasst auch Aufrufe zum Apache-UIMA-Framework. Zusätzlich benutzt die Komponente `EfXForecastClassifier` intern die Weka-Bibliothek[9] für maschinelles Lernen. Auch die beiden Komponenten von Dritten setzen auf maschinelles Lernen und bauen dabei auf Modellen auf, die recht groß sind. Komprimiert ist das Modell für den `TreeTagger` ca. 35 MB und das für `StanfordNER` ca. 36 MB groß. Diese

[7] s. http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/ (zuletzt besucht am 28.05.2014)

[8] s. http://nlp.stanford.edu/software/CRF-NER.shtml (zuletzt besucht am 28.05.2014)

[9] s. http://www.cs.waikato.ac.nz/~ml/weka/ (zuletzt besucht am 28.05.2014)

Tabelle 5.3: Das Softwaresystem in der zweiten Fallstudie umfasst die dargestellten sieben Extraktionsalgorithmen. Jeder Algorithmus A hat bei der Analyse eines Satzes aus einem unstrukturierten Text die erwartete mittlere Antwortzeit $t(A)$.

Extraktionsalgorithmus A	**Analysebeschreibung für** A	$t(A)$	A **aus Quelle**
EfXSentenceSplitter	Trennt den Text in Sätze auf	0,47 ms	[76]
EfXTokenizer	Trennt jeden Satz in Token auf	0,60 ms	[76]
TreeTagger	Klassifiziert jedes Token nach Wortart	0,59 ms	[65]
EfXForecastClassifier	Klassifiziert jeden Satz als Vorhersage bzw. Nichtvorhersage	0,29 ms	[76]
StanfordNER	Erkennt Organisationsnamen in jedem Satz	2,52 ms	[24, 22]
EfXTimeRecognizer	Erkennt Zeitinformationen in jedem Satz	0,36 ms	[76]
EfXMoneyRecognizer	Erkennt Geldinformationen in jedem Satz	0,64 ms	[76]

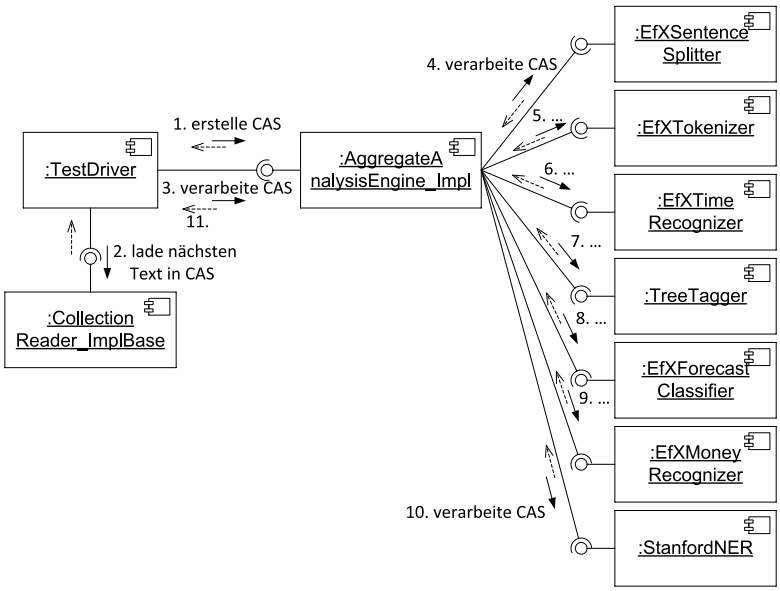

Abbildung 5.3: Der Anwendungsfall der zweiten Fallstudie mit allen benötigten Komponentenobjekten. Die Schritte 1 bis 11 stehen für Operationsaufrufe, die für jeden Eingabetext, den das Informationsextraktionssystem verarbeitet, wiederholt werden.

Modelle spezifizieren, wie die jeweiligen Komponenten eine Analyse vornehmen und welche Informationen aus dem Text bzw. über den Text mit einbezogen werden.

5.2.2 Performance-Daten sammeln

Zum Sammeln der Daten sind der Testfall sowie die Performance-Kontrakte aller beteiligten Komponenten erforderlich. Zum Testfall gehört dabei auch die Testumgebung, die insbesondere die Instrumentierung und den Testtreiber umfasst. Bei dieser Fallstudie kommt eine kleine Java-Anwendung als Testtreiber zum Einsatz, die, wie in Abbildung 5.3 zu sehen ist, die Texte lädt und diese zur Verarbeitung an die UIMA-Pipeline übergibt. Zur Instrumentierung (vgl. Unterabschnitt 4.2.1) kommt das Messframework Kieker zum Einsatz [73, 74]. Dabei wurden die Messfühler mittels aspektorientierter Programmierung in den Bytecode injiziert, um die Aufrufe der Komponentenoperationen zu instrumentieren. Kieker speichert alle Messdaten, die die Messfühler während der Ausführung messen, in einer Datei ab, hierbei wird ein Eintrag für das Eintreten in bzw. das Austreten aus einer Komponentenoperation erfasst. Jeder Eintrag zu einem Operationseintritt bzw. Operationsaustritt enthält den Eintragstypen, eine Kieker-spezifische Verfolgungs-ID, einen Zeitstempel sowie den voll-qualifizierten Namen der aufgerufenen Operation. Unter der Verfolgungs-ID wird die jeweilige Umgebung einer Messung einmalig festgehalten, dies umfasst die Thread-ID, den Hostnamen und eine optionale Session-ID. Der Overhead der Kieker-Instrumentierung ist relativ gering. Mit aktivierter Instrumentierung benötigt der gesamte Testfall im Mittel 1468 ms, was verglichen mit einer Ausführung ohne Instrumentierung einer um 1,6% erhöhten Laufzeit entspricht.

Anders als bei der BTrace-basierten Lösung aus der ersten Fallstudie (s. Abschnitt 5.1) erfasst Kieker nicht direkt den Stacktrace eines Operationsaufrufs. Das erfordert die Konvertierung der Daten (s. Abschnitt 4.2) in das für das Python/R-Skript nötige Datenformat. Dafür muss der Systemarchitekt ein Konvertierungswerkzeug erstellen, das den Stacktrace aus den protokollierten Einträgen rekonstruiert. Das Werkzeug muss dabei die zeitliche Abfolge der Ein- und Austrittsereignisse je Verfolgungs-ID berücksichtigen. In dieser Fallstudie baut das Konvertierungswerkzeug auf Kieker auf und umfasst 285 Zeilen Java-Quelltext. Das Konvertierungswerkzeug kommt nach jedem Testlauf zum Einsatz, um die angefallenen Daten zu konvertieren.

Die Testumgebung beschränkt sich auch hier auf einen einzelnen PC. Dabei kam ein Intel Mobile Core 2 Duo mit 2,2 GHz und 2 Kernen sowie 4 Threads

zum Einsatz. Der PC verfügt über 4 GB RAM und führt das Windows-7-Betriebssystem in der 32-Bit-Version aus. Auf dem PC wird sowohl die UIMA-Pipeline als auch der Testtreiber ausgeführt. Der Testtreiber lädt dabei je einen Text und veranlasst dessen Analyse durch die UIMA-Pipeline. Um die Caches aufzuwärmen, wird die gesamte Sammlung dreimal verarbeitet und die Instrumentierung wird dann erst für den vierten Durchlauf aktiviert. Die Verarbeitung wird zudem von einer einzelnen UIMA-Pipeline übernommen, so dass die Pipeline und die enthaltenen PAEs nicht nebenläufig ausgeführt werden. Allerdings nutzen einige Extraktionsalgorithmen, wie z. B. der `TreeTagger`, mehrere Prozesse und Threads, um von Parallelverarbeitung zu profitieren.

Die Performance-Vorhersage basiert auf den mittleren Antwortzeiten der Extraktionsalgorithmen aus der Tabelle 5.3. Aus diesen Angaben müssen PCM-Kontrakte für die jeweiligen Komponenten erstellt werden. Zunächst müssen die Werte auf die lokale Referenzhardware normalisiert werden. Dies ist wichtig, um im ganzen Projekt konsistente Angaben über den Verbrauch von Hardwareressourcen machen zu können. Denn im PCM können keine Einheiten für die Ressourcenverbräuche angegeben werden, so dass immer eine projektinterne Referenz definiert werden muss, die den Zahlwerten eine Bedeutung verleiht. Beim Import von Performance-Kontrakten muss somit ggf. von einer Referenz auf eine andere konvertiert werden. In der Fallstudie haben die Komponentenentwickler dafür einen eingebauten Test zur Verfügung gestellt, der die Antwortzeit der `EfXTokenizer`-Komponente misst. Die in der Fallstudie genutzte Referenzhardware führte den eingebauten Test ca. 17,9% langsamer aus als die der Komponentenentwickler, was einen Hardwarekorrekturfaktor von 1,179 ergibt. Nach Angabe der Komponentenentwickler ist die Komplexität der Extraktionsalgorithmen linear in der Tokenanzahl je Satz. Lediglich `StanfordNER` bildet hier eine Ausnahme, da diese Komponente quadratische Komplexität hat. Die in Tabelle 5.3 angegebene Performance wurden von den Komponentenentwicklern auf einer Textsammlung mit einer durchschnittlichen Tokenzahl von 19,49 Tokens je Satz ermittelt. Aus diesen Informationen kann der Systemarchitekt Service-Effect-Specifications (SEFFs) für die Komponentenoperationen erstellen. Die SEFFs sollen mittels einer Formel die Antwortzeit der jeweiligen Operation annähern, indem der CPU-Verbrauch der Operation spezifiziert wird. Zum Beispiel sieht die Formel für den CPU-Verbrauch der `process`-Operation der Komponente `EfXMoneyRecognizer` folgendermaßen aus:

$$(0,64ms \cdot 1,179)/19,49 \cdot cas.tokens.NUMBER\_OF\_ELEMENTS$$

In der gezeigten Formel wird die von den Entwicklern angegebene mittlere Antwortzeit von 0,64 ms mit dem Hardware-Korrekturfaktor 1,179 multipliziert. Damit erhält man die zu erwartende mittlere Antwortzeit für die Referenzhardware dieses Projekts. Als Nächstes wird die durchschnittliche Anzahl von 19,49 Tokens je Satz der Referenztextsammlung mithilfe der Angaben im PCM-Verwendungsmodell für die in diesem Testfall verwendete Textsammlung korrigiert. Das heißt, dass der Systemarchitekt diese Angaben für den jeweils verwendeten Testdatensatz im Verwendungsmodell als Eigenschaft des Eingabeparameters modellieren muss. Wenn die vollständige PCM-Instanz für den Testfall vorliegt, wird sie vom Systemarchitekten, wie gehabt, simuliert und die Antwortzeitdatenreihen werden entsprechend aus der Palladio-Workbench exportiert (s. Unterabschnitt 4.2.2).

5.2.3 Entscheidung unterstützen

Im Folgenden wird die Skript-Konfiguration durch den Systemarchitekten sowie die Generierung der Entscheidungsunterstützungsartefakte Performance-Report und Blame-Graph beschrieben.

Skript-Konfiguration

Zunächst muss der Systemarchitekt die Konfiguration für das Analyseskript bereitstellen (s. Unterabschnitt 4.3.3). Dazu erstellt der Systemarchitekt die Zuordnung der Komponentenoperationen aus der Implementierung zu ihrer Modellierung in der PCM-Instanz. Aufgrund des simplen Aufrufbaums, bei dem die `process`-Komponentenoperation jeder PAE in der Pipeline genau einmal je Dokument aufgerufen wird, ist die Zuordnung einfach herzustellen. Die Zuordnung der komprimierten Java-Stacktraces auf der einen und der PCM-Datendatei auf der anderen Seite ist in Tabelle 5.4 zu sehen.

Zusätzlich muss der Systemarchitekt die Konvertierungsfaktoren für die Antwortzeitdatenreihen aus dem Test und der Performance-Vorhersage angeben. Damit werden alle Datenreihen auf eine einheitliche Zeiteinheit konvertiert. In dieser Fallstudie müssen die Messungen aus dem Test von Nanosekunden auf Millisekunden konvertiert werden, wogegen die Daten aus der PCM-Simulation bereits in Millisekunden gegeben sind. Die beiden beschriebenen Maßnahmen zur Skript-Konfiguration konnten in dieser Fallstudie in ca. 45 Minuten vom Systemarchitekten erledigt werden. In dieser Fallstudie war die Zuordnung der Komponentenoperationen aufgrund der simplen Aufrufbeziehungen (vgl. Unterabschnitt 5.1.3) einfacher als in der CoCoME-Fallstudie.

Tabelle 5.4: Zuordnung vom komprimierten Java-Stacktrace zur CSV-Datei mit den Antwortzeitdaten für einen bestimmten Operationsaufruf aus der PCM-Simulation

Java-Stacktrace	**PCM-CSV-Datei**
(15) AggregateAnalysisEngine_impl.process	aggregateProcess02.csv
(0) EfXSentenceSplitter.process	EfXRBSentenceSplitter02.csv
(2) EfXTokenizer.process	EfXRBTokenizer02.csv
(4) EfXTimeRecognizer.process	EfXTimeRecognizer02.csv
(6) TreeTaggerPOSLemmaChunker.process	TT4jChunker02.csv
(8) EfXForecastClassifier.process	EfXForecastClassifier02.csv
(10) EfXMoneyRecognizer.process	EfXMoneyRecognizer02.csv
(12) StanfordNER.process	StanfordNER02.csv

Performance-Report

Der Performance-Report enthält die Werte der Lageparameter, die numerischen KS-Test-Ergebnisse sowie die Box-Whisker-Plots für die Datenreihen aus dem Test und der Performance-Vorhersage (s. Unterabschnitt 4.3.2). Tabelle 5.5 zeigt die Lageparameter für die Operation „(2) EFXTokenizer.process“ aus diesem Testfall. Anhand der Lageparameter ergibt sich kein einheitliches Bild, welche Datenreihe die größeren Werte hat. Das dazu passende Box-Whisker-Plot ist in Abbildung 5.4 zu sehen. Das Diagramm gibt dem Systemarchitekten einen Hinweis darauf, dass die Werte aus dem Test niedriger sind, da die Box, also 50% der Messwerte, auf der rechten Seite nur mit dem unteren Teil der Box auf der linken Seite überlappt. Allerdings gibt es bei der Test-Datenreihe auch Ausreißer, die deutlich über dem Maximum der Datenreihe aus der PCM-Simulation liegen. Daher muss auch in Betracht gezogen werden, dass die Werte beider Datenreihen insgesamt in etwa gleich sind.

Blame-Graph

Der Blame-Graph für die UIMA-Fallstudie ist in Abbildung 5.5 zu sehen. Er zeigt die beteiligten Komponentenoperationen und die Aufrufhierarchie mittels der vertikalen Boxanordnung (vgl. Unterabschnitt 4.3.2). Dadurch lässt sich auch hier klar erkennen, dass das Framework die Pipeline steuert und jedes Komponentenobjekt, also jede PAE, innerhalb der Pipeline nacheinander

Tabelle 5.5: Numerische Werte der Lageparameter für Operation „(2) EFXTokenizer.process“ aus dem Performance-Report

Lageparameter	PCM-Antwortzeit	Test-Antwortzeit
Minimum	10,89 ms	3,31 ms
25%-Quartil	10,89 ms	10,97 ms
Median	21,78 ms	15,51 ms
75%-Quartil	32,67 ms	24,72 ms
Maximum	65,33 ms	163,39 ms
Mittelwert	23,97 ms	22,16 ms

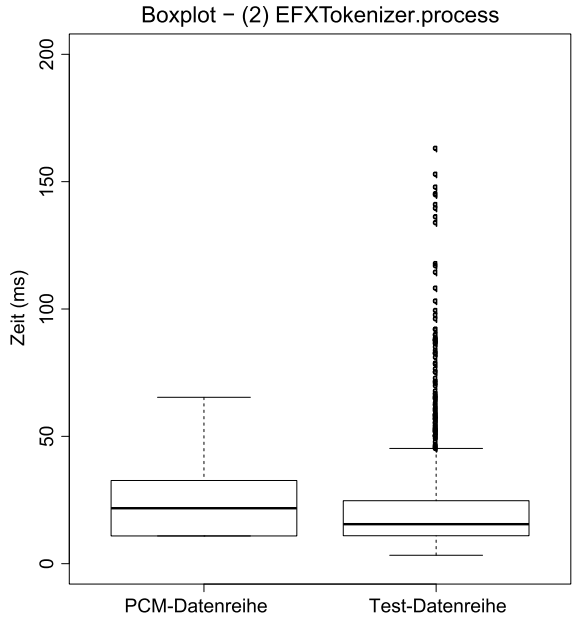

Abbildung 5.4: Box-Whisker-Plot für die Operation „2) EFXTokenizer.process“ aus dem Performance-Report

aufruft (vgl. Abbildung 5.3). Außerdem zeigt der Blame-Graph, ob Komponentenoperationen beschuldigt werden, zum festgestellten Fehler beizutragen. In dieser Fallstudie beschuldigt der Blame-Graph keine Komponentenoperationen, so dass die Operationen im Blame-Graph in Abbildung 5.5 nur als grüne Kopftrapeze und gelbe Hexagone dargestellt sind. Dabei gibt er für die Operation „EFXTokenizer.process“ an, dass die Operation nicht beschuldigt wird (die Box ist ein grünes Kopftrapez). Damit sind die Angaben im Blame-Graph konsistent mit denen aus dem Performance-Report.

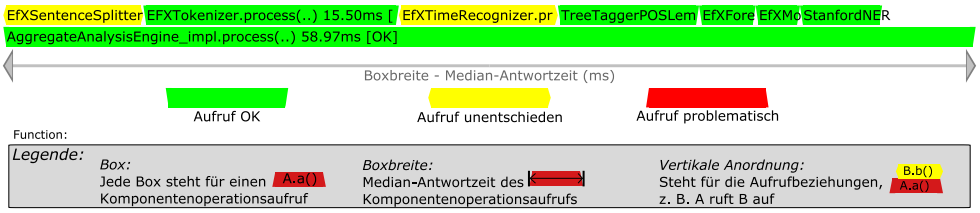

Abbildung 5.5: Der Blame-Graph für den zu untersuchenden Testfall in der UIMA-Fallstudie

5.2.4 Ergebnis interpretieren

Der Blame-Graph in Abbildung 5.5 umfasst zwei Operationen, bei denen der KS-Test keine Entscheidung getroffen hat und die deshalb als gelbe Hexagone dargestellt sind. Alle anderen Operationen halten ihren Kontrakt ein und sind dementsprechend als grüne Kopftrapeze abgebildet. Der Systemarchitekt muss also für die beiden unentschiedenen Operationen `EfXTimeRecognizer.process` und `EfXSentenceSplitter.process` prüfen, ob diese zum Fehler beitragen oder nicht. Dazu betrachtet er zunächst den Performance-Report. Bei den Box-Whisker-Plots sind bei beiden Diagrammen jeweils die Boxen für die Test-Datenreihe am unteren Ende der Box für die Performance-Vorhersage-Datenreihe. Das weist darauf hin, dass die jeweilige Test-Datenreihe etwas geringere oder zumindest in etwa gleiche Werte aufweist wie die zugehörige Performance-Vorhersage-Datenreihe. Die p-Werte der KS-Tests liegen bei 4,66% für die Operation `EfXSentenceSplitter.process` und bei 0,14% für die Operation `EfXTimeRecognizer.process`. Das heißt, wenn die p-Wert-Schwelle von 5% etwas niedriger gelegen hätte, wäre die Operation `EfXSentenceSplitter.process` nicht beschuldigt worden. Auch der p-Wert für die Operation `EfXTimeRecognizer.process` liegt relativ nah an der Grenze, wenn man bedenkt, dass der p-Wert bei über 1000 Messwerten je Datenreihe sehr schnell

gegen Null konvergiert. Diese Hinweise legen nahe, dass der Systemarchitekt sorgfältig prüfen sollte, ob der Fehler eine andere Ursache hat. Beispielsweise könnte der Fehler auch durch die Komponentenverteilung, die Auswahl der Hardware oder durch die Middleware-Auswahl bzw. -Konfiguration verursacht worden sein.

5.2.5 Analyse der Qualität der Kategorisierung

Im Folgenden wird die Qualität der Kategorisierung durch das primäre Entscheidungskriterium, also den KS-Test, analysiert. Der KS-Test kann unentschieden sein, anstatt das untersuchte Datenreihenpaar für eine Komponentenoperation klar als beschuldigt bzw. nicht beschuldigt zu kategorisieren. Daher soll im Folgenden untersucht werden, ob die Spanne in der der KS-Test unentschieden ist, sinnvoll verringert werden kann.

Die Qualität der Kategorisierung hängt primär von der p-Wert-Schwelle von 5% ab, die für die Fallstudien verwendet wurde. Um eine Analyse der Kategorisierungsqualität zu ermöglichen, wird in der Komponente `EfXTokenizer` ein Fehler injiziert. Der Fehler erhöht die Antwortzeit dieser Komponente, indem die Komponente zunächst für eine zufällige Zeit wartet, bevor die Verarbeitung des jeweiligen Eingabetextes beginnt. Die Wartezeit wird gleichverteilt aus dem Intervall [0 ms, t ms] gewählt, so dass die mittlere Wartezeit a den Wert $a = t/2$ hat. Die Performance-Blame-Analysis wird mit dem derartig modifizierten System mehrfach wiederholt. Die Analyse beginnt mit einer oberen Intervallgrenze $t = 2ms$. Die mittlere Wartezeit wird dann bei jeder Wiederholung um eine Millisekunde erhöht, indem die obere Intervallgrenze um zwei Millisekunden angehoben wird. Dies wird solange wiederholt bis der Blame-Graph die Komponente `EfXTokenizer` als beschuldigt darstellt.

Um die Qualität der Kategorisierung zu bewerten, wird die Spanne ermittelt, für die der KS-Test bei der Komponentenoperation `EfXTokenizer.process` keine Entscheidung trifft. Wie die Fallstudie gezeigt hat, wird die unmodifizierte Komponentenoperation nicht beschuldigt (vgl. Abbildung 5.5). Im Laufe der Analyse der Qualität der Kategorisierung wird dies aufgrund der je Iteration wachsenden Wartezeit in dieser Komponentenoperation zunächst in eine unentschiedene und schließlich in eine negative Bewertung umschlagen. Die ermittelte Spanne vermittelt dem Systemarchitekten einen Eindruck davon, wie stark sich Messwerte verändern müssen, damit auch der KS-Test zu einem anderen Ergebnis kommt.

In der Analyse der Qualität der Kategorisierung hat sich die Bewertung des KS-Tests, wie vermutet bis hin zur Beschuldigung der Komponentenoperation verändert. Dies ist in Abbildung 5.6 zu sehen. Auf der linken Seite werden zunächst die Datenreihe aus der PCM-Simulation sowie die ursprüngliche Testdatenreihe der Operation `EfXTokenizer.process` wiedergegeben (vgl. Abbildung 5.4). Die drei Datenreihen auf der rechten Seite sind Testdatenreihen, die für die Analyse der Qualität der Kategorisierung aufgezeichnet wurden. Bei diesen Testdatenreihen wird die Antwortzeit durch eine mittlere Wartezeit von 1 ms, 10 ms bzw. 11 ms erhöht. Nur die ursprüngliche Testdatenreihe wird klar nicht beschuldigt und ist als „grün“ gekennzeichnet. Bei den Testdatenreihen mit 1 ms bzw. 10 ms Wartezeit trifft der KS-Test keine Entscheidung, so dass sie mit „gelb“ kenntlich gemacht sind. Die Testdatenreihe mit 11 ms Wartezeit wird schließlich beschuldigt und ist somit als „rot“ dargestellt. In diesem Beispiel (s. Abbildung 5.6) ist der KS-Test also für eine Spanne von 10 ms unentschieden und beschuldigt die Operation bei einer im Mittel um 11 ms höheren Antwortzeit. Wenn man diese Spanne nun mit der ursprünglichen Testdatenreihe vergleicht, bei der das 75%-Quartil lediglich 24,72 ms (vgl. Tabelle 5.5) beträgt, sind 10 ms eine enorme Veränderung. Somit ist eine strengere p-Wert-Schwelle für den KS-Test wünschenswert. Wenn man die p-Wert-Schwelle von 5% auf 1% verringert, so bewirkt das, dass die Testdatenreihe mit 1 ms und 10 ms Wartezeit klar als nicht beschuldigt bzw. beschuldigt bewertet worden wären. Die p-Werte der anderen Testdatenreihen mit Wartezeit liegen bei höchstens 0,00088%. Die p-Wert-Schwelle derartig niedrig anzusetzen ist hingegen nicht sinnvoll, da dies die Wahrscheinlichkeit für falsche Bewertungen zu stark erhöht. Die neu gewählte p-Wert-Schwelle von 1% verringert die Spanne in der der KS-Test unentschieden ist von 10 ms auf 8 ms, was in diesem Beispiel einer 20% kleineren Spanne entspricht. Voraussichtlich schneidet die 1% p-Wert-Schwelle, die sich in diesem Beispiel bewährt hat, auch im Allgemeinen besser ab, als die initiale p-Wert-Schwelle von 5%. Wenn man die p-Wert-Schwelle weiter optimieren will, sollte man allgemein die Chance für unentschiedene Bewertungen und für fehlerhafte Bewertung in Abhängigkeit von der Anzahl der Messwerte berechnen. Daraus lassen sich verschiedene p-Wert-Schwellen für verschiedene Messwertzahlstufen ableiten, die zugleich für eine trennscharfe Kategorisierung und eine geringe Fehlerchance sorgen.

5.2.6 Ergebnisse der Fallstudie

In dieser Fallstudie wurde gezeigt, dass PBlaman auf Systeme angewandt werden kann, die auf dem UIMA-Framework beruhen und damit eine Pipes-

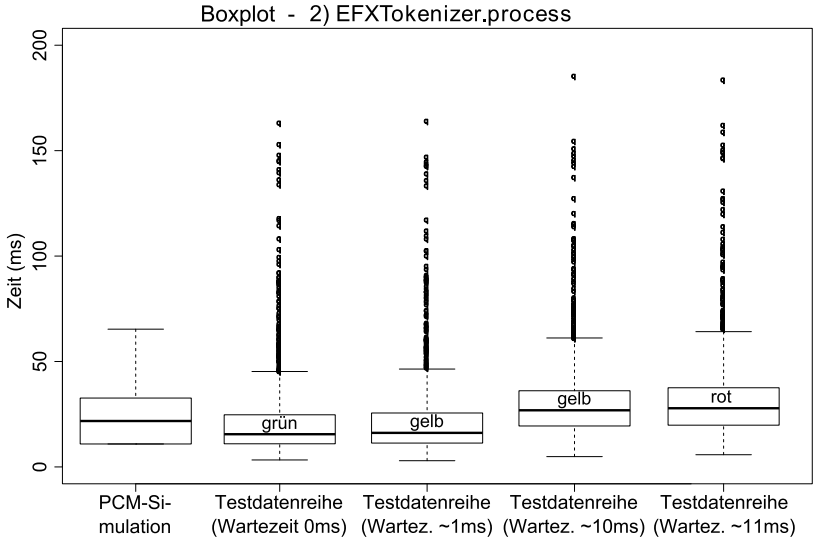

Abbildung 5.6: Die Abbildung zeigt die Analyse der Qualität der Kategorisierung durch den KS-Tests für die UIMA-Fallstudie. Die ursprünglichen beiden Datenreihen sind links abgebildet. Rechts sind drei Test-Datenreihen zu sehen, bei denen die jeweils angegebene mittlere Wartezeit die Antwortzeit erhöht hat. In jeder Box steht die Bewertung des KS-Tests. Grün steht für nicht beschuldigt, gelb für unentschieden und rot für beschuldigt.

and-Filters-Architektur haben. Da PBlaman die Performance-Blame-Analysis auf den Vergleich reduziert, ob die Testdatenreihe höhere Werte aufweist als die Datenreihe aus der Performance-Vorhersage, ist dies nicht überraschend. Damit kann PBlaman angewandt werden, wenn der Systemarchitekt die erforderlichen Werte im Test sammeln kann und eine zum zu untersuchenden Testfall äquivalente PCM-Instanz erstellen kann. Zudem kann man in dieser Fallstudie exemplarisch erkennen, dass bei der Betrachtung geänderter Eingabewerte nur kleine Anpassungen in der PCM-Instanz nötig sind. Bei den vorgestellten Kontrakten (vgl. Unterabschnitt 5.2.2) muss lediglich ein Eingabeparameter, nämlich der für die Anzahl der Tokens, im Verwendungsmodell geändert werden. Die Tatsache, dass das PCM Parameter in den verschiedenen Diagrammen erlaubt, vereinfacht die Anpassung an veränderte Eingabeparameter in unterschiedlichen Testfällen. Der Blame-Graph war in dieser Fallstudie weniger hilfreich als in der CoCoME-Fallstudie, weil die Aufrufhierarchie hier aufgrund der Pipes-and-Filters-Architektur sehr simpel ist. Die anderen Informationen, also die Blame-Bewertung und die Medianantwortzeit der Operationen, waren weiterhin nützlich für den Systemarchitekten. Außerdem wurde die Qualität der Kategorisierung durch das primäre Entscheidungskriterium, den KS-Test, beispielhaft an einer Komponentenoperation analysiert. Das Ergebnis war, dass der KS-Test in einem relativ großen Wertebereich keine Entscheidung trifft, so dass es empfehlenswert ist, die p-Wert-Schwelle von 5% auf 1% zu senken. Im untersuchten Beispiel hätte dies dazu geführt, dass der Wertebereich, in dem der KS-Test unentschieden ist, um 20% kleiner ausfällt.

5.3 Zusammenfassung

Die beiden Fallstudien zeigen, dass PBlaman sich auf verschiedene Systeme anwenden lässt. Dabei wurde ein System mit Schichtarchitektur (s. Abschnitt 5.1) und eines mit einer Pipes-and-Filters-Architektur (s. Abschnitt 5.2) untersucht. PBlaman reduziert die Frage, ob eine Komponentenoperation beschuldigt werden soll, auf die Frage, ob die Antwortzeitdatenreihe aus dem Test größere Werte beinhaltet als die aus der Performance-Vorhersage. Daher ist anzunehmen, dass der Ansatz sich auch für Systeme mit anderen Architekturstilen eignet. PBlaman stellt den angesprochenen Datenreihenvergleich in zwei Entscheidungsunterstützungsartefakten dar, dem Performance-Report und dem Blame-Graphen. Beide liefern konsistente Ergebnisse, es sei denn, der KS-Test ist unentschieden. In diesem Fall kann es vorkommen, dass die betreffende Operation anhand eines Vergleichs der Testdatenreihe mit der Performance-Vorhersage-Datenreihe aufgrund des Performance-Reports klar beschuldigt

bzw. nicht beschuldigt werden kann. Im Rahmen der Analyse werden dabei die Simulationsergebnisse einer PCM-Instanz als Performance-Vorhersage-Datenreihen genutzt. Dabei hat sich herausgestellt, dass das PCM die Arbeit durch die Berücksichtigung von Parametern erleichtert. So kann der Systemarchitekt eine PCM-Instanz beispielsweise einfach auf andere Eingabeparameter für einen bestehenden Testfall anpassen, indem er im Verwendungsmodell den entsprechenden Eingabeparameter verändert (vgl. Unterabschnitt 5.2.2). Bei der Anwendung von PBlaman muss der Systemarchitekt generell verschiedene manuelle Aufgaben erledigen. Dabei war das Sammeln der Daten mittels der Simulation einer zum Testfall äquivalenten PCM-Instanz auf der einen Seite sowie einer wiederholten Testausführung mit der passenden Instrumentierung auf der anderen Seite im Rahmen der Fallstudien bei Weitem am zeitaufwendigsten. Die Konfiguration des Auswertungsskripts nahm höchstens eine Stunde in Anspruch (vgl. Unterabschnitte 5.1.3 und 5.2.3). Hinzu kommt letztlich noch die manuelle Interpretation der Entscheidungsunterstützungsartefakte (vgl. Unterabschnitte 5.1.4 und 5.2.4). Schließlich wurde die Qualität der Kategorisierung durch das primäre Entscheidungskriterium im Rahmen der zweiten Fallstudie (s. Unterabschnitt 5.2.5) untersucht. Diese Analyse hat das Ziel, die Spanne sinnvoll zu verringern, in der der KS-Test unentschieden ist, statt eine Operation zu beschuldigen bzw. nicht zu beschuldigen. Die p-Wert-Schwelle, die entscheidet, ob ein KS-Test als bestanden gilt, wurde als Ergebnis der Analyse von 5% auf 1% gesenkt, wodurch der Wertebereich, in dem der KS-Test keine Entscheidung trifft, für die im Beispiel untersuchte Komponentenoperation um 20% verringert werden konnte.

Kapitel 6

Bewertung und Grenzen des PBlaman-Ansatzes

Der PBlaman-Ansatz wird in Abschnitt 6.1 zunächst mit den verbleibenden Anforderungen aus Unterabschnitt 3.4.3 bewertet und dann mit den verwandten Arbeiten aus Abschnitt 3.3 verglichen. In Abschnitt 6.2 werden die Grenzen des Verfahrens diskutiert. Dieser Abschnitt beschreibt die prinzipbedingten Grenzen des Verfahrens. Unzulänglichkeiten, die durch Verbesserungen am PBlaman-Ansatz beseitigt werden können, werden im Rahmen des Ausblicks in Abschnitt 7.2 diskutiert.

6.1 Bewertung des PBlaman-Ansatzes

Im Folgenden wird der PBlaman-Ansatz bewertet. Zunächst werden in Unterabschnitt 6.1.1 die verbleibenden Anforderungen (s. Unterabschnitt 3.4.3) durchgegangen. Dabei wird begründet, ob und wie PBlaman die jeweilige verbleibende Anforderung erfüllt. Da die verbleibenden Anforderungen die zuvor aufgestellten Anforderungen an Performance-Blame-Analysis-Ansätze (s. Abschnitt 3.2) und die Visualisierungsanforderungen (s. Unterabschnitt 3.3.4) aufgreifen, werden diese Anforderungen im Hinblick auf PBlaman ebenfalls diskutiert. In Unterabschnitt 6.1.2 wird PBlaman mit den verwandten Arbeiten (s. Abschnitt 3.3) verglichen. Dieser Vergleich orientiert sich wiederum an den verbleibenden Anforderungen. Hierbei soll herausgestellt werden, in welchen Punkten PBlaman eine Verbesserung des Stands der Technik darstellt.

6.1.1 Bewertung anhand der verbleibenden Anforderungen

Im Folgenden werden die einzelnen verbleibenden Anforderungen aus dem Unterabschnitt 3.4.3 durchgegangen. Dabei wird zunächst diskutiert, aus welchen Einzelanforderungen an Performance-Blame-Analysis-Ansätze die jeweilige verbleibende Anforderung besteht. Dann wird erläutert, ob die

Einzelanforderungen erfüllt sind und ob damit auch die jeweilige verbleibende Anforderung insgesamt erfüllt ist.

Anforderung Z1 (Testfallbezug)

Die erste verbleibende Anforderung Z1 fordert, dass sich die Analyse nach den Anforderungen A2 bis A5 jeweils, wie in Anforderung A1 gefordert, spezifisch auf den untersuchten Testfall bezieht. Die Analyse befasst sich dabei mit den Komponenten (Anforderung A2), dem System außerhalb der Komponenten (Anforderung A3), den Komponentenoperationen (Anforderung A4) sowie den einzelnen Aspekten der Komponentenverwendung (Anforderung A5).

Zunächst muss festgestellt werden, welche der Analyseanforderungen A2 bis A5 von PBlaman abgedeckt werden. In erster Linie analysiert PBlaman Komponentenoperationen (Anforderung A4), falls wenigstens eine Komponentenoperation beschuldigt wird, wird die zugehörige Komponente ebenfalls beschuldigt (Anforderung A2). Wird keine Komponentenoperation und damit auch keine Komponente beschuldigt, so kann man davon ausgehen, dass der Fehler außerhalb der Komponenten zu suchen ist, womit die Anforderung A3 abgedeckt ist. Einzelne Aspekte der Komponentenverwendung (Anforderung A5), also beispielsweise die Komposition oder die Komponentenverteilung, werden nicht explizit analysiert. Diese Aspekte können höchstens mit zusätzlichen Analysen auf den gesammelten Daten untersucht werden.

Bei PBlaman basieren die diskutierten Analysen auf Daten, die während der Ausführung des untersuchten Testfalls bzw. bei der Analyse einer äquivalenten PCM-Instanz anfallen. Damit ist der Testfallbezug nach Anforderung A1 klar gegeben. Obwohl PBlaman nicht das gesamte Analysespektrum abdeckt, erfüllt PBlaman die Anforderung Z1, da die Analysen einen klaren Testfallbezug haben.

Anforderung Z2 (Performance-Spezifikation und Spezifikationswerkzeuge)

Die verbleibende Anforderung Z2 fordert, dass die Analysen nach Anforderung A2 bis A5 jeweils eine geeignete Spezifikation für die erwartete Performance verwenden. Zudem soll es Werkzeuge geben, mit denen die Angaben zur erwarteten Performance verständlich erfasst und mit wenig Aufwand erstellt werden können.

Bei PBlaman wird das PCM (vgl. Abschnitt 2.3) für die Spezifikation eines Performance-Modells verwendet. Aus dem Modell wird dann die erwartete Performance durch Simulation des Modells (vgl. Unterabschnitt 2.3.6) abgeleitet. Die erwartete Performance wird also nur indirekt spezifiziert. Dies hat den Vorteil, dass man Änderungen im System oder im Testfall durch Änderungen im Performance-Modell nachvollziehen kann und dann daraus die neue erwartete Performance ableiten kann. Man braucht also nur wenig Wissen darüber, wie sich Änderungen des Systems auf die Performance auswirken. Die Auswirkungen der Änderungen werden stattdessen vom PCM durch die Simulation des Performance-Modells ermittelt. Die Simulation liefert dabei die geforderten Performance-Daten, wie sie in ähnlicher Form auch im Test anfallen. Falls noch weitere Daten oder Zusammenhänge während der Simulation beobachtet werden sollen, so kann die Simulation um Messfühler erweitert werden, die die geforderten Messwerte ermitteln. Zusammenfassend lässt sich feststellen, dass das PCM eine geeignete Spezifikationssprache für die erwartete Performance darstellt.

Die Anwender müssen beim PCM zunächst das Performance-Modell erstellen und dieses dann simulieren. Diese Arbeitsschritte werden von der Palladio-Workbench unterstützt. Die Palladio-Workbench erlaubt es die verschiedenen PCM-Diagramme grafisch zu erstellen. Das PCM unterstützt ein Rollenkonzept, bei dem jede Diagrammart von einer bestimmten Rolle, die über das nötige Fachwissen verfügt, erstellt wird (vgl. Abschnitt 2.3). Die Diagramme werden über Referenzen und Parameter lose gekoppelt, woraus PCM-Instanzen entstehen. Diese Art der Kopplung bietet beispielsweise die Möglichkeit, dasselbe Systemmodell an verschiedene Verteilungsmodelle zu koppeln. Es entstehen also verschiedene PCM-Instanzen, bei denen das einmal modellierte System auf verschiedenen Hardwarekonfigurationen eingesetzt wird. Zudem lassen sich mithilfe der Palladio-Workbench vollständige PCM-Instanzen vollautomatisch simulieren. Somit lässt sich sagen, dass die grafische Erstellung des Modells und die lose Kopplung der einzelnen Modellteile die Palladio-Workbench zu einem geeigneten und flexiblen Werkzeug machen, um PCM-Instanzen zu spezifizieren und zu simulieren.

Zudem haben Reussner et al. [57] auch den Modellierungsaufwand untersucht, der benötigt wird, um PCM-Instanzen zu erstellen. Dabei haben sie herausgefunden, dass der Aufwand PCM-Instanzen zu spezifizieren, etwa 25% höher ist als äquivalente Modelle mit dem Ansatz „Software Performance Engineering“ [68] zu erstellen. Für diesen Mehraufwand erhält man allerdings ein Modell, das man ganz oder in Teilen wiederverwenden kann, um beispielsweise verschiedene Testfälle mit unterschiedlichen Eingaben oder Hardwarekonfigurationen abzubilden. Zudem verteilt sich der Aufwand beim

PCM durch die Rollenaufteilung auf verschiedene Personen. Bei dem in dieser Arbeit betrachteten Szenario erstellen die Komponentenentwickler einen Teil des Modells und der Systemarchitekt den anderen Teil. Somit ergibt sich für den Systemarchitekten ein vertretbarer Aufwand, der sich mittels Wiederverwendung sogar nochmals auszahlen kann. Damit ist die verbleibende Anforderung Z2 erfüllt.

Anforderung Z3 (spezialisiert auf komponentenbasierte Systeme)

Die verbleibende Anforderung Z3 fordert, dass die Analysen nach den Anforderungen A2 bis A5 speziell auf die Besonderheiten komponentenbasierter Systeme eingehen. Dies ist bei PBlaman der Fall, da sich die Ermittlung der erwarteten Performance auf die Simulation eines komponentenbasierten Systems, das in einer PCM-Instanz modelliert wurde, stützt. Im Test werden ebenfalls die Antwortzeiten aller am zu untersuchenden Testfall beteiligten Komponentenoperationen gemessen. Auf dieser Grundlage entscheidet die Entscheidungsunterstützung von PBlaman, ob eine Komponentenoperation beschuldigt werden kann oder nicht. Damit ist die verbleibende Anforderung Z3 erfüllt.

Anforderung Z4 (Visualisierung der Entscheidung und Interpretationshilfen)

Die verbleibende Anforderung Z4 fordert, dass die Visualisierung nach den Anforderungen V5 und A11 eine klare Entscheidung darstellen soll. Dazu muss im Rahmen der Analyse auch eine klare Entscheidung getroffen werden. Zudem soll die Visualisierung nach der Visualisierungsanforderung V6 Interpretationshilfen bieten, die direkt der Performance-Blame-Analysis zu Gute kommen. Die betreffende Visualisierung soll nach Visualisierungsanforderung V7 auch kompakt sein.

PBlaman entscheidet im Rahmen der Entscheidungsunterstützung für jede Komponentenoperation, die im Testfall aufgerufen wird, ob diese langsamer ist als in der PCM-Simulation, dem Testorakel, vorhergesagt. Diese Entscheidung ist relevant für die Performance-Blame-Analysis und wird im Blame-Graph mittels der Boxfarben und -formen dargestellt. Zudem handelt es sich beim Blame-Graphen um eine kompakte Darstellung eines Aufrufkontextbaums, da er direkt von der kompakten Darstellung Flame-Graph abgeleitet ist (vgl. Unterabschnitt 3.3.4). Damit ist die verbleibende Anforderung Z4 erfüllt.

Anforderung Z5 (Visualisierung Übersicht und Detailansicht)

Laut der verbleibenden Anforderung Z5 soll die Visualisierung eines Performance-Blame-Analysis-Ansatzes sowohl eine Übersicht über die im Testfall beteiligten Entitäten als auch Detailansichten für die jeweils zugehörigen Datenreihen anbieten. In PBlaman gibt der Blame-Graph eine Übersicht über die am Testfall beteiligten Komponentenoperationen. Dabei werden im Blame-Graph die Aufrufhierarchie, die Medianantwortzeit sowie die Bewertung der Komponentenoperation im Rahmen der Performance-Blame-Analysis wiedergegeben. Zudem gibt es den Performance-Report, der für jede Komponentenoperation ein Box-Whisker-Plot enthält, in dem die Antwortzeitdatenreihe aus dem Test der aus der Performance-Vorhersage gegenübergestellt wird. Im Performance-Report sind außerdem die numerischen Werte der Lageparameter und die p-Werte für die KS-Tests enthalten, die die Visualisierungen nachvollziehbar machen sollen. Mit diesen beiden Entscheidungsunterstützungsartefakten erfüllt PBlaman auch die verbleibende Anforderung Z5.

Zusammenfassung

Zusammenfassend lässt sich sagen, dass PBlaman alle verbleibenden Anforderungen erfüllt. Dabei entspricht PBlaman nur fast allen Einzelanforderungen, die den verbleibenden Anforderungen zugrunde liegen. Die Anforderung A5 wird nicht abgedeckt, da PBlaman die Analyse der Aspekte der Komponentenverwendung nicht unterstützt. Um diese Anforderungen dennoch zu erfüllen, müsste der Systemarchitekt zusätzliche Analysen auf den gesammelten Daten anstellen. Insgesamt ergibt sich für PBlaman die folgende Bewertung:

Anforderung	Z1	Z2	Z3	Z4	Z5
PBlaman	+	+	+	+	+

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

6.1.2 Bewertung gegenüber verwandten Arbeiten

Im Folgenden wird PBlaman mit dem jeweils besten Ansatz aus den drei Kategorien der verwandten Arbeiten (vgl. Abschnitt 3.3) verglichen. Die verwandten Arbeiten sind danach kategorisiert, ob sie ein einzelnes Ablaufverfolgungsprotokoll analysieren (s. Unterabschnitt 3.3.1), zwei Ablaufverfolgungsprotokolle vergleichen (s. Unterabschnitt 3.3.2) oder ein Ablaufverfolgungs-

protokoll mit einer Spezifikation vergleichen (s. Unterabschnitt 3.3.3). Die besten Vertreter ihrer jeweiligen Kategorie sind: Magpie von Barham et al. [1], RanCorr von Marwede et al. [48] und Pip von Reynolds et al. [59]. Die vergleichende Diskussion orientiert sich dabei an den verbliebenen Anforderungen Z1 bis Z5 (s. Unterabschnitt 3.4.3) sowie den darin angesprochenen Einzelanforderungen an Performance-Blame-Analysis-Ansätze und Visualisierungen.

Kategorie: Analyse eines einzelnen Ablaufverfolgungsprotokolls

Zunächst wird PBlaman mit dem Magpie-Ansatz verglichen, der von Barham et al. [1] vorgestellt wurde. Der Ansatz Magpie untersucht ein Ablaufverfolgungsprotokoll auf Besonderheiten. Bei dieser Analyse verfolgt er zwei Aspekte. Zum einen können beliebige Messwerte ermittelt und in die Visualisierung aufgenommen werden. Die verschiedenen beobachteten Ereignistypen und ihre Beziehungen untereinander können über sogenannte Ereignisschemata beschrieben werden. Zum anderen führt Magpie automatisch eine Analyse der Ressourcenverbräuche aus, da die Ressourcenverbräuche immer erfasst werden. Die aufgezeichneten Ressourcenverbräuche werden in der Analyse von Scheduling-Artefakten befreit und in Cluster zusammengefasst. So kann Magpie selten auftretende Abläufe finden und zur Analyse vorschlagen.

Zwar ist die Analyse von Magpie testfallbezogen (verbleibende Anforderung Z1), aber nicht auf komponentenbasierte Systeme spezialisiert (verbleibende Anforderung Z3). Verglichen mit der PBlaman-Analyse kann Magpie nicht sicher bewerten, ob eine Komponentenoperation im Rahmen der Performance-Blame-Analysis zu beschuldigen ist (Anforderung A4). Dies hängt einerseits damit zusammen, dass Magpie nur selten vorkommende Abläufe finden kann und keine Spezifikation für die erwartete Performance in die Analyse einbezieht (verbleibende Anforderung Z2). Andererseits muss die Auswertung der Antwortzeit erst durch ein Ereignisschema spezifiziert werden. Daher ist eine spezialisierte Analyse mit Bezug darauf von Magpie auch nicht zu erwarten. Diese Argumentation lässt sich auf die Bewertung von Komponenten nach Anforderung A2 übertragen. Da Magpie nur seltenes Verhalten findet, lässt sich nie wirklich ausschließen, dass der Fehler nicht doch in einer Komponente zu suchen ist (Anforderung A3). Dagegen werden bei PBlaman nicht nur Komponentenoperationen direkt im Hinblick auf die Performance-Blame-Analysis bewertet, sondern dieses Ergebnis lässt sich auch auf die Komponenten übertragen und es kann auch ausgeschlossen werden, dass der Fehler in den Komponenten begründet liegt, wenn keine

Komponente beschuldigt wird. Magpie kann hingegen bei der Analyse der Komponentenverwendung nach Anforderung A5 punkten. Diese Analyse wird von PBlaman nicht direkt unterstützt. Bei Magpie lassen sich hingegen beliebige Daten, wie im Ereignisschema spezifiziert, auswerten. Diese Daten werden dann auch in der Visualisierung mit aufgenommen. Somit kann der Systemarchitekt anhand einer darauf aufbauenden Analyse verschiedene Aspekte der Komponentenverwendung untersuchen.

Außerdem erfüllt Magpie die verbleibende Anforderung Z5 nur teilweise, da der Ansatz zwar Übersichten über einzelne Abläufe aber keine Details zu den Ereignissen visualisiert. Dagegen bietet PBlaman mit dem Blame-Graphen eine Übersicht und mit dem Performance-Report eine Detailansicht. Wie schon angesprochen kann Magpie im Gegensatz zu PBlaman in der Analyse keine sicheren Entscheidungen bezüglich der Performance-Blame-Analysis treffen, so dass auch keine derartigen Entscheidungen visualisiert werden können (Anforderung V5 bzw. A11). Zudem bietet die in Magpie zur Visualisierung verwendete Gleisplanvisualisierung weder eine Interpretationshilfe (Anforderung V6) noch ist die Darstellung kompakt (Anforderung V7). Damit erfüllt Magpie die verbleibende Anforderung Z4 nicht. Insgesamt ergibt sich also die folgende Bewertung für diesen Ansatz:

Anforderung	Z1	Z2	Z3	Z4	Z5
Barham et al. [1]	+	-	-	-	o

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Kategorie: Vergleich von zwei Ablaufverfolgungsprotokollen

Als nächstes muss sich PBlaman gegen den Ansatz RanCorr von Marwede et al. [48] behaupten. RanCorr vergleicht das Ablaufverfolgungsprotokoll aus einer Testfallausführung mit einem Referenzablaufverfolgungsprotokoll. Also ist RanCorr genau wie PBlaman testfallbezogen (verbleibende Anforderung Z1). RanCorr berechnet Abweichungswerte, die wiedergeben, wie weit die Messwerte für die jeweilige Komponentenoperation (Anforderung A3), Komponente (Anforderung A2) oder den jeweiligen Hardwareknoten von der Referenz abweichen. Dabei ist die Analyse auf komponentenbasierte Systeme spezialisiert (verbleibende Anforderung Z3). Wie bei PBlaman ist auch RanCorr nicht direkt dazu in der Lage die Komponentenverwendung nach Anforderung A5 zu bewerten. Bei RanCorr lässt sich nur dann ausschließen, dass ein Fehler nach Anforderung A3 nicht im Bereich der Komponenten

liegt, wenn einige Bedingungen gelten. Dazu muss einerseits das Referenzablaufverfolgungsprotokoll fehlerfrei sein, da zuvor vorhandene Fehler sonst nicht gefunden werden. Andererseits sucht RanCorr allgemein nach Abweichungen, so dass auch Verbesserungen gefunden werden, die aber im Rahmen der Performance-Blame-Analysis uninteressant sind. Mit diesen Einschränkungen ist das Referenzablaufverfolgungsprotokoll, im Gegensatz zum PCM bei PBlaman, nicht besonders gut als Testorakel geeignet (verbleibende Anforderung Z2). Dagegen verwendet PBlaman mit dem PCM die geeignetere Spezifikation der erwarteten Performance.

Die Analyse von RanCorr stellt Abweichungen fest, trifft aber keine Entscheidung nach Anforderung A11. RanCorr gibt aber wohl eine Interpretationshilfe nach Anforderung V6. Die Darstellung ist zwar nicht sehr kompakt (Anforderung V7), aber die Interaktionsmöglichkeiten mit der Darstellung (Anforderung V8) machen dies wieder wett. Trotzdem muss auch bei der verbleibenden Anforderung Z4 PBlaman vorgezogen werden, da hier eine klare Entscheidung bezüglich der Performance-Blame-Analysis getroffen wird, die dann auch entsprechend dargestellt wird. RanCorr bietet eine Darstellung an, in der verschiedene Abstraktionsebenen interaktiv angezeigt werden können. Damit verfügt RanCorr über eine Übersichtsdarstellung, die optional auch sehr viele Details anzeigen kann, wodurch die Detaildarstellung schon integriert ist. Damit erfüllt RanCorr, wie auch PBlaman, die verbleibende Anforderung Z5. Damit ergibt sich die folgende Gesamtbewertung für RanCorr:

Anforderung	Z1	Z2	Z3	Z4	Z5
„RanCorr“ [48]	+	o	+	o	+

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Kategorie: Vergleich eines Ablaufverfolgungsprotokolls mit einer Spezifikation

Schließlich wird PBlaman auch mit dem Ansatz Pip von Reynolds et al. [59] verglichen. Beide Ansätze vergleichen das Ablaufverfolgungsprotokoll einer Testfallausführung mit einer Spezifikation der erwarteten Performance. Damit lässt sich feststellen, dass beide Ansätze eine testfallbezogene Analyse nach der verbleibenden Anforderung Z1 bieten. Die Analyse der Komponentenoperationen nach Anforderung A4 stützt sich bei beiden Ansätzen direkt auf den Vergleich der Antwortzeiten mit der spezifizierten erwarteten Antwortzeit.

Dies kann auf die Komponenten übertragen werden (Anforderung A2) und auf dieser Grundlage kann auch ausgeschlossen werden, dass der Fehler noch in den Komponenten zu suchen ist, falls keine Komponente beschuldigt wird (Anforderung A3). Der Ausschluss, dass der Fehler in den Komponenten zu suchen ist, erfordert im Ansatz Pip eine Performance-Spezifikation, die alle Aufrufe von Komponentenoperationen abdeckt. Für die Bewertung der Aspekte der Komponentenverwendung (Anforderung A5) können bei Pip mittels der dort genutzten Spezifikationssprache auch Tests für das Zusammenspiel verschiedener Komponenten formuliert werden. PBlaman bietet für solche Analysen keine direkte Unterstützung, was allerdings nicht an der Spezifikationssprache liegt, sondern an der derzeit eingeschränkten Analyse. Aufgrund der möglichen Analysen ist die in Pip verwendete Spezifikationssprache für die erwartete Performance im Sinne der verbleibenden Anforderung Z2 für die Performance-Blame-Analysis geeignet. Dabei ermöglicht Pip sogar eine etwas weitergehende Analyse als PBlaman.

Eine PCM-Instanz ist nach Anforderung A9 voraussichtlich mit weniger Aufwand zu erfassen als eine Pip-Spezifikation. Zum einen sieht das PCM eine arbeitsteilige Spezifikation mithilfe eines Rollenkonzepts und der Wiederverwendung der einzelnen Diagramme vor. Zum anderen werden im PCM lediglich die performancerelevanten Teile des System modelliert und nicht direkt die Performance der Operationsaufrufe in verschiedenen Aufrufkontexten. Die Performance der Operationen wird beim PCM von der Simulation berechnet. Bei Pip muss der Systemarchitekt dagegen genau überlegen, welche Aufrufbeziehungen bestehen und welche Performance dafür angemessen ist. Dabei hilft es auch nur bedingt, dass der Systemarchitekt eine Pip-Spezifikation anhand einer Testfallausführung erstellen kann. Denn er muss die dort beobachtete Performance für jede Aufrufbeziehung überprüfen und ggf. revidieren, um schon existierende Performance-Fehler aus der Spezifikation zu entfernen. Zudem muss der Systemarchitekt auch bei einer generierten Pip-Spezifikation die Vollständigkeit prüfen, da veränderte Eingabedaten oder andere Faktoren geänderte Abläufe nach sich ziehen können. Die Pip-Spezifikation muss dabei auch für alle ähnlichen Abläufe vollständig sein, da Pip keinerlei Wiederverwendung oder Parameter kennt. Ein Nachteil von Pip gegenüber des PBlaman-Ansatzes ist auch, dass seine Analyse gemäß der verbleibenden Anforderung Z3 nicht auf komponentenbasierte Systeme spezialisiert ist. Bei Pip muss der Systemarchitekt dafür Sorge tragen, dass wenigstens alle Komponentenoperationen und die Nachrichten, die sie verursachen, gemessen werden. Dann kann der Systemarchitekt in der Pip-Spezifikationssprache Annahmen über die Komponentenoperationen festlegen. Wenn nur die angesprochenen Daten vorliegen, benötigt der Sys-

temarchitekt den geringsten Aufwand zum Erstellen der Pip-Spezifikation, da er die weiteren Details nicht in der Spezifikation berücksichtigen muss. Andernfalls muss der Systemarchitekt die weiteren gemessenen Details in der Spezifikation entsprechend wiedergeben oder aber partiell spezifizierte Abläufe einsetzen. Beide Maßnahmen machen die Spezifikation schwerer zu lesen.

Pip entscheidet in seiner Analyse, welche Aspekte der Spezifikation im Testfall nicht eingehalten worden sind. Diese Entscheidung wird aber nicht, wie in der verbleibenden Anforderung Z4 gefordert, visualisiert. PBlaman stellt seine Entscheidungen dagegen im Blame-Graphen dar. Pip stellt die beobachteten Abläufe im Rahmen einer Baumdarstellung dar. Einzelne Abläufe können auch innerhalb einer Gleisplanvisualisierung visualisiert werden. Zudem können zu einzelnen Pip-Aufgaben verschiedene detaillierte Ansichten der gemessenen Datenreihe eingesehen werden. Damit erfüllt Pip die verbleibende Anforderung Z5, in der sowohl Übersichten als auch Detailansichten gefordert werden. Insgesamt ergibt sich die folgende Bewertung für Pip:

Anforderung	Z1	Z2	Z3	Z4	Z5
Reynolds et al. [59]	+	o/+	-	o	+

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

Zusammenfassung

Tabelle 6.1 fasst die Bewertung PBlamans gegenüber den verwandten Ansätzen zusammen. Demnach ist PBlaman von allen hier gegenübergestellten Verfahren am geeignetsten für die Performance-Blame-Analysis. Dabei fallen Magpie und RanCorr wegen Defiziten in der Analyse (nach verbleibender Anforderung Z2) prinzipbedingt heraus. Bei Magpie wird kein Testorakel bei der Analyse verwendet und das in RanCorr verwendete Referenzablaufverfolgungsprotokoll eignet sich nur bedingt als Testorakel. Pip eignet sich zwar für die Performance-Blame-Analysis, hat aber gegenüber PBlaman einige Nachteile. Diese bestehen darin, dass die Analyse nicht explizit auf komponentenbasierte Systeme zugeschnitten ist (nach verbleibender Anforderung Z3), der Aufwand zum Erstellen der Spezifikation der erwarteten Performance durch fehlende Rollentrennung und Wiederverwendung höher ist (nach verbleibender Anforderung Z2) und die Visualisierung die Entscheidungen nicht darstellt (nach verbleibender Anforderung Z4).

Tabelle 6.1: Zusammenfassende Bewertung der verbleibenden Anforderungen des jeweils besten Ansatzes der verwandten Arbeiten je Kategorie und PBlamans

Anforderung	Z1	Z2	Z3	Z4	Z5
Barham et al. [1]	+	-	-	-	o
„RanCorr“ [48]	+	o	+	o	+
Reynolds et al. [59]	+	o/+	-/o	o	+
PBlaman	+	+	+	+	+

Anforderung ist ...
... erfüllt (+) / ... mit Einschränkungen erfüllt (o) / ... nicht erfüllt (-)

6.2 Grenzen des PBlaman-Ansatzes

In seiner jetzigen Form hat der PBlaman-Ansatz eine Reihe von Einschränkungen. Im Folgenden werden einschränkende Annahmen und prinzipbedingte Einschränkungen des Ansatzes vorgestellt. Die Unzulänglichkeiten, die durch weitere Arbeit an PBlaman beseitigt werden können, werden in Abschnitt 7.2 im Rahmen des Ausblicks dargestellt.

Zunächst soll darauf hingewiesen werden, dass im Rahmen der vorgestellten Fallstudien Instrumentierungen eingesetzt wurden, die einen gewissen Overhead verursachen (vgl. Unterabschnitte 5.1.2 und 5.2.2). Der Overhead ist in den Messungen der Antwortzeit enthalten, so dass möglicherweise Komponenten beschuldigt werden, die ohne diesen Overhead nicht zu beanstanden wären. Dies lässt sich im Rahmen eines testbasierten Verfahrens nicht vermeiden und kann lediglich mit möglichst performanten Messframeworks abgemildert werden.

PBlaman ist derzeit auf die Analyse von Antwortzeitdaten von Komponentenoperationen beschränkt. Falls keine Abweichungen zwischen den Performance-Kontrakten und der Implementierung der Komponentenoperationen gefunden werden, nimmt der Ansatz an, dass der Fehler andernorts zu suchen ist, beispielsweise in der Komposition oder der Komponentenverteilung. Wenngleich die Entscheidungsunterstützung die anderen möglichen Fehlerbereiche nicht direkt abdeckt, so kann eine Analyse der gesammelten Daten dennoch auch über diese Bereiche Aufschluss geben. Dazu können die Daten mit anderen Ansätzen analysiert werden, wie zum den Ansätzen von Srinivas und Srinivasan [70] bzw. Marwede et al. [48].

PBlaman erfordert Performance-Kontrakte, die die Struktur der Komponenten, also ihre Operationen und Operationssignaturen, und Benutzung der Komponenten, also die Reihenfolge und Häufigkeit der Operationsaufrufe, korrekt wiedergeben. Wenn dies gegeben ist, beeinflusst die Qualität der Performance-Kontrakte zwar das Analyseergebnis, führt aber nicht zu ungültigen oder falschen Ergebnissen. Zudem muss der Systemarchitekt mit den angesprochenen Performance-Kontrakten für die Komponenten ein korrektes Systemmodell erstellen. Beim Ressourcenmodell muss der Systemarchitekt sicherstellen, dass sich die Ressourcenkapazitäten, genau wie die Ressourcenverbräuche in den Performance-Kontrakten, an einem projektinternen Standard orientieren. Die Modellierung anhand eines projektinternen Standards für Ressourcenverbräuche und -kapazitäten wird schon vom PCM gefordert, so dass sich diese Anforderung schon aus der Verwendung des PCM ergibt. In der Simulation müssen die angegebenen Ressourcenverbräuche und Ressourcenkapazitäten zudem zu realistischen Antwortzeiten führen, um einen Vergleich mit den Antwortzeiten aus dem Test zu ermöglichen.

Zu den bisher genannten Punkten muss der Systemarchitekt beim Aufbau der vollständigen PCM-Instanz auch bedenken, dass er ggf. auch Middleware-Komponenten im Modell mit abbilden muss. Andernfalls können Middleware-Komponenten nicht durch PBlaman bewertet werden und sie können unbemerkt die Analyseergebnisse der regulären Komponenten beeinflussen. Idealerweise sollten Middleware-Hersteller Performance-Kontrakte zusammen mit ihrer Middleware ausliefern. Dabei können sie die Unterstützung für Middleware-Komponenten im PCM [32] nutzen. Um Middleware-Komponenten mit PBlaman zu analysieren, müssen sie nicht nur in der PCM-Instanz modelliert werden, sondern es müssen auch die zugehörigen Operationsantwortzeiten im Test gemessen werden. Bei den in dieser Arbeit vorgestellten Fallstudien werden Middleware-Komponenten nur teilweise berücksichtigt. Die CoCoME-Fallstudie (s. Abschnitt 5.1) berücksichtigt keine Middleware-Komponenten, wogegen die UIMA-Fallstudie (s. Abschnitt 5.2) die UIMA-Pipeline als Komponentenobjekt modelliert, wobei es sich um eine Middleware-Komponente handelt.

PBlaman basiert darauf, die Unterschiede zwischen dem jeweiligen Performance-Kontrakt und der Implementierung einer Komponentenoperation zu bewerten. Wenn nun alle Performance-Kontrakte korrekt wären, würden sich die Daten aus dem Test und der Performance-Vorhersage nicht unterscheiden und PBlaman würde dies entsprechend wiedergeben. Das Ergebnis der Analyse wäre korrekt, jedoch nur wenig hilfreich. Generell hängt das Ergebnis von der Qualität und der Ausgestaltung der Performance-Kontrakte ab. Übermäßig einfach gehaltene Performance-Kontrakte führen wahrscheinlich nur

in speziellen Fällen zu realistischen Ergebnissen. Andernfalls können sich die Komponentenentwickler dazu entscheiden, absichtlich falsche Performance-Kontrakte zu liefern. Beispielsweise werden bei „sicheren Kontrakten“ in der Simulation immer höhere Antwortzeiten ausgewiesen, als im Test gemessen werden. PBlaman wird Komponenten mit solchen Kontrakten nicht beschuldigen, aber diese Kontrakte verringern auch die Chance, dass der Systemarchitekt sich gerade für Komponenten mit solchen Kontrakten entscheidet. Wenn noch andere Komponenten mit ähnlicher Funktionalität zur Verfügung stehen, die bessere Performance versprechen, ist es wahrscheinlich, dass der Systemarchitekt nicht zu den Komponenten mit sicherem Kontrakt greift. Im Gegensatz dazu gibt es auch „schmeichelhafte Kontrakte“, bei denen in der Simulation immer niedrigere Werte ausgewiesen werden, als im Test gemessen werden. Komponenten mit solchen Performance-Kontrakten werden, wegen ihrer vermeintlichen guten Performance, wahrscheinlich vom Systemarchitekten ausgewählt. Wenn es zur Performance-Blame-Analysis mit PBlaman kommt, werden diese Komponenten aber unweigerlich beschuldigt, zum Fehler beizutragen. Die Diskussion zeigt, dass PBlaman auch bei falschen Kontrakten die richtigen Ergebnisse liefert. Dennoch sind diese Ergebnisse nur hilfreich, wenn es nicht marktüblich ist, dass Komponentenentwickler ausschließlich entweder mit sicheren oder mit schmeichelhaften Kontrakten agieren.

Kapitel 7

Zusammenfassung und Ausblick

Zunächst wird diese Arbeit in Abschnitt 7.1 zusammengefasst. Dabei liegt das Augenmerk auf dem hier vorgestellten Ansatz „kontraktbasierte Performance-Blame-Analysis“ (PBlaman). Anschließend wird in Abschnitt 7.2 ein Ausblick gegeben. Zum einen werden konkrete Verbesserungsmöglichkeiten für PBlaman vorgestellt. Dabei werden die Bereiche Palladio-basierte Testfälle, Messung und Simulation, Visualisierung sowie Evaluierung berücksichtigt. Zum anderen wird über PBlaman hinausgehend ein erweitertes Verfahren zur Performance-Analyse diskutiert und auch ein Vorschlag zur dynamischen Komposition von Performance-Analyse-Systemen beschrieben.

7.1 Zusammenfassung

In dieser Arbeit wird der Ansatz „kontraktbasierte Performance-Blame-Analysis“ (PBlaman) [11, 12, 13] vorgestellt. PBlaman baut auf einem modellbasierten Entwicklungsprozess für komponentenbasierte Systeme [41] auf. Im Rahmen des Entwicklungsprozesses werden von verschiedenen Rollen Performance-Modelle mithilfe des „Palladio Component Model“ (PCM) [8, 56] erfasst. Mithilfe der Performance-Modelle kann schon früh im Entwicklungsprozess die Systemarchitektur evaluiert werden. Wenn im Rahmen der Systemtests dennoch ein Performance-Fehler auftritt, muss eine Performance-Blame-Analysis durchgeführt werden. Dabei wird untersucht, ob der Fehler innerhalb einer oder mehrerer Komponenten liegt oder ob er in der Komponentenverwendung gesucht werden muss.

Der Performance-Blame-Analysis-Ansatz PBlaman nutzt die PCM-Performance-Kontrakte der eingesetzten Komponenten als Testorakel. PBlaman prüft so für jede Komponentenoperation, ob sie zum Performance-Fehler beiträgt. Für jede Komponentenoperation werden zwei Antwortzeitdatenreihen, die einerseits aus dem Test und andererseits aus der Performance-Vorhersage stammen, miteinander verglichen. Wenn die Testdatenreihe höhere Antwortzeitwerte aufweist, werden die Komponentenoperation und damit auch die

Komponente beschuldigt, zum Fehler beizutragen. Dieser Vergleich ist nur dann valide, wenn die PCM-Instanz, die zur Performance-Vorhersage simuliert wurde, äquivalent zum Testfall ist, in dem die Testdatenreihe gemessen wurde. Um dies sicherzustellen, sollen die Tester Performance-Testfälle mithilfe der Testfallnotation „Palladio-basierte Testfälle“ erstellen. In dieser Testfallnotation wird schon für die Spezifikation des Testfalls eine PCM-Instanz konstruiert. Die PCM-Instanz kann dann später auch zur Simulation des Testfalls genutzt werden. Der Testfall spezifiziert mithilfe dieser PCM-Instanz, welche Last an das zu testende System zu richten ist, welche Komponentenobjekte das System bilden und wie diese Komponentenobjekte auf die Hardware verteilt werden. Mithilfe dieser PCM-Instanz können die Tester ein JUnit-Testskript generieren, aufgrund dessen sie den Testfall implementieren und ausführen können.

Der eigentliche Vergleich der Datenreihen aus dem Test und der Performance-Vorhersage erfolgt mithilfe der Lageparameterpunktschätzer und einem statistischen Test, dem Kolmogoroff-Smirnow-Test (KS-Test) [66]. Der KS-Test ist das primäre Entscheidungskriterium. In PBlaman kategorisiert er eine Komponentenoperation aufgrund des Datenreihenvergleichs als beschuldigt, nicht beschuldigt, oder unentschieden. Diese Kategorisierung wird innerhalb eines Blame-Graphen mithilfe der Farbe und Form des jeweiligen Komponentenoperationselements dargestellt.

Der Blame-Graph stellt einen Aufrufkontextbaum auf Ebene der Komponentenoperationen dar. Dabei visualisiert er neben der Aufrufhierarchie auch für jede Operation die Blame-Kategorisierung nach KS-Test und die Medianantwortzeit aus dem Test. Der Systemarchitekt kann mithilfe des Blame-Graphen also nicht nur die beschuldigten Komponentenoperationen erkennen, sondern er kann auch anhand der anderen dargestellten Informationen festlegen, in welcher Reihenfolge die Komponentenoperationen näher untersucht werden sollen. Zur näheren Untersuchung der Komponentenoperationen kann der Systemarchitekt das zweite Entscheidungsunterstützungsartefakt nutzen, den Performance-Report.

Der Performance-Report stellt für jede Operation beide Antwortzeitdatenreihen in einem Box-Whisker-Plot gegenüber. Im Box-Whisker-Plot werden unter anderem die Quartile (also 25%-Quartil, Median und 75%-Quartil) dargestellt, die das sekundäre Entscheidungskriterium darstellen. Zudem umfasst der Performance-Report auch die numerischen Werte für die Quartile, die weiteren Lageparameter und auch die numerischen KS-Test-Ergebnisse. Die numerischen Werte helfen dem Systemarchitekten, die Entscheidung des KS-Tests nachzuvollziehen und auch das Box-Whisker-Plot lässt sich mit den

numerischen Werten der dort dargestellten Lageparameter besser einschätzen.

Die Anwendbarkeit von PBlaman wurde in zwei Fallstudien demonstriert. Dabei wurde auch gezeigt, dass die Ergebnisse des Blame-Graphen und des Performance-Reports konsistent sind. Es kann lediglich für solche Komponentenoperationen Ausnahmen geben, bei denen das im Blame-Graph dargestellte KS-Test-Ergebnis unentschieden ist. Die erste Fallstudie [12] hat das „Common Component Modeling Example“ (CoCoME) [33] untersucht. Das CoCoME realisiert ein Handelssystem einer Supermarktkette, bei dem eine Schichtarchitektur zum Einsatz kommt. Darüber hinaus wurden die Ergebnisse aus der ersten Fallstudie auch in der zweiten Fallstudie bestätigt. In der zweiten Fallstudie [13] wurde ein Informationsextraktionssystem auf Basis des Apache-UIMA-Frameworks untersucht. Das System sucht in einer Sammlung von unstrukturierten Texten Aussagen über die finanzielle Entwicklung von Unternehmen über die Zeit. Dabei realisiert das System eine Pipes-and-Filters-Architektur.

Da die Anwendung von PBlaman in beiden Fallstudien erfolgreich war, liegt es nahe, dass der PBlaman-Ansatz generell für Systeme mit dem jeweils untersuchten Architekturstil verwendet werden kann. Darüber hinaus ist PBlaman voraussichtlich sogar unabhängig von der Architektur des zu untersuchenden Systems anwendbar. Dies ist darin begründet, dass PBlaman die Entscheidung, ob eine Komponentenoperation beschuldigt wird, darauf zurückführt, ob die Antwortzeitdatenreihe aus dem Test höhere Werte beinhaltet als die Datenreihe aus der Performance-Vorhersage. Für die Anwendbarkeit von PBlaman ist also nicht die Architektur das Entscheidendste, sondern ob die Antwortzeitdatenreihen sowohl im Test des komponentenbasierten Systems als auch in der Performance-Vorhersage in einem äquivalenten Szenario mit Bezug auf denselben Messkontext ermittelt werden können.

In der zweiten Fallstudie wurde außer der Anwendbarkeit PBlamans auch die Qualität der Kategorisierung durch das primäre Entscheidungskriterium, den KS-Test, am Beispiel einer Komponentenoperation untersucht. Diese Analyse hat das Ziel, den Bereich, in dem der KS-Test eine Komponentenoperation als unentschieden bewertet, zu verringern und stattdessen häufiger eine klare Entscheidung, also „beschuldigt“ bzw. „nicht beschuldigt“, herbeizuführen. Das Ergebnis der Analyse ist, die p-Wert-Schwelle von 5% auf 1% zu senken. Die p-Wert-Schwelle wird dazu genutzt, um die Nullhypothese der KS-Tests zu verwerfen. Aus der gesenkten p-Wert-Schwelle ergab sich in dem untersuchten Beispiel eine um 20% verringerte Spanne, in der der KS-Test die Komponentenoperation als unentschieden bewertet.

7.2 Ausblick

Im Folgenden werden zunächst Verbesserungen für den PBlaman-Ansatz erläutert. Dabei werden die Bereiche Testfälle, Messung und Simulation, Visualisierung sowie Evaluierung berücksichtigt. Im Anschluss werden noch zwei über PBlaman hinausgehende Ansätze diskutiert. Zum einen soll die Performance-Analyse durch die Berücksichtigung von Zusatzdaten verbessert werden. Zum anderen sollen Performance-Analyse-Systeme nach den individuellen Vorgaben des Systemarchitekten dynamisch komponiert werden.

PBlaman-Erweiterungen im Bereich Testfälle

Wenn alle Messpunkte eines Systems in der PCM-Instanz vorliegen und auch eine Zuordnung der Komponenten zu ihrer jeweiligen Implementierung vorhanden ist, kann die PCM-Instanz auch dazu genutzt werden, um den Messcode für den Test des komponentenbasierten Systems zu generieren. Dies ist insbesondere dann einfach, wenn der Messcode beispielsweise durch aspektorientierte Programmierung vollständig vom Quelltext der Komponente getrennt werden kann. Hier ist eine erweiterbare Generierung des Messcodes erforderlich, bei der Messcode für verschiedene Messframeworks, wie BTrace oder Kieker, durch unterschiedliche Generierungsregeln erstellt werden kann.

Die Palladio-basierten Testfälle könnten derart erweitert werden, dass die erwartete Performance in den Testfällen spezifiziert werden kann. Zudem könnte die Umsetzung als JUnit-Testskript soweit verbessert werden, dass es für den Lastgenerator eine zusätzliche Koordinierungsinstanz gibt. Die Koordinierungsinstanz kann beispielsweise nach jeweils einigen Sekunden einen weiteren Knoten für die Lastgenerierung aktivieren. Dies könnte beispielsweise eine Portierung des Testskripts nach JMeter[1] leisten. Außerdem können Palladio-basierte Testfälle mit einem Ansatz kombiniert werden, bei dem die Vorbedingungen eines Testfalls durch die Ausführung verschiedener Komponentenoperationen automatisch hergestellt wird [21, 27].

PBlaman-Erweiterungen im Bereich Messung und Simulation

In Zukunft könnte PBlaman noch weiter verbessert werden. Zunächst könnten die bereits angesprochenen Messfühler für die Palladio-Workbench bzw.

[1] s. JMeter-Webseite: https://jmeter.apache.org (zuletzt besucht am 02.08.2014)

das `SensorFramework` implementiert werden. So kann die Performance-Vorhersage des PCM die Antwortzeiten im Zusammenhang mit dem erforderlichen Messkontext, insbesondere dem Stacktrace, erfassen.

Außerdem könnte PBlaman für die Performance-Metriken Durchsatz und Ressourcenverbrauch erweitert werden. Hierzu muss zunächst die jeweilige Metrik definiert werden, so dass der Bezug zu den beobachtbaren Ereignissen, wie Aufruf oder Rückgabe aus einer Operation, und der Messkontext genau festgelegt sind. Diese Metriken müssen dann im Test und in der Performance-Vorhersage, ggf. wieder mit einem eigenen Messfühler, erfasst werden. Wenn der Messkontext auch den Stacktrace, die Thread-ID sowie die jeweilige Bezugszeit der Metrik beinhalten, kann die Analyse voraussichtlich ohne Änderungen mit den Durchsatz- oder Ressourcenverbrauchmesswerten arbeiten und wie gehabt den Blame-Graph und den Performance-Report erstellen. Diese Voraussetzungen werden zum Beispiel von der CPU-Zeit eines Komponentenoperationsaufrufs oder dem Durchsatz eines Operationsaufrufs erfüllt. Dagegen ist dies für den prozentualen Ressourcenverbrauch schwierig, da der Zusammenhang zwischen dem prozentualen Ressourcenverbrauch und dem Stacktrace häufig nicht klar ist.

PBlaman könnte auch dadurch verbessert werden, dass es die Nettoantwortzeit der Komponentenoperationen berücksichtigt. Derzeit handelt es sich bei den Antwortzeitmessungen einer Operation um Messungen, die die Antwortzeiten der von dort aufgerufenen Operationen mit beinhalten. Daher kann es vorkommen, dass mehr Operationen als nötig beschuldigt werden, da die hohe Antwortzeit einer aufgerufenen Operation dazu führt, dass auch die aufrufende Operation beschuldigt wird. Um PBlaman auf Netto-Antwortzeiten umzustellen, muss auf Testseite die Instrumentierung einen klaren Zusammenhang zwischen aufrufender und aufgerufener Operation erfassen. Dann lässt sich anschließend im Rahmen der Datenvorbereitung die Netto-Antwortzeit berechnen. Analog dazu muss auch ein entsprechender Messfühler für die PCM-Simulation implementiert werden.

Zudem könnte PBlaman dahingehen erweitert werden, dass die Messung der Antwortzeit weiter verfeinert wird. Zum Beispiel kann die Antwortzeit in die CPU-Zeit und verschiedene Wartezeiten unterteilt werden. Einige Wartezeiten werden sich wahrscheinlich nur schwer im Test oder der Performance-Vorhersage erfassen lassen, da sie beispielsweise vom Scheduling des Betriebssystems verursacht werden. Bei anderen Wartezeiten ist dies nicht der Fall, z. B. bei Netzwerklatenzen. Netzwerklatenzen können direkt im PCM modelliert werden. Das PCM ist ebenfalls dazu geeignet Wartezeiten zu erfassen, die durch Synchronisationsmechanismen der Middleware, wie bei-

spielsweise Thread-Pools, verursacht werden. Die angesprochenen positiven Beispiele dürften sich auch im Test messen lassen. Darüber hinaus könnten auch andere Elemente, die schon im PCM vorgesehen sind, in der Analyse berücksichtigt werden. Das betrifft beispielsweise die Fehlerbehandlung innerhalb von Komponentenoperationen. Aber es wäre auch möglich die Antwortzeiten auf Blöcke innerhalb einer Komponentenoperation zu beziehen. Dazu wäre allerdings, wie von Groenda [31, 30] beschrieben, ein Abgleich der PCM-Instanz mit dem Quelltext der Komponenten nötig. Darüber hinaus kann PBlaman auch erweitert werden, damit es komponierte Komponenten, die intern aus verschiedenen anderen Komponenten bestehen, direkt unterstützt. Die PCM-Simulation kann bereits die nötigen Messwerte liefern. Für die Performance-Blame-Analysis müssen die Testmessungen auf dieser Ebene vorliegen, es müssten also auch alle internen Komponentenoperationsaufrufe gemessen werden. Zudem muss die Zuordnung der Stacktraces zu den PCM-Elementen auf diese Schachtelungsebene erweitert werden. Schließlich sollte die Schachtelung auch bei der Visualisierung berücksichtigt werden.

PBlaman-Erweiterungen im Bereich Visualisierung

Im Bereich der Visualisierung könnte im Performance-Report das KS-Test-Ergebnis auch im Rahmen des Box-Whisker-Plots wiedergegeben werden, indem die Boxen entsprechend eingefärbt werden. Im Blame-Graphen könnte die Nettoantwortzeit je Komponentenoperation aufgenommen werden, z. B. als schmaler grauer Balken über dem jeweiligen Trapez oder Hexagon. Möglicherweise ist es auch sinnvoll, den Blame-Graphen mit verschiedenen Sichten zu erweitern, indem das Overlay-Konzept aus der Softwarekartografie [45] umgesetzt wird. So könnte eine Sicht die Medianantwortzeit und eine andere die Nettoantwortzeit auf dem Aufrufkontextbaum einblenden.

Außerdem zeigt der Blame-Graph die Medianantwortzeit der jeweiligen Komponentenoperationen an, ohne auch die Anzahl der Aufrufe anzuzeigen. Um einschätzen zu können, wie viel Zeit die Operation insgesamt verbraucht, ist es hilfreich auch die Anzahl der Aufrufe der jeweiligen Komponentenoperation zu kennen. Dies ließe sich leicht in den Blame-Graphen mit aufnehmen.

Derzeit zeigt der Blame-Graph weder Prozess- noch Hostgrenzen an, was zu einer inkonsistenten Darstellung der Aufrufhierarchie führt. Gregg [29] hat vorgeschlagen, für solche Abgrenzungen in Flame-Graphen [28] weitere Boxen einzuschieben, die diese Grenzen verdeutlichen. Diese Maßnahme lässt sich auf Blame-Graphen übertragen. Einerseits müssen die Daten dazu jeden Messwert im Kontext von Host und Prozess betrachten. Andererseits

müssen die Daten die Kopplung des Aufrufs einer Komponentenoperation in einem Prozess mit dessen Ausführung in einem anderen Prozess erfassen. Während die Erfassung des Host- und Prozesskontext unproblematisch ist, ist die Kopplung technisch schwierig zu messen. Dies zeigt sich auch in der Datenerfassung in den in dieser Arbeit dargestellten Fallstudien. In der CoCoME-Fallstudie können die Messwerte zwar nach Host und Prozess abgegrenzt werden, aber die Kopplung wird von der Instrumentierung nicht unterstützt. Dies führt dazu, dass intern aufgerufene Operationen, die in einem anderen Prozess ausgeführt werden, wie Operationen dargestellt werden, die direkt vom Testtreiber aufgerufen werden (s. Unterabschnitt 5.1.5). Die Kieker-Instrumentierung in der zweiten Fallstudie kann Host und Prozess zu jeder Antwortzeitmessung festhalten. Zudem lässt es Kieker zu, die für die Kopplung erforderlichen Daten zu erfassen. Die tatsächliche Kopplung innerhalb eines Messfühlers muss aber auch hier erst manuell erstellt werden. Ein solche Kopplung wurde von van Hoorn et al. [73] schon auf Grundlage von Kieker realisiert. Dabei mussten sie die übergebenen Parameter der beobachteten Aufrufe um eine ID für die jeweilige Aufrufbeobachtung erweitern, damit sowohl für den Aufrufer und auch den Aufgerufenen ein einheitliches Identifikationsmerkmal existiert. Dies ist eine Maßnahme, die bei komponentenbasierten Systemen häufig schwierig zu realisieren ist, da der zusätzliche Parameter zumeist nachträglich in fertigen Komponenten implementiert werden muss.

PBlaman-Erweiterungen im Bereich Evaluierung

Im Bereich der Evaluierung könnte eine empirische Studie die Frage klären, ob die vorgeschlagenen Visualisierungen angemessen und benutzerfreundlich sind. Für den gesamten PBlaman-Ansatz wäre eine weitere Evaluierung an einem industriellen Beispiel wünschenswert. Zudem wäre auch eine Fallstudie mit einem ereignisbasierten System sinnvoll. Die PCM-Unterstützung für ereignisbasierte Systeme sowie die PBlaman-Analyse mithilfe des Datenreihenvergleichs sprechen dafür, dass PBlaman auch in diesem Bereich anwendbar ist. Dennoch stellt die Übertragung auf diesen sehr andersartigen Architekturstil eine Herausforderung dar. Zunächst muss der Begriff Komponentenoperation auf die Ereignisbehandlungsroutinen (engl. „event handler“) übertragen werden. Aufgrund dieser Definition muss die Antwortzeitmetrik fußen, die wiederum für die Datenermittlung in Test und Simulation maßgeblich ist. Ggf. muss also eine spezielle Instrumentierung für ereignisbasierte Systeme zum Einsatz kommen. Schließlich könnte noch untersucht werden, welche Arten von Fehlern durch PBlaman gefunden werden können.

Erweiterter Performance-Analyse-Ansatz

Bei der Analyse der Performance-Daten ist PBlaman auf die Antwortzeiten und die Aufrufhierarchie beschränkt. Die Umgebung wird nur im Zuge der Middleware-Komponenten beachtet und auch die Geschäftsdaten fließen nicht ein. Beide Aspekte sind jedoch wichtig. Im Rahmen der Umgebung ist das Verhalten des unterliegenden Betriebssystems bzw. der unterliegenden „Java Virtual Machine“ wichtig für die Performance der dort ausgeführten Komponenten. Ähnlich wichtig für die Performance ist auch, welche Daten im Rahmen der Komponentenausführung verwendet werden. Dabei kann es wichtig sein, einzelne Aufrufe einem Benutzer oder aber einer Session zuordnen zu können oder auch die damit zusammenhängenden Geschäftstransaktionen zu betrachten. Diese Daten sind häufig ohnehin vorhanden, weil diese Daten teils aufgrund von gesetzlichen Vorschriften [75] vorgehalten werden müssen.

Die in dieser Arbeit betrachteten verwandten Arbeiten bieten diese Funktionalität genauso wenig wie PBlaman. Dabei gibt es auf der einen Seite Ansätze, die wie PBlaman auf die Analyse von Aufrufbäumen spezialisiert sind, z. B. der Ansatz RanCorr von Marwede et al. [48]. Auf der anderen Seite gibt es Ansätze die Ereignislisten untersuchen, wie z. B. der Ansatz von Jiang et al. [38] oder der Ansatz Magpie [1]. Dabei können bei Magpie sogar als Paare zusammenhängende Ereignisse spezifiziert werden, wie z. B. Start und Ende eines Aufrufs, aber diese Information wird nicht ausreichend in der Analyse berücksichtigt.

Eine Lösung, die die Zusatzinformationen konsequent mit einbezieht, sollte die in Abbildung 7.1 genannten Schritte beinhalten. Je nachdem welches System untersucht wird, muss man unterschiedliche Zusatzdaten berücksichtigen. Die Zusatzdaten müssen zunächst gesichtet werden und dann sollte eine Datenstruktur für diese Daten aufgebaut werden. Im einfachsten Fall nimmt man lediglich alle verfügbaren Daten in einer Liste auf. Diese Datenstruktur lässt sich jedoch noch erweitern, indem beispielsweise mehrere Listen von Ereignissen für jede Ereigniskategorie angelegt werden, z. B. eine für Geschäftsereignisse, eine für Systemereignisse und eine für Ereignisse in der „Java Virtual Machine“. Zudem können auch hier Baumstrukturen nötig sein, um hierarchische Abhängigkeiten von Ereignissen betrachten zu können.

Die Datenstruktur mit den Zusatzdaten muss dann den Performance-Daten zugeordnet werden, d. h. die Daten sollten in den Aufrufbaum integriert

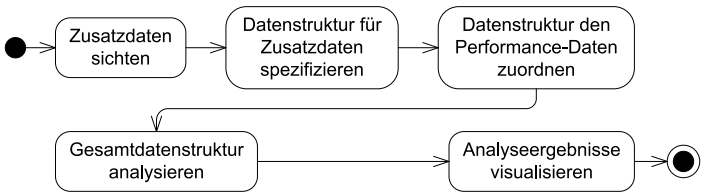

Abbildung 7.1: Schritte für eine erweiterte Performance-Analyse mit Zusatzdaten, wie z. B. Betriebssystem- oder Geschäftsereignissen

werden. Diese Art der Integration sieht den Aufrufbaum als primäre Datenstruktur. Es kann aber auch sinnvoll sein, die Zusatzdaten als primäre Datenstruktur zu nutzen. Die Zusatzdaten sind teils ohnehin vorhanden, wogegen die Performance-Daten erst durch einen zusätzlichen Testlauf ermittelt werden.

Nun wird die zuvor erstellte kombinierte Datenstruktur analysiert. Dies erfordert voraussichtlich neue Algorithmen, die mit den unterschiedlichen Daten und Datenrepräsentationen aus den verschiedenen Quellen umgehen können. Dabei kann es sinnvoll sein, je nach Vollständigkeit des Datenbestands unterschiedliche Analysen zu ermöglichen, so dass zunächst eine initiale Analyse möglich ist, die später verfeinert werden kann. Idealerweise sollte automatisch die passendste Analyse nach Vollständigkeit des Datenbestands gewählt werden.

Zuletzt müssen diese Analyseergebnisse sinnvoll aufbereitet und visualisiert werden. Da hier verschiedene Zusatzinformationen zu den Performance-Daten hinzukommen sollen, bietet es sich an, verschiedene Sichten zu generieren, die ggf. mithilfe des Overlay-Konzepts aus dem Bereich der Softwarekartografie [45] integriert sind. Beispielsweise kann ein Aufrufkontextbaum als Grundvisualisierung dienen, auf dessen Darstellung dann verschiedene Zusatzinformationen eingeblendet werden können.

Dynamisch komponierte Performance-Analyse-Systeme

Um eine Performance-Analyse für ein bestimmtes System zu realisieren, muss der Systemarchitekt über viel Wissen verfügen. Er muss sich zunächst Gedanken machen, welche Metriken mit welchem Messkontext wichtig für die Analyse sind. Dazu setzt er beispielsweise das Verfahren „Goal-Question-Metric“ (GQM) [2] ein, bei dem er ausgehend von Zielen, die erforderlichen

Metriken ableitet. Dann muss er Messfühler beschaffen (im Sinne einer komponentenbasierten Entwicklung (s. Unterabschnitt 2.2.2)), die die erforderlichen Messdaten liefern. Die Messfühler müssen im Rahmen eines Werkzeugs definiert werden, dass die Daten in performanter Weise sammeln kann. Die Messdaten müssen dann wiederum in einem Analyseschritt aufbereitet und schließlich visualisiert werden. Diese Infrastruktur zusammenzustellen, erfordert einen hohen Aufwand und Know-how in den genannten Bereichen.

Die in dieser Arbeit dargestellten verwandten Ansätze und PBlaman stellen spezialisierte Analyseverfahren dar. Bei diesen Verfahren wird eine bestimmte Analyseaufgabe auf der Grundlage eines definierten Satzes an Metriken mit einer festgelegten Auswertung und Analyse gelöst. Dabei definieren einige Verfahren auch eine feste Lösung zum Ermitteln der zugehörigen Messdaten. Andere Verfahren stellen diesbezüglich nur eine beispielhafte prototypische Implementierung bereit. Diese spezialisierten Verfahren lassen sich nur mit einigem Aufwand in ein benutzerdefiniertes Analyseverfahren einbeziehen. Dabei ließe sich sowohl die Messdatenermittlung als auch die Visualisierung austauschbar gestalten, da der Kern der genannten Verfahren meist die Analyse ist. Es gibt bereits Ansätze, die schon eine konsequente Trennung der Bereiche Messung und Auswertung vorsehen. Zwei entsprechende Ansätze sind Kieker [74, 73] und das Software Performance Cockpit [79, 80]. Hier geben die Auswertungen an, welche Metriken sie benötigen. Dieser Zusammenhang wird aber in diesen beiden Ansätzen nicht ausreichend genutzt, sondern die Analysen resultieren nur dann im gewünschten Ergebnis, wenn ein Messdatensatz die geforderten Metriken enthält. Ob die Analyse sinnvoll anwendbar ist, wird also erst zum Zeitpunkt ihrer Ausführung oder sogar erst bei der Begutachtung der Analyseergebnisse ersichtlich.

Wünschenswert wäre, wenn der Systemarchitekt lediglich aus einem Katalog an Auswertungen auswählt und das Performance-Analyse-System dann mittels eines Analysesystemgenerators komponiert wird. Ein Performance-Analyse-System besteht üblicherweise aus den Komponenten Instrumentierung und Messung, Analyse sowie Visualisierung (s. rechte Seite in Abbildung 7.2). Dies stimmt auch mit der üblichen Vorgehensweise bei der Performance-Analyse nach DeRose [17] (vgl. Abschnitt 3.1) überein. Der Systemarchitekt soll zunächst eine oder mehrere Analysen aus einem Katalog (s. Abbildung 7.2) auswählen, da die Analysen es ermöglichen bestimmte Untersuchungsfragen zu klären. Sozusagen kann mithilfe des Analysesystemgenerators der Ansatz „Goal-Question-Metric“ in den Ansatz „Goal-Question-Analysis“ überführt werden. Mit der Auswahl eines Analyseverfahrens soll der Analysesystemgenerator zugleich eine mögliche Menge von Messfühlern sowie Visualisierungen präsentieren, die zu den gewählten Analysen

passen. Das heißt, dass die Abhängigkeiten zwischen der Analyse und den Visualisierungen sowie die Abhängigkeit zwischen den Messfühlern und der Analyse jeweils explizit in einem Abhängigkeitsmodell (s. Abbildung 7.2) spezifiziert werden müssen. Wenn der Systemarchitekt alle Elemente für die Performance-Analyse gewählt hat, wird die Komponist-Komponente (s. Abbildung 7.2) innerhalb des Analysesystemgenerators die gewählten Elemente zu einem Performance-Analyse-System zusammenstellen. Der Analysesystemgenerator besteht also zusammengefasst aus den Komponenten Katalog, Komponist und Abhängigkeitsmodell.

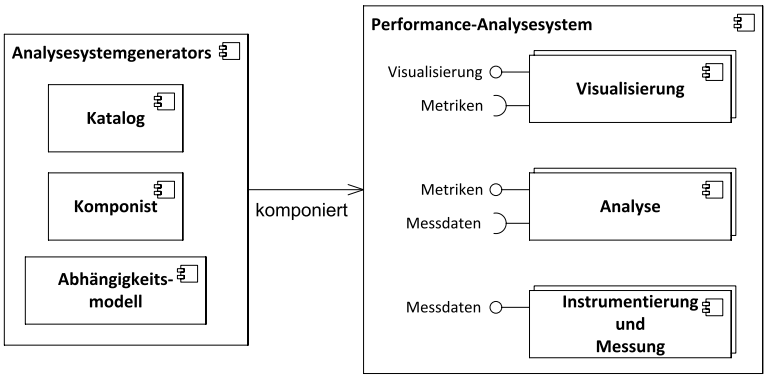

Abbildung 7.2: Der Analysesystemgenerator soll ein Performance-Analyse-System aus den Komponenten komponieren, die der Systemarchitekt mithilfe des Katalogs wählt.

Der Analysesystemgenerator könnte noch dahin gehend erweitert werden, dass er selbst die Komponenten des Performance-Analyse-Systems ausführt. Insbesondere kann der Analysesystemgenerator das zu untersuchende System mit einer Instrumentierung versehen und die Analyse der Messdaten durchführen. Idealerweise sollte sowohl eine Offline- als auch eine Online-Analyse der Daten möglich sein. Im Falle einer Online-Analyse wäre es wünschenswert, wenn sich der Systemarchitekt jederzeit für einen anderen Satz an Analysen mit entsprechend anderen Messdaten entscheiden könnte. So lassen sich, wie von Miller et al. [50] beschrieben, verschiedene Sachverhalte in einem Testlauf untersuchen.

Der Katalog des Analysesystemgenerators könnte so erweitert werden, dass das zu untersuchende komponentenbasierte System automatisch Einfluss auf die möglichen Messungen hat. Die Komponenten und die Umgebung des zu untersuchenden Systems würden also die Auswahl der Messfühler automatisch einschränken. Idealerweise müsste mit jeder Komponente, jeder

Middleware, jedem Betriebssystem und ggf. auch mit jeder Hardware eine Spezifikation mitgeliefert werden, in der dargestellt wird, welche Messpunkte hier verfügbar sind. Nur die im zu untersuchenden System und seiner Umgebung verfügbaren Messpunkte können dann instrumentiert werden. Dementsprechend müssen die zur Verfügung stehenden Messfühler ebenfalls angeben, an welchen Messpunkten sie eingesetzt werden können. So ließe sich die Instrumentierung zielsicher für das zu untersuchende System generieren. Allerdings kann so nicht nur die Auswahl an Messfühlern sinnvoll eingeschränkt werden, sondern mittels der expliziten Abhängigkeiten auch die Auswahl an Analysen und Visualisierungen. Zum Beispiel würde der Analysesystemgenerator Analysen bezüglich des automatischen Speichermanagements durch einen „Garbage-Collector“ bei einem zu untersuchenden System mit manuellem Speichermanagement nicht anbieten.

Zusammenfassung

Zusammenfassend lässt sich sagen, dass PBlaman noch vielfältig erweitert werden kann. Insbesondere die angesprochenen Verbesserungen im Rahmen der Messung und Simulation sowie der Visualisierung können PBlaman Stück für Stück von einer prototypischen Implementierung zu einem Produkt weiterentwickeln. Dazu ist es natürlich auch nötig weitere Fallstudien oder sogar industrielle Beispiele zu bearbeiten, um die Auswirkung der vorgenommenen Verbesserungen bewerten zu können.

Im Rahmen der über PBlaman hinausgehenden Forschungsarbeit sollen Zusatzdaten in die Analyse mit einbezogen werden. Auf PBlaman übertragen könnte dies zum Beispiel für eine über den Stacktrace hinausgehende Differenzierung der Messdaten genutzt werden. Beispielsweise könnte die Benutzer-Session ein zusätzliches Merkmal zur Gruppierung von Anfragepfaden bilden. Ein zusätzlicher Aspekt der weiteren Forschungsarbeit ist die konsequente komponentenbasierte Aufbereitung eines Performance-Analyse-Systems. Dies soll sogar zur dynamischen Komposition dieser Systeme genutzt werden. PBlaman müsste dementsprechend in Komponenten der Kategorien Instrumentierung und Messung, Analyse sowie Visualisierung zerlegt werden. Für PBlaman stehen zwei verschiedene Instrumentierungskomponenten zur Verfügung, eine für jede Fallstudie. Dazu kommt die PBlaman-Analyse, die aus mehreren atomaren Analysekomponenten besteht. Es gibt eine Analysekomponente für die jeweilige Datenvorbereitung und eine für jeden Lageparameter und eine für den KS-Test. Zudem können auch mehrere Visualisierungskomponenten unterschieden werden: eine für den Blame-Graphen,

eine für den Box-Whisker-Plot, sowie eine für die verschiedenen numerischen Ausgaben. Mittels dieser komponentenbasierten Aufbereitung kann der Systemarchitekt die Analysen wählen, die er benötigt, um die aktuellen Fragen bezüglich des zu untersuchenden Systems zu beantworten. Dabei könnte er sogar nur Teile von PBlaman wiederverwenden, wenngleich sich die Zusammenstellung von PBlaman, wie sie in dieser Arbeit beschrieben ist, in den Fallstudien bewährt hat.

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

Ablaufverfolgungsprotokoll
: Das Ablaufverfolgungsprotokoll (engl. „trace“) ist eine Sammlung von Beobachtungen (bzw. Messwerten), die verschiedene Sachverhalte, wie z. B. Operationsanfragen beschreiben können. Ein Ablaufverfolgungsprotokoll entsteht durch Messungen, die beispielsweise während eines Testlaufs stattfinden. Falls mehrere Ablaufverfolgungsprotokolle, beispielsweise auf verschiedenen Hardwareknoten, anfallen, ist es nötig die verschiedenen Ablaufverfolgungsprotokolle zusammenzuführen.

allocation model
: s. Verteilungsmodell

Anfragegraph
: Basiert auf dem Aufrufkontextbaum und eliminiert dessen Redundanz durch Zusammenlegen identischer Teilbäume.

Anfragepfad
: Ein Anfragepfad ist eine Liste von Antwortzeitbeobachtungen, die zu den Operationsaufrufen gehören, die durch eine Anfrage von außerhalb des Systems an eine Systemoperation bewirkt wurden.

angebotene Schnittstelle
: Die angebotene Schnittstelle (engl. „provided interface“) einer Komponente gibt an, welche Operationen diese Komponente anderen Komponenten anbietet. In diesem Rahmen wird die Funktionalität (und ggf. auch nicht-funktionale Eigenschaften) der Operation vertraglich festgelegt.

Antwortzeit
: Die Antwortzeit ist die verstrichene Zeit zwischen einem Start- und Endeereignis. In dieser Arbeit wird Antwortzeit von Komponentenoperationen als Zeit zwischen dem Aufruf der Operation und der Rückgabe des Kontrollflusses durch die Operation auf dem Hardwareknoten, der die Komponentenoperation ausführt, definiert.

assembly
: s. Zusammenstellen

assembly model
: s. Systemmodell

Aufrufbaum
: Ein Aufrufbaum baut auf dem Anfragepfad auf. Er macht die Aufrufbeziehungen zwischen den Operationen im Anfragepfad explizit. Wenn eine Operation A die Operation B einmal aufruft, ist B ein Kind von A. Bei mehrfachen Aufrufen hat die Operation A mehrere Operation-B-Kinder.

Aufrufkontextbaum
: Fasst im Unterschied zum Aufrufbaum, der jeden Operationsaufruf gesondert darstellt, Aufrufe derselben Operation von einem bestimmten Knoten aus zusammen. So können beispielsweise Aufrufe aus Schleifen platzsparend dargestellt werden.

benötigte Schnittstelle
: Die benötigte Schnittstelle (engl. „provided interface“) einer Komponente gibt an, welche Operationen von anderen Komponenten benötigt werden. Dabei wird vertragliche festgelegt, welche Funktionalität diese Operation bieten muss.

Bereitstellung
: Die Bereitstellung (engl. „provisioning“) ist ein Arbeitsablauf im Prozess der Entwicklung eines komponentenbasierten Systems. Im Rahmen der Bereitstellung (engl. „provisioning“) beschafft der Systemarchitekt die geplanten Komponenten. Dabei können die Komponenten von Dritten beschafft oder innerhalb der Organisation zur Verfügung gestellt werden. Wenn nötig können Komponenten auch extra für das System erstellt werden. Dabei werden Komponenten in einem Prozess erstellt, der entkoppelt von der Systemerstellung abläuft.

component repository
: s. Komponenten-Repository

CSV-Datei
: CSV steht für „comma separated values“. Eine CSV-Datei ist eine Textdatei, bei der jede Zeile durch ein festgelegtes Trennzeichen, das Komma, in Spalten aufgeteilt wird. Teils werden auch andere Trennzeichen verwendet.

deployment
: s. Verteilung und Auslieferung

FLOP
: Gleitkommaoperation; aus dem englischen „**fl**oating point **op**eration“

FLOPS
: Gleitkommaoperationen pro Sekunde; aus dem englischen „**fl**oating point **op**erations per **s**econd“

Fußtrapez
: Ein Fußtrapez ist ein Trapez, das seine Basis (längere parallele Seite) unten hat.

Gleisplanvisualisierung
: Die Gleisplanvisualisierung (engl. „train-schedule visualization“) stellt die Zeit meist auf der X-Achse dar und verschiedene Systemelemente auf der Y-Achse. Dann werden Zeitspannen für die verschiedenen Systemelemente auf zur X-Achse parallelen Gleisen bzw. Lebenslinien dargestellt. Diese Darstellung ist ähnlich dem UML-Sequenzdiagramm. Zudem kann die Gleisplanvisualisierung auch als Spielart des Gantt-Diagramms [37] gesehen werden.

installierte Komponente
: Eine installierte Komponente findet sich, wenn eine Komponentenimplementierung auf einem Hardwareknoten installiert wurde. Die installierte Komponente kann ein- oder mehrfach auf dem Hardwareknoten ausgeführt werden.

Komponenten-Repository
: Das Komponenten-Repository (engl. „component repository“) ist ein Diagramm aus dem PCM. Dort werden die Komponenten mit ihren Schnittstellen spezifiziert. Innerhalb der Schnittstellen werden dabei die Komponentenoperationen mit ihren Signaturen definiert. Das Komponenten-Repository wird vom Komponentenentwickler zur Verfügung gestellt.

Komponentenimplementierung
: Eine Komponentenimplementierung realisiert eine Komponentenspezifikation. Verschiedene Komponentenimplementierungen stehen auf dem Komponentenmarkt zueinander in Konkurrenz. Eine Komponentenimplementierung kann auf einem oder mehreren Hardwareknoten installiert werden.

Komponentenobjekt
: Ein Komponentenobjekt ist eine ausgeführte Instanz einer Komponente, die durch die Ausführung einer installierten Komponente auf einem bestimmten Hardwareknoten entsteht.

Komponentenspezifikation
: Die Komponentenspezifikation umfasst die angebotenen und benötigten Schnittstellen der Komponente. Damit werden die gewünschten funktionalen und nichtfunktionalen Eigenschaften einer Komponente

angegeben. Eine Komponentenspezifikation kann von mehreren Komponentenimplementierungen realisiert werden.

Kopftrapez
Ein Kopftrapez ist ein Trapez, das seine Basis (längere parallele Seite) oben hat.

Lageparameter
Lageparameter sollen die Lage der Elemente der Stichprobe (bzw. der Grundgesamtheit) in Relation zur Messskala beschreiben. Bekannte Lageparameter sind das arithmetische Mittel, das Minimum, das Maximum und die Quartile, also 25%-Quartil, Median und 75%-Quartil.

Nettoantwortzeit
Die Antwortzeit einer Methode bzw. Operation, bei der die Antwortzeit aufgerufener Methoden bzw. Operationen herausgerechnet wurde.

Offline-Analyse
Eine Offline-Analyse von Messdaten findet statt, nachdem ein Lauf des Systems abgeschlossen ist. Dieses Verfahren wird häufig während der Entwicklung im Rahmen von Tests verwendet. Es ist üblicherweise einfacher zu handhaben, als die Online-Analyse, da hier während des Betriebs lediglich Daten gemessen werden und die Analyse später auf demselben System erfolgen kann. Beim Test kann ein höherer Grad an Overhead zur Messung der Daten in Kauf genommen werden als im Betrieb.

Online-Analyse
Eine Online-Analyse von Messdaten findet statt, während das System läuft. Dieses Verfahren wird häufig beim Produktivbetrieb des Systems verwendet. Dann wird es auch als Monitoring bezeichnet [43]. Insbesondere im Produktivbetrieb muss die Erhebung von Messdaten häufig auf ein Maß beschränkt werden, dass das System nicht nennenswert belastet. Zudem muss eine Infrastruktur geschaffen werden, die die Analyse parallel zum Betrieb erlaubt.

p-Wert
Der p-Wert (engl. p-value) ist ein Wahrscheinlichkeitswert, der angibt wie wahrscheinlich es ist, die vorliegenden Daten zu erhalten, wenn man annimmt, dass die Nullhypothese gilt. Der p-Wert wird auch als Signifikanzwert oder auch als Überschreitungswahrscheinlichkeit bezeichnet.

Palladio Component Model
Das „Palladio Component Model“ (deutsch: Palladio-Komponentenmodell) ermöglicht es Modelle von komponentenbasierten Systemen zu

erstellen. Mithilfe dieser Modelle können verschiedene nicht-funktionale Eigenschaften, insbesondere die Performance, eines solchen Systems in verschiedenen Szenarien untersucht werden. Das PCM stellt verschiedene Diagramme für verschiedene am Entwicklungsprozess beteiligte Rollen bereit. Die Diagrammarten sind das Komponenten-Repository (engl. „component repository“), die Service-Effect-Specification (SEFF), das Systemmodell (engl. „assembly model“), das Ressourcenmodell (engl. „ressource environment“), das Verteilungsmodell (engl. „allocation model“) und das Verwendungsmodell (engl. „usage model“).

PCM
s. Palladio Component Model

Performance-Kontrakt
Ein Performance-Kontrakt spezifiziert die Performance einer Komponente. Mit dem Kontrakt versprechen die Komponentenentwickler, dass die genannte Performance auch eingehalten wird. Eine solche Performance-Zusage wird meist für jede Komponentenoperationen einzeln gegeben. Außerdem kann der Kontrakt auf bestimmte Umstände eingeschränkt sein. In einem Kontrakt können die folgenden Punkt spezifiziert sein: 1) Performance-Schwellwerte; 2) Abhängigkeiten zur Umgebung; 3) Verbindung zwischen Performance und Umgebung; 4) Einschränkungen.

Performance-Test
In einem Performance-Test wird die Performance (oder Effizienz) eines Systems getestet. Dabei werden Testbedingungen bezüglich der Performance-Metriken Antwortzeit, Durchsatz und Ressourcenverbrauch überprüft. Aufgrund von verschiedenen Effekten in der Umgebung des Systems (z. B. Caching oder Scheduling), wird bei Performance-Tests immer eine Datenreihe dieser Metriken ermittelt, um Ausreißer kompensieren zu können. Aufgrund der meist hohen Anzahl von (parallelen) Wiederholungen ist Testautomatisierung unabdingbar.

Pfad
s. Anfragepfad

provided interface
s. angebotene Schnittstelle

provisioning
s. Bereitstellung

Punktschätzer
Ein Punktschätzer ist eine Funktion, die aus einer Stichprobe einen statistischen Parameter als Abschätzung dieses Parameters für die Grundgesamtheit berechnet. In dieser Arbeit werden Lageparameter für die

Datenreihe aus einem Test bzw. einer Simulation berechnet, um auf das allgemeine Verhalten in allen Läufen dieses Tests bzw. dieser Simulation zu schließen.

QoS
Die englische Abkürzung steht für „quality of service“ oder die Dienstgüte. Mit der Dienstgüte wird bewertet, in welchem Maße ein Dienst nichtfunktionale Eigenschaften wie Performance und Zuverlässigkeit erfüllt. Die konkreten Anforderungen hängen vom Einsatzbereich und den jeweiligen Benutzern des Dienstes ab und werden in der Regel individuell festgelegt. Der Begriff stammt aus dem Telefoniebereich [36] und ist mittlerweile auch in der geschilderten allgemeinen Bedeutung gängig. Dabei tritt er insbesondere bei Webservices häufig auf. Dort ist die QoS häufig Bestandteil einer vertraglichen Vereinbarung mit einem Dienstanbieter. Jedoch wird der Begriff auch beispielsweise im Kontext von komponentenbasierten Systemen verwendet.

QoS-Analyse
Die QoS-Analyse ist ein Arbeitsablauf im Prozess der Entwicklung eines komponentenbasierten Systems. Im Rahmen der QoS-Analyse wird die geplante Architektur des komponentenbasierten Systems analysiert und verfeinert. Dazu werden die bisher erstellten Modelle oder speziell erstellte Analysemodelle, wie zum Beispiel eine Palladio-Component-Model-Instanz (PCM-Instanz), untersucht. Dabei werden nicht-funktionale Eigenschaften analysiert.

required interface
s. benötigte Schnittstelle

resource environment
s. Ressourcenmodell

Ressourcenmodell
Das Ressourcenmodell ist ein Diagramm aus dem PCM. Das Ressourcenmodell spezifiziert die dem System zur Verfügung stehende Hardware, die jeweiligen Hardwareressourcen sowie deren Kapazität.

SEFF
s. Service-Effect-Specification

Service-Effect-Specification
Die Service-Effect-Specification (SEFF) ist ein Diagramm aus dem PCM. Die Komponentenentwickler geben mittels eines Kontrollflussmodells

eine Performance-Zusicherung für ihre Komponenten ab. Das Kontrollflussmodell spezifiziert die Performance-relevanten Aspekte der jeweiligen Komponentenoperation. Insbesondere werden Ressourcenverbräuche für interne Berechnungen und Aufrufe von anderen Operationen innerhalb des Kontrollflusses modelliert.

Systemarchitekt
Der Systemarchitekt ist der Softwarearchitekt für das komponentenbasierte Gesamtsystem. Wird auch als Systemsoftwarearchitekt bezeichnet.

Systemmodell
Das Systemmodell (engl. „assembly model“) ist ein Diagramm aus dem PCM. Das Systemmodell spezifiziert wie die Komponenten aus einem oder mehreren Komponenten-Repositories miteinander gekoppelt werden, so dass das gewünschte Gesamtsystem entsteht. Im Systemmodell werden Komponentenobjekte eingeplant.

Systemoperation
Eine Systemoperation ist eine Komponentenoperation (erreichbar über die angebotene Schnittstelle der Komponente), die für externe Benutzer (außerhalb des Systems) zur Verfügung steht.

trace
s. Ablaufverfolgungsprotokoll

train-schedule visualization
s. Gleisplanvisualisierung

usage model
s. Verwendungsmodell

Verteilung und Auslieferung
Die Verteilung und Auslieferung (engl. „deployment“) ist ein Arbeitsablauf im Prozess der Entwicklung eines komponentenbasierten Systems. Der Arbeitsablauf der Verteilung und Auslieferung (engl. „deployment“) beschreibt wie aus einem System ein Produkt erstellt wird. Es werden also Verkaufs- und Schulungsunterlagen erstellt, sowie Akzeptanztests durchgeführt. Zudem installiert der System-Deployer das System in der Produktivumgebung und nimmt es in Betrieb. Dabei werden die Komponenten auf die verfügbaren Hardwareknoten verteilt.

Verteilungsmodell
Das Verteilungsmodell ist ein Diagramm aus dem PCM. Das Verteilungsmodell spezifiziert, wie die Komponentenobjekte aus dem Systemmodell auf die Hardware aus dem Ressourcenmodell verteilt werden.

Verwendungsmodell
: Das Verwendungsmodell (engl. „usage model“) ist ein Diagramm aus dem PCM. Das Verwendungsmodell spezifiziert wie ein System von Benutzern gebraucht wird. Dabei wird ein Kontrollfluss modelliert, der verschiedene Anfragen an das System und Bedenkzeiten enthält. Zudem wird der Grad an Parallelität und die Wiederholrate der Benutzerlast angegeben.

Zusammenstellen
: Das Zusammenstellen (engl. „assembly“) ist ein Arbeitsablauf im Prozess der Entwicklung eines komponentenbasierten Systems. Im Rahmen des Zusammenstellens integriert der Systemarchitekt existierende Komponenten und Systeme in das Gesamtsystem, wie es in der Komposition des Systems beschrieben ist.

Literaturverzeichnis

[1] Barham, Paul ; Donnelly, Austin ; Isaacs, Rebecca ; Mortier, Richard: Using Magpie for request extraction and workload modelling. In: *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. Berkeley, CA, USA : USENIX Association, 2004, 259-272

[2] Basili, Victor ; Caldiera, Gianluigi ; Rombach, H. D.: Goal Question Metric Paradigm. In: Marciniak, J. J. (Hrsg.): *Encyclopedia of Software Engineering* Bd. 1. New York, NY, USA : John Wiley & Sons, Inc., 1994, S. 528–532

[3] Beck, Kent: *JUnit pocket guide*. Sebastopol, CA, USA : O'Reilly Media, 2004. – ISBN 978–0596007430

[4] Becker, Steffen: *Coupled model transformations for QoS enabled component-based software design*. Uhlhornsweg 49-55, 26129 Oldenburg, Universität Oldenburg, Diss., 2008

[5] Becker, Steffen ; Dencker, Tobias ; Happe, Jens: Model-Driven Generation of Performance Prototypes. Version: 2008. http://dx.doi.org/10.1007/978-_3-_540-_69814-_2_7. In: Kounev, S. (Hrsg.) ; Gorton, I. (Hrsg.) ; Sachs, K. (Hrsg.): *Performance Evaluation: Metrics, Models and Benchmarks* Bd. 5119. Berlin, Deutschland : Springer Berlin Heidelberg, 2008. – DOI 10.1007/978–3–540–69814–2_7, S. 79–98

[6] Becker, Steffen ; Happe, Jens ; Koziolek, Heiko: Putting Components into Context: Supporting QoS-Predictions with an explicit Context Model. In: Reussner, R. (Hrsg.) ; Szyperski, C. (Hrsg.) ; Weck, W. (Hrsg.): *Proceedings of the 11th International Workshop on Component Oriented Programming (WCOP'06)*. Karlsruhe, Deutschland : Universität Karlsruhe, 2006, 1-6

[7] Becker, Steffen ; Hauck, Michael ; Trifu, Mircea ; Krogmann, Klaus ; Kofron, Jan: Reverse Engineering Component Models for Quality Predictions. In: Capilla, R. (Hrsg.) ; Ferenc, R. (Hrsg.) ; Dueñas, J. C.

(Hrsg.): *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR)*. Los Alamitos, CA, USA : IEEE Computer Society, 2010, 199-202

[8] Becker, Steffen ; Koziolek, Heiko ; Reussner, Ralf: The Palladio component model for model-driven performance prediction. In: *Journal of Systems and Software* 82 (2009), Nr. 1, S. 3–22. http://dx.doi.org/10.1016/j.jss.2008.03.066. – DOI 10.1016/j.jss.2008.03.066

[9] Beydeda, Sami: Research in testing COTS components - built-in testing approaches. In: *The 3rd ACS/IEEE International Conference on Computer Systems and Applications*. Los Alamitos, CA, USA : IEEE Computer Society, 2005, S. 101–104

[10] Boehm, B. W.: Guidelines for Verifying and Validating Software Requirements and Design. In: Samet, P. A. (Hrsg.): *Proceedings of Euro IFIP*. Amsterdam, Netherlands : North-Holland Publishing Company, 1979, S. 711–719

[11] Brüseke, Frank ; Engels, Gregor ; Becker, Steffen: Palladio-based performance blame analysis. In: *Proceedings of the 16th international workshop on Component-oriented programming (WCOP '11)*. New York, NY, USA : Association for Computing Machinery, 2011. – ISBN 978–1–4503–0726–0, S. 25–32

[12] Brüseke, Frank ; Engels, Gregor ; Becker, Steffen: Decision support via automated metric comparison for the palladio-based performance blame analysis. In: Seelam, S. (Hrsg.): *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE '13)*. New York, NY, USA : Association for Computing Machinery, 2013. – ISBN 978–1–4503–1636–1, S. 77–88

[13] Brüseke, Frank ; Wachsmuth, Henning ; Engels, Gregor ; Becker, Steffen: PBlaman: Performance Blame Analysis based on Palladio Contracts. In: *Concurrency and Computation Practice and Experience* 26 (2014), Nr. 12, S. 1975–2004. http://dx.doi.org/10.1002/cpe.3226. – DOI 10.1002/cpe.3226. – ISSN 1532–0634

[14] Buschmann, Frank ; Meunier, Regine ; Rohnert, Hans ; Sommerlad, Peter ; Stal, Michael: *Pattern-Oriented Software Architecture: A System of Patterns*. New York, NY, USA : John Wiley & Sons, Inc., 1996. – ISBN 0–471–95869–7

[15] Cheesman, John ; Daniels, John: *UML Components: A Simple Process for Specifying Component-Based Software.* Boston, MA, USA : Addison-Wesley, 2000. – ISBN 0–201–70851–5

[16] Crnkovic, I. ; Chaudron, M. ; Larsson, S.: Component-Based Development Process and Component Lifecycle. In: Dini, P. (Hrsg.): *Proceedings of the International Conference on Software Engineering Advances (ICSEA '06).* Los Alamitos, CA, USA : IEEE Computer Society, oct. 2006, S. 44–53

[17] DeRose, Luiz ; Pantano, Mario ; Reed, Daniel ; Vetter, Jeffrey: Performance Issues in Parallel Processing Systems. Version: 2000. http://dx.doi.org/10.1007/3-_540-_46506-_5_6. In: Haring, G. (Hrsg.) ; Lindemann, C. (Hrsg.) ; Reiser, M. (Hrsg.): *Performance Evaluation: Origins and Directions* Bd. 1769. Berlin, Deutschland : Springer Berlin Heidelberg, 2000. – DOI 10.1007/3–540–46506–5_6. – ISBN 978–3–540–67193–0, S. 141–159

[18] Devore, Jay L. ; Berk, Kenneth N. ; Casella, G. (Hrsg.) ; Fienberg, S. (Hrsg.) ; Olkin, I. (Hrsg.): *Modern Mathematical Statistics with Applications.* Second Edition. New York, NY, USA : Springer New York, 2012 (Springer Texts in Statistics). http://dx.doi.org/10.1007/978-_1-_4614-_0391-_3

[19] Dmitriev, Mikhail: Selective profiling of Java applications using dynamic bytecode instrumentation. In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* Los Alamitos, CA, USA : IEEE Computer Society, 2004, S. 141–150

[20] ej-technologies GmbH: *Java Profiler - JProfiler.* http://www.ej-_technologies.com/products/jprofiler/overview.html. – (zuletzt besucht am 20.09.2014)

[21] Engels, Gregor ; Güldali, Baris ; Lohmann, Marc: Towards Model-Driven Unit Testing. In: Hearnden, D. (Hrsg.) ; Süss, J.G. (Hrsg.) ; Rapin, N. (Hrsg.) ; Baudry, B. (Hrsg.): *Proceedings of the workshop on Model Design and Validation (MoDeVa 2006), Toulouse (France).* Berlin / Heidelberg : Le Commissariat à l'Energie Atomique - CEA, Oktober 2006, S. 16–29

[22] Faruqui, Manaal ; Padó, Sebastian: Training and Evaluating a German Named Entity Recognizer with Semantic Generalization. In: Pinkal, M. (Hrsg.) ; Rehbein, I. (Hrsg.) ; Schulte im Walde, S. (Hrsg.) ; Storrer, A. (Hrsg.): *Proceedings of the Conference on Natural Language Processing*

(KONVENS 2010). Saarbrücken, Deutschland : universaar, 2010, S. 129–133

[23] Ferrucci, David ; Lally, Adam: UIMA: an architectural approach to unstructured information processing in the corporate research environment. In: *Natural Language Engineering* 10 (2004), September, Nr. 3-4, S. 327–348. http://dx.doi.org/10.1017/S1351324904003523. – DOI 10.1017/S1351324904003523

[24] Finkel, Jenny R. ; Grenager, Trond ; Manning, Christopher D.: Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. In: *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL '05)*. Stroudsburg, PA, USA : Association for Computational Linguistics, 2005, S. 363–370

[25] Fischer, Christian: *Automatische Generierung von Testskripten aus Palladio-Modellen*, University of Paderborn, Bachelorarbeit, 2013

[26] Gao, Jerry ; Tsao, H.-S. J. ; Wu, Ye: *Testing and quality assurance for component-based software*. Norwood, MA, USA : Artech House, 2003. – ISBN 1–58053–480–5

[27] Güldali, Baris: *Integrated Contract-based Testing into the Model-driven Software Development*, Universität Paderborn, Diss., 2014

[28] Gregg, Brendan: *Flame Graphs*. http://dtrace.org/blogs/brendan/2011/12/16/flame-_graphs/. – (zuletzt besucht am 20.09.2014)

[29] Gregg, Brendan: *Blazing Performance with Flame Graphs*. Präsentation auf der 27th Large Installation System Administration Conference (LISA '13), Washington, D.C., USA. https://www.usenix.org/conference/lisa13/technical-_sessions/plenary/gregg. Version: November 2013

[30] Groenda, Henning: An Accuracy Information Annotation Model for Validated Service Behavior Specifications. Version: 2011. http://dx.doi.org/10.1007/978-_3-_642-_21210-_9_36. In: Dingel, J. (Hrsg.) ; Solberg, A. (Hrsg.): *Models in Software Engineering* Bd. 6627. Berlin, Deutschland : Springer Berlin Heidelberg, 2011. – DOI 10.1007/978–3–642–21210–9_36. – ISBN 978–3–642–21209–3, S. 369–383

[31] Groenda, Henning: *Certifying Software Component Performance Specifications*. Karlsruhe, Germany, Karlsruher Institut für Technologie (KIT), Diss., September 2013. http://dx.doi.org/10.5445/KSP/1000036063. – DOI 10.5445/KSP/1000036063

[32] Happe, Jens ; Becker, Steffen ; Rathfelder, Christoph ; Friedrich, Holger ; Reussner, Ralf: Parametric performance completions for model-driven performance prediction. In: *Performance Evaluation* 67 (2010), Nr. 8, S. 694–716. http://dx.doi.org/10.1016/j.peva.2009.07.006. – DOI 10.1016/j.peva.2009.07.006. – ISSN 0166–5316

[33] Herold, Sebastian ; Klus, Holger ; Welsch, Yannick ; Deiters, Constanze ; Rausch, Andreas ; Reussner, Ralf ; Krogmann, Klaus ; Koziolek, Heiko ; Mirandola, Raffaela ; Hummel, Benjamin ; Meisinger, Michael ; Pfaller, Christian: CoCoME - The Common Component Modeling Example. Version: 2008. http://dx.doi.org/10.1007/978-_3-_540-_85289-_6_3. In: Rausch, A. (Hrsg.) ; Reussner, R. (Hrsg.) ; Mirandola, R. (Hrsg.) ; Plášil, F. (Hrsg.): *The Common Component Modeling Example* Bd. 5153. Berlin, Deutschland : Springer Berlin Heidelberg, 2008. – DOI 10.1007/978–3–540–85289–6_3, S. 16–53

[34] IEEE Computer Society: IEEE Standard for Software and System Test Documentation. In: *IEEE Std 829-2008* (2008). http://dx.doi.org/10.1109/IEEESTD.2008.4578383. – DOI 10.1109/IEEESTD.2008.4578383

[35] International Organization for Standardization: *ISO/IEC TR 9126-3: Software engineering — product quality — Part 3 : internal metrics*. Geneva, Switzerland : ISO, 2003

[36] International Telecommunication Union (ITU): *Terms and definitions related to quality of service and network performance including dependability*. Version: August 1994. http://www.itu.int/rec/T-_REC-_E.800/en

[37] Jain, Raj: *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling: Techniques for Experimental Design, Measurement, Simulation and Modeling*. New York, NY, USA : John Wiley & Sons, 1991. – ISBN 0–471–50336–3

[38] Jiang, Zhen M. ; Hassan, A.E. ; Hamann, G. ; Flora, P.: Automated performance analysis of load tests. In: Kontogiannis, K. (Hrsg.) ; Xie, T. (Hrsg.): *IEEE International Conference on Software Maintenance (ICSM*

2009). Los Alamitos, CA, USA : IEEE Computer Society, 2009. – ISSN 1063–6773, S. 125–134

[39] Kłaczewski, Paweł ; Wytrębowicz, Jacek: j2eeprof – a tool for testing multitier applications. In: Sacha, K. (Hrsg.): *Software Engineering Techniques: Design for Quality* Bd. 227. New York, NY, USA : Springer US, 2007 (IFIP International Federation for Information Processing). – ISBN 978–0–387–39387–2, S. 199–210

[40] Klaussner, Christian: *Extensible Performance Prototype Transformations for Multiple Platforms*, Universität Paderborn, Bachelorarbeit, 2014

[41] Koziolek, Heiko ; Happe, Jens: A QoS Driven Development Process Model for Component-Based Software Systems. Version: 2006. http://dx.doi.org/10.1007/11783565_25. In: Gorton, I. (Hrsg.) ; Heineman, G. (Hrsg.) ; Crnkovic, I. (Hrsg.) ; Schmidt, H. (Hrsg.) ; Stafford, J. (Hrsg.) ; Szyperski, C. (Hrsg.) ; Wallnau, K. (Hrsg.): *Component-Based Software Engineering* Bd. 4063. Berlin, Deutschland : Springer Berlin Heidelberg, 2006. – DOI 10.1007/11783565_25, S. 336–343

[42] Kramer, Max E. ; Durdik, Zoya ; Hauck, Michael ; Henss, Jörg ; Küster, Martin ; Merkle, Philipp ; Rentschler, Andreas: Extending the Palladio Component Model using Profiles and Stereotypes. In: Becker, S. (Hrsg.) ; Happe, J. (Hrsg.) ; Koziolek, A. (Hrsg.) ; Reussner, R. (Hrsg.): *Palladio Days 2012 Proceedings*. Karlsruhe : KIT, Faculty of Informatics, 2012 (Karlsruhe Reports in Informatics 2012,21), 7-15. – (Proceedings als technischer Bericht erschienen)

[43] Kreft, Klaus ; Langer, Angelika: Java-Performance - Teil 3: Wie funktionieren Profiling-Tools? In: *JavaSPEKTRUM* (2005), November, Nr. 6. http://www.angelikalanger.com/Articles/EffectiveJava/23.ProfilingTools/23.ProfilingTools.html

[44] Krogmann, Klaus ; Reussner, Ralf: Palladio – Prediction of Performance Properties. Version: 2008. http://dx.doi.org/10.1007/978-_3-_540-_85289-_6_12. In: Rausch, A. (Hrsg.) ; Reussner, R. (Hrsg.) ; Mirandola, R. (Hrsg.) ; Plášil, F. (Hrsg.): *The Common Component Modeling Example* Bd. 5153. Berlin, Deutschland : Springer Berlin Heidelberg, 2008. – DOI 10.1007/978–3–540–85289–6_12, S. 297–326

[45] Krogmann, Klaus ; Schweda, Christian ; Buckl, Sabine ; Kuperberg, Michael ; Martens, Anne ; Matthes, Florian: Improved Feedback for Architectural Performance Prediction Using Software Cartography

Visualizations. In: Mirandola, R. (Hrsg.) ; Gorton, I. (Hrsg.) ; Hofmeister, C. (Hrsg.): *Architectures for Adaptive Software Systems* Bd. 5581. Berlin, Deutschland : Springer Berlin Heidelberg, 2009 (Lecture Notes in Computer Science). – ISBN 978–3–642–02350–7, S. 52–69

[46] Kruchten, Philippe: *The rational unified process: an introduction.* Third Edition. Boston, MA, USA : Addison-Wesley, 2004. – ISBN 0–321–19770–4

[47] Lehrig, Sebastian ; Zolynski, Thomas: Performance Prototyping with ProtoCom in a Virtualised Environment: A Case Study. In: S. Becker, R. R. J. Happe H. J. Happe (Hrsg.) ; FZI, Karlsruhe, Germany (Veranst.): *Proceedings to Palladio Days 2011* FZI, Karlsruhe, Germany, 2011, 7-14

[48] Marwede, Nina ; Rohr, Matthias ; van Hoorn, André ; Hasselbring, Wilhelm: Automatic Failure Diagnosis Support in Distributed Large-Scale Software Systems Based on Timing Behavior Anomaly Correlation. In: Ferenc, R. (Hrsg.) ; Knodel, J. (Hrsg.) ; Winter, A. (Hrsg.): *13th European Conference on Software Maintenance and Reengineering (CSMR '09).* Los Alamitos, CA, USA : IEEE Computer Society, März 2009. – ISSN 1534–5351, S. 47–58

[49] Meyer, Bertrand: Applying 'design by contract'. In: *Computer* 25 (1992), Nr. 10, S. 40–51. http://dx.doi.org/10.1109/2.161279. – DOI 10.1109/2.161279

[50] Miller, Barton P. ; Callaghan, Mark D. ; Cargille, Jonathan M. ; Hollingsworth, Jeffrey K. ; Irvin, R. B. ; Karavanic, Karen L. ; Kunchithapadam, Krishna ; Newhall, Tia: The Paradyn parallel performance measurement tool. In: *Computer* 28 (1995), Nov, Nr. 11, S. 37–46. http://dx.doi.org/10.1109/2.471178. – DOI 10.1109/2.471178. – ISSN 0018–9162

[51] Molyneaux, Ian: *The Art of Application Performance Testing: Help for Programmers and Quality Assurance.* Sebastopol, CA, USA : O'Reilly Media, 2009. – ISBN 0–596–52066–2

[52] Object Management Group (OMG): *UML Profile for MARTE: Modeling and Analysis of Real-time and Embedded Systems (MARTE 1.1).* http://www.omg.org/spec/MARTE/1.1/. Version: Juni 2011

[53] Object Management Group (OMG): *Unified Modeling Language Specification (UML 2.4.1).* http://www.omg.org/spec/UML/2.4.1/. Version: August 2011

[54] Organization for the Advancement of Structured Information Standards (OASIS) ; Lally, A. (Hrsg.) ; Verspoor, K. (Hrsg.) ; Nyberg, E. (Hrsg.): *Unstructured Information Management Architecture (UIMA), Version 1.0*. 2009 http://docs.oasis-_open.org/uima/v1.0/uima-_v1.0.html

[55] R Development Core Team: *R: A Language and Environment for Statistical Computing*. Wien, Österreich: R Foundation for Statistical Computing, 2011. http://www.R-_project.org. – ISBN 3-900051-07-0

[56] Reussner, Ralf ; Becker, Steffen ; Burger, Erik ; Happe, Jens ; Hauck, Michael ; Koziolek, Anne ; Koziolek, Heiko ; Krogmann, Klaus ; Kuperberg, Michael: The Palladio Component Model / KIT, Fakultät für Informatik. Version: 2011. http://digbib.ubka.uni-_karlsruhe.de/volltexte/1000022503. Karlsruhe, 2011. (Karlsruhe reports in informatics ; 2011,14). – Forschungsbericht

[57] Reussner, Ralf ; Becker, Steffen ; Koziolek, Anne ; Koziolek, Heiko: An Empirical Investigation of the Component-Based Performance Prediction Method Palladio. Version: 2013. http://dx.doi.org/10.1007/978-_3-_642-_37395-_4_13. In: Münch, J. (Hrsg.) ; Schmid, K. (Hrsg.): *Perspectives on the Future of Software Engineering*. Berlin, Deutschland : Springer Berlin Heidelberg, 2013. – DOI 10.1007/978–3–642–37395–4_13. – ISBN 978–3–642–37394–7, S. 191–207

[58] Reussner, Ralf ; Firus, Viktoria ; Becker, Steffen: Parametric Performance Contracts for Software Components and their Compositionality. In: Weck, W. (Hrsg.) ; Bosch, J. (Hrsg.) ; Szyperski, C. (Hrsg.): *Proceedings of the 9th International Workshop on Component-Oriented Programming (WCOP 04)*, 2004

[59] Reynolds, Patrick ; Killian, Charles ; Wiener, Janet L. ; Mogul, Jeffrey C. ; Shah, Mehul A. ; Vahdat, Amin: Pip: detecting the unexpected in distributed systems. In: *Proceedings of the 3rd conference on Networked Systems Design & Implementation (NSDI'06)*. Berkeley, CA, USA : USENIX Association, 2006, 115-128. – http://usenix.org/events/nsdi06/tech/reynolds.html

[60] Rohr, Matthias ; Giesecke, Simon ; Hasselbring, Wilhelm: Timing Behavior Anomaly Detection in Enterprise Information Systems. In: Cardoso, J. (Hrsg.) ; Cordeiro, J. (Hrsg.) ; Filipe, J. (Hrsg.): *Proceedings of the Ninth International Conference on Enterprise Information Systems (ICEIS'07)*, INSTICC Press, Juni 2007. – ISBN 978–972–8865–88–7, S. 494–497

[61] Rolia, J.A. ; Sevcik, K.C.: The Method of Layers. In: *IEEE Transactions on Software Engineering* 21 (1995), August, Nr. 8, S. 689–700. http://dx.doi.org/10.1109/32.403785. – DOI 10.1109/32.403785. – ISSN 0098–5589

[62] Rutar, Nick ; Hollingsworth, Jeffrey: Assigning Blame: Mapping Performance to High Level Parallel Programming Abstractions. In: Sips, H. (Hrsg.) ; Epema, D. (Hrsg.) ; Lin, H.-X. (Hrsg.): *Euro-Par 2009 Parallel Processing* Bd. 5704. Berlin, Deutschland : Springer Berlin Heidelberg, 2009 (Lecture Notes in Computer Science). – ISBN 978–3–642–03868–6, S. 21–32

[63] Sambasivan, Raja R. ; Zheng, Alice X. ; De Rosa, Michael ; Krevat, Elie ; Whitman, Spencer ; Stroucken, Michael ; Wang, William ; Xu, Lianghong ; Ganger, Gregory R.: Diagnosing performance changes by comparing request flows. In: *Proceedings of the 8th USENIX conference on Networked systems design and implementation (NSDI'11)*. Berkeley, CA, USA : USENIX Association, 2011. – ISBN 978–931971–84–3, 43-56

[64] Sarawagi, Sunita: Information Extraction. In: *Foundations and Trends in Databases* 1 (2008), Nr. 3, S. 261–377

[65] Schmid, Helmut: Improvements in Part-of-Speech Tagging with an Application to German. In: *Proceedings of the ACL SIGDAT-Workshop*, 1995, S. 47–50

[66] Schwarze, Jochen: *Grundlagen der Statistik - Band 2: Wahrscheinlichkeitsrechnung und induktive Statistik*. 9. Auflage. Herne : Verlag Neue Wirtschafts-Briefe, 2009. – ISBN 978–3–482–56869–5

[67] Schwarze, Jochen: *Grundlagen der Statistik Band 1: Beschreibende Verfahren*. 11. Auflage. Herne : Verlag Neue Wirtschafts-Briefe, 2009. – ISBN 978–3–482–59481–6

[68] Smith, Connie U. ; Williams, Lloyd G.: *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Boston, MA, USA : Addison-Wesley, 2002. – ISBN 0–201–72229–1

[69] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest*. 3. überarbeitete und aktualisierte Auflage. dpunkt.verlag, 2005. – ISBN 3–89864–358–1

[70] Srinivas, Kavitha ; Srinivasan, Harini: Summarizing application performance from a components perspective. In: *Proceedings of the 10th European software engineering conference held jointly with 13th ACM*

SIGSOFT international symposium on Foundations of software engineering. New York, NY, USA : Association for Computing Machinery, 2005 (ESEC/FSE-13). – ISBN 1–59593–014–0, S. 136–145

[71] Szyperski, Clemens ; Gruntz, Dominik ; Murer, Stephan: *Component Software: Beyond Object-Oriented Programming*. Second Edition. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0–201–74572–0

[72] Thereska, Eno ; Salmon, Brandon ; Strunk, John ; Wachs, Matthew ; Abd-El-Malek, Michael ; Lopez, Julio ; Ganger, Gregory R.: Stardust: tracking activity in a distributed storage system. In: *SIGMETRICSPerformance Evaluation Review* 34 (2006), Juni, Nr. 1, S. 3–14. http://dx.doi.org/10.1145/1140103.1140280. – DOI 10.1145/1140103.1140280. – ISSN 0163–5999

[73] van Hoorn, André ; Rohr, Matthias ; Hasselbring, Wilhelm ; Waller, Jan ; Ehlers, Jens ; Frey, Sören ; Kieselhorst, Dennis: Continuous Monitoring of Software Services: Design and Application of the Kieker Framework / Universität Kiel. Version: November 2009. http://eprints.uni-_kiel.de/14459/. Department of Computer Science, Kiel University, Germany, November 2009. (Technische Berichte des Instituts für Informatik). – Forschungsbericht

[74] van Hoorn, André ; Waller, Jan ; Hasselbring, Wilhelm: Kieker: a framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE '12)*. New York, NY, USA : Association for Computing Machinery, 2012. – ISBN 978–1–4503–1202–8, S. 247–248

[75] Visky, Enikö V.: Log-Management: Wichtige gesetzliche Pflicht für Unternehmen. In: *TecChannel Compact* (2009), Nr. 08, S. 77–82

[76] Wachsmuth, Henning ; Prettenhofer, Peter ; Stein, Benno: Efficient Statement Identification for Automatic Market Forecasting. In: *Proceedings of the 23rd International Conference on Computational Linguistics (COLING '10)*. Stroudsburg, PA, USA : Association for Computational Linguistics, 2010, S. 1128–1136

[77] Wachsmuth, Henning ; Rose, Mirko ; Engels, Gregor: Automatic Pipeline Construction for Real-Time Annotation. In: Gelbukh, A. (Hrsg.):

Proceedings of the 14th International Conference on Intelligent Text Processing and Computational Linguistics (CICLing) Bd. 7816. Berlin, Deutschland : Springer Berlin Heidelberg, 2013 (Lecture Notes in Computer Science). – ISBN 978–3–642–37246–9, S. 38–49

[78] Wachsmuth, Henning ; Stein, Benno: Optimal Scheduling of Information Extraction Algorithms. In: *Proceedings of the 24th International Conference on Computational Linguistics (Poster Session).* Mumbai, Indien, Dezember 2012, S. 1281–1290

[79] Westermann, Dennis ; Happe, Jens ; Farahbod, Roozbeh: An Experiment Specification Language for Goal-driven, Automated Performance Evaluations. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing.* New York, NY, USA : Association for Computing Machinery, 2013 (SAC '13). – ISBN 978–1–4503–1656–9, S. 1043–1048

[80] Westermann, Dennis ; Happe, Jens ; Hauck, Michael ; Heupel, Christian: The Performance Cockpit Approach: A Framework for Systematic Performance Evaluations. In: Chaudron, M.R.V. (Hrsg.): *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010).* Los Alamitos, CA, USA : IEEE Computer Society, September 2010, S. 31–38

Anhang A

Instrumentierung

Im Folgenden werden für beide Fallstudien Auszüge aus dem Instrumentierungscode diskutiert. Dabei wird zunächst die BTrace-Instrumentierung in Abschnitt A.1 erläutert, die in der CoCoME-Fallstudie (s. Abschnitt 5.1) zum Einsatz kommt. Im Anschluss wird in Abschnitt A.2 die Kieker-Instrumentierung beschrieben, die in der Fallstudie zur Analyse unstrukturierter Texte (s. Abschnitt 5.2) genutzt wird.

A.1 BTrace-Instrumentierung

Für die Instrumentierung der CoCoME-Fallstudie (s. Abschnitt 5.1) wurde BTrace[1] verwendet. BTrace ermöglicht es Java-Programme zu instrumentieren. Dabei wird der Messquelltext immer in speziellen Programmen ausgelagert. Diese getrennte Definition der Messfühler kann als aspektorientierte Programmierung bezeichnet werden. Die BTrace-Messprogramme werden in einem eingeschränkten Java-Dialekt verfasst und müssen dementsprechend vor ihrer Verwendung zunächst mit dem BTrace-Compiler in Java-Bytecode übersetzt werden. Anschließend übernimmt das BTrace-Framework die Ausführung des Messprogramms, indem es sich über die sogenannte Java-Agent-Schnittstelle in die Ausführung des zu untersuchenden Systems einklinkt.

BTrace-Messprogramme sind wie schon erwähnt in einem eingeschränkten Java-Dialekt verfasst. Dabei wird zunächst ein Klasse mit der Annotation `@BTrace` erstellt, deren Methoden dann entsprechend ihrer Annotation für die Instrumentierung des zu untersuchenden Systems eingesetzt werden. In dieser Arbeit wird vor allem die Messung von Operationsantwortzeiten thematisiert. Auf Java übertragen bedeutet das, dass die Antwortzeit bestimmter Methoden in bestimmten Klassen gemessen werden muss. Daher wird hier

[1] s. https://kenai.com/projects/btrace (zuletzt besucht am 24.05.2014)

ausschließlich die Annotation `@OnMethod` für Messmethoden benötigt. Listing 1 zeigt zwei Beispiele für derartige Messmethoden. Die Attribute „clazz“ und „method“ der Annotation `@OnMethod` geben an, auf welche Klasse und Methode sich die Messmethode bezieht. Dabei zeigt das „+“ vor dem Klassennamen an, dass auch abgeleitete Klassen mit eingeschlossen werden sollen. Das Attribut „location“ gibt an, an welchem Punkt im Verlauf des Methodenaufrufs der Messfühler ausgeführt werden soll. In diesem Fall wird der Messfühler nach dem Ende der Methode ausgeführt. Wie aus den Eingabeparametern der Messmethoden hervorgeht, ermittelt BTrace automatisch die Antwortzeit der Methode und gibt den Wert an die Messmethode weiter. Die Messmethode selbst ruft dann die Hilfsmethode `printExeOrOut` auf, die die benötigten Werte in eine Datei schreibt.

```
1  @OnMethod(
2    clazz = "+org.cocome.tradingsystem.inventory.
       application.store.CashDeskConnectorIf"
3  , method = "bookSale"
4  , location = @Location(Kind.RETURN)
5  )
6  public static void instrumentOverall(@Duration long
       duration) {
7    printExeOrOut("EXE", "overall", duration);
8  }
9
10 @OnMethod(
11   clazz = "+org.cocome.tradingsystem.inventory.data
       .store.StoreQueryIf"
12 , method = "queryStockItemById"
13 , location = @Location(Kind.RETURN)
14 )
15 public static void instrumentQueryStockItem(
     @Duration long duration) {
16   printExeOrOut("EXE", "queryStockItem", duration);
17 }
```

Listing 1: Messcode für zwei Komponentenoperationen aus der CoCoME-Fallstudie

Die Hilfsmethode `printExeOrOut` ist in Listing 2 zu sehen. Diese Hilfsmethode ist ein gutes Beispiel dafür, dass BTrace absichtlich eingeschränkt ist, um möglichst performante Messfühler zu erzwingen. Es werden zumeist nur die Methoden aus der Klasse `BTraceUtils` verwendet, denn nur diese Methoden oder aber in derselben Klasse definierte Methoden dürfen in einem BTrace-Messfühler aufgerufen werden. In der gezeigten Hilfsmethode wird

zunächst ein String-Puffer initialisiert, in den nacheinander die verschiedenen Messwerte im CSV-Format (mit Semikolon als Trennzeichen) geschrieben werden. Dabei wird der Eintrag zunächst in dem String-Puffer gesammelt und dann geschrieben, um zu vermeiden, dass Teile verschiedener Einträge abwechselnd geschrieben werden. Zunächst wird eine Messkennung, gegeben durch den Parameter „category“, zusammen mit einem Schlüssel, aus Eingabeparametern „key“, in den Puffer geschrieben. Der Schlüssel wird später als Kurzbezeichnung für die Stacktraces verwendet, die auf dieser Operation enden. Als nächstes werden die Thread-ID, der Zeitstempel und die Antwortzeit an den Puffer angehängt. Schließlich folgt noch der aktuelle Stacktrace. Der fertige String mit allen Messdaten wird dann über die `println`-Methode in eine Datei geschrieben.

```
1  private static void printExeOrOut(String category,
       String key, long duration) {
2    java.lang.Appendable buffer = BTraceUtils.Strings
       .newStringBuilder();
3    BTraceUtils.Strings.append(buffer, category);
4    BTraceUtils.Strings.append(buffer, " > ");
5    BTraceUtils.Strings.append(buffer, key);
6    BTraceUtils.Strings.append(buffer, ";");
7
8    BTraceUtils.Strings.append(buffer, BTraceUtils.
       Strings.str(BTraceUtils.threadId(BTraceUtils.
       currentThread())));
9    BTraceUtils.Strings.append(buffer, ";");
10
11   BTraceUtils.Strings.append(buffer, BTraceUtils.
       timestamp("yyyy-MM-dd HH:mm:ss.SSS"));
12   BTraceUtils.Strings.append(buffer, ";");
13
14   BTraceUtils.Strings.append(buffer, BTraceUtils.
       Strings.str(duration));
15   BTraceUtils.Strings.append(buffer, ";");
16
17   BTraceUtils.Strings.append(buffer, BTraceUtils.
       jstackStr(4));
18   BTraceUtils.Strings.append(buffer, ";Done");
19   BTraceUtils.println(buffer);
20 }
```

Listing 2: Hilfsmethode für die BTrace-Instrumentierung, die die benötigten Messwerte zusammenträgt und in eine Datei schreibt.

Um die BTrace-Instrumentierung zu aktivieren, muss die BTrace-jar-Datei als Java-Agent in die Java-Virtual-Machine geladen werden (s. Listing 3). Dazu muss mit dem Parameter „scriptdir“ angegeben werden, wo das kompilierte BTrace-Programm zu finden ist. Zudem spezifiziert der Parameter „scriptOutputFile“ die Ausgabedatei für die BTrace-Ausgaben.

```
1 -javaagent:./btrace-agent.jar=scriptdir=./UC8,
    scriptOutputFile=./UC8/2014-08-09_11.15.59--
    Store1.txt
```

Listing 3: Aufruf der BTrace-Instrumentierung mittels `-javaagent`-Parameter der Java-Virtual-Machine

A.2 Kieker-Instrumentierung

Die Instrumentierung der zweiten Fallstudie (s. Abschnitt 5.2) setzt auf das Messframework Kieker [74, 73]. Auch bei diesem Framework stellt die aspektorientierte Programmierung die einfachste Möglichkeit dar, Messfühler in eine Java-Anwendung einzubringen. Kieker bietet für diesen Zweck schon diverse vorgefertigte Messfühler an. Kieker realisiert die aspektorientierte Programmierung für Java mithilfe des Frameworks AspectJ[2]. Es muss also in einer AspectJ-XML-Datei spezifiziert werden, welche Messfühler auf welche Klassen und Methoden angewendet werden. Die AspectJ-XML-Datei, die in der zweiten Fallstudie zum Einsatz kam, ist in Listing 5 zu sehen. Das `weaver`-Tag bildet den ersten wichtigen Block, in dem die Klassen benannt werden, in denen überhaupt Aspekte, wie Messfühler, angewendet werden sollen. Dabei muss kein vollqualifizierter Klassenname angegeben werden, sondern es können auch Platzhalter verwendet werden. Im Beispiel werden die jeweils angegebenen Pakete berücksichtigt. Der nächste wichtige Block ist das `concrete-aspect`-Tag, das den einzigen Aspekt, also Messfühler, für die gezeigte Kieker-Konfiguration beschreibt. Dieser konkrete Aspekt wird automatisch von der abstrakten Klasse `operationExecution.AbstractAspect` (vgl. Listing 5) abgeleitet und auf die im `pointcut`-Tag mit dem Namen „`monitoredOperation`“ genannten Methoden angewendet. Das `pointcut`-Tag gibt in jedem durch oder („`||`“) verbundenen Teilausdruck eine Signatur an, dessen Ausführung, also „`execution`“,

[2] s. Webseite https://eclipse.org/aspectj/ (zuletzt besucht am 09.08.2014)

instrumentiert werden soll. Die Signaturen sind dabei wiederum mit Platzhaltern für den Rückgabewert („*“) und die Eingabeparameter („..“) angegeben. Hier sind wiederum nicht alle Klassen einzeln aufgeführt, da die gemeinsamen Oberklassen der Komponentenimplementierungen instrumentiert werden, so dass eine weitere Komponente in einer UIMA-Pipeline hier vermutlich nicht explizit hinzugefügt werden müsste.

Um die Kieker-Instrumentierung mittels AspectJ zu aktivieren, muss die entsprechende Jar-Datei wiederum als Java-Agent über den entsprechenden Parameter der Java-Virtual-Machine geladen werden. Dies ist in Listing 4 zu sehen. Bei dieser Fallstudie wurde der Testfall zunächst drei Mal durchgespielt, bevor die Messung gestartet wurde. Dazu lässt sich die Instrumentierung in Kieker zunächst programmatisch abschalten und sie kann dementsprechend im weiteren Verlauf wieder eingeschaltet werden. Der in Listing 5 angegebene Messfühler wird zwar unabhängig davon in die betreffenden Klassen eingebracht, aber der Messfühler prüft vor jeder möglichen Messung, ob er Messungen nehmen darf. Somit erlaubt es die Deaktivierung der Messung zunächst die Caches aufzuwärmen, ohne dass ein nennenswerter Overhead entsteht.

```
1 -javaagent:lib/kieker-1.6_aspectj.jar
```

Listing 4: Aufruf der Kieker-Instrumentierung mittels `-javaagent`-Parameter der Java-Virtual-Machine

```
1  <!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "
     http://www.eclipse.org/aspectj/dtd/aspectj_1_5_0
     .dtd">
2  <aspectj>
3   <weaver>
4     <include within="de.upb.efxtools.ae..*"/>
5     <include within="de.upb.efxtools.application.*"/
        >
6     <include within="org.apache.uima.
        analysis_component.*"/>
7     <include within="org.apache.uima.analysis_engine
        .*"/>
8     <include within="org.apache.uima.analysis_engine
        .impl.*"/>
9   </weaver>
10
11  <aspects>
12   <concrete-aspect
13     name="de.upb.efxtools.monitoring.
         OperationExecutionAspectFull_process"
14     extends="kieker.monitoring.probe.aspectj.flow.
         operationExecution.AbstractAspect"
15   >
16     <pointcut
17       name="monitoredOperation"
18       expression="
19           (execution(* de.upb.efxtools.application.
              AnnotationCounter.count(..)))
20           || (execution(* de.upb.efxtools.ae.
              EfXAnalysisEngine.process(..)))
21           || (execution(* org.apache.uima.
              analysis_engine.impl.
              AggregateAnalysisEngine_impl.process
              (..)))
22           || (execution(* org.apache.uima.
              analysis_engine.impl.
              AnalysisEngineImplBase.process(..)))
23       "
24     />
25   </concrete-aspect>
26  </aspects>
27 </aspectj>
```

Listing 5: Spezifikation des aspektorientierten Messfühlers für Kieker basierend auf AspectJ

Anhang B

Beispiel für ein JUnit-Testskript aus Palladio-basiertem Testfall

Im Folgenden wird ein Beispiel für ein JUnit-Testskript (s. Unterabschnitt 4.1.2) diskutiert, dass aus einem Palladio-basierten Testfall erstellt wurde. Um das Testskript zu erstellen, wurde auf die PCM-Instanz aus der zweiten Fallstudie (s. Abschnitt 5.2) zurückgegriffen, bei dem ein UIMA-basiertes System zur Analyse von unstrukturierten Texten untersucht wird. Der Testfall überprüft die mittlere Antwortzeit, die die UIMA-Pipeline benötigt, um die gesuchten Informationen aus einem einzelnen Text zu extrahieren. Das Kernstück der PCM-Instanz für einen Palladio-basierten Testfall ist das Verwendungsmodell (s. Unterabschnitt 2.3.5), das beispielhaft in Abbildung B.1 dargestellt ist. Im Verwendungsdiagramm wird eine geschlossene Benutzergruppe („closed workload“) spezifiziert, bei der ein Benutzer das Szenario wiederholt ausführt, ohne zwischen den Ausführungen eine Bedenkzeit abzuwarten. Das Szenario besteht aus nur einem Schritt, in dem die UIMA-Pipeline mit einem CAS-Objekt, das (im realen System) den aktuellen Text enthält, aufgerufen wird. Im Modell wird für den Eingabeparameter lediglich die Anzahl der Tokens in einem Eingabetext mithilfe einer Wahrscheinlichkeitsfunktion angegeben. So wird die Verteilung der Tokenzahl in der im Testfall verwendeten Textsammlung spezifiziert.

Aus der vorliegenden PCM-Instanz wird dann zunächst die Testsuite erstellt (vgl. Listing 6), bei der die Testfälle in Kombination mit der Benutzerlastsimulation initialisiert und ausgeführt werden. Dabei wird hier zunächst der Testfall für das beschriebene Szenario initialisiert und in der Variablen `usageScenarioTest` abgelegt. Dieser Testfall wird dann je nach gewählter Workload in verschiedene `TestDecorator` (vgl. Abbildung 4.2) gekapselt. Hier wird der Testfall zunächst in einen `ThinkTimeRepeatedTest` gekapselt, bei dem gesteuert wird, dass der Test wiederholt mit einer möglichen Bedenkzeit zwischen den einzelnen Ausführungen aufgerufen wird. Dabei wird die Bedenkzeit in der Klasse `RV_defaultUsageScenario_Workload` spezifiziert. Die gewünschte Anzahl der Iterationen soll in der

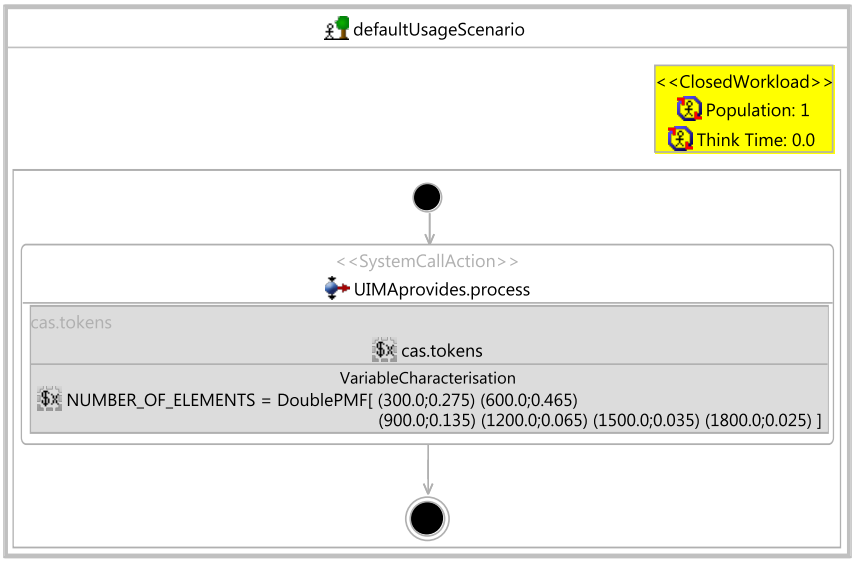

Abbildung B.1: PCM-Verwendungsmodell aus der UIMA-Fallstudie

Klasse `TestDuration` vom Tester festgelegt werden und ist standardmäßig auf 10 festgelegt. Dieser `ThinkTimeRepeatedTest` wird dann wiederum in einen `LoadTest`[1]gekapselt, der dafür sorgt, dass der Testfall mit der angegebenen Anzahl paralleler Benutzer ausgeführt wird. Der `LoadTest` wird dann in die TestSuite eingefügt und ausgeführt. Dabei erlaubt es die Testsuite, mithilfe der Methoden `addSetUpTests` und `addTearDownTests` weitere Testfälle vor- und nach den Tests für die modellierten Szenarien einzuschieben. Damit lässt sich beispielsweise die erforderliche Vorbedingung der Tests herstellen, wie von Güldali [27, 21] vorgeschlagen.

Der Testfall für das Szenario wird in einer separaten Datei generiert. Dabei ist insbesondere die Methode `testScenario` zu beachten, die in Listing 7 dargestellt ist. Wie im Verwendungsmodell (s. Abbildung B.1) spezifiziert, wird hier lediglich, die UIMA-Pipeline aufgerufen. Dazu wird zunächst eine Variable der Schnittstelle `UIMAprovides` spezifiziert. In der Testmethode `testScenario` wird dann die `process`-Methode aus der Schnittstelle `UIMAprovides` aufgerufen. Die Schnittstelle heißt in der Implementierung jedoch nicht lediglich `UIMAprovides`. Dieses Probleme hätte sich umgehen lassen, indem der Tester den Implementierungsnamen in einer Anmerkung

[1]Die Klasse `LoadTest` stammt aus dem JUnitPerf-Framework: http://www.clarkware.com/software/JUnitPerf.html (zuletzt besucht am 06.09.2014)

```
1  public static Test suite() {
2    int users = 0;
3    LoadTest workloadTest = null;
4    Test usageScenarioTest = null;
5    ThinkTimeRepeatedTest thinkTimeRepeatedTest =
       null;
6    Timer interArrivalTimer = null;
7    Timer thinkTimer = null;
8
9    SimultaneousLoadTestSuite
       simultaneousLoadTestSuite = new
       SimultaneousLoadTestSuite();
10
11   //building TestCase for Scenario:
       defaultUsageScenario
12   usageScenarioTest = new TestMethodFactory(
13     TestCase_defaultUsageScenario.class
14   , "testScenario"
15   );
16
17   users = 1;
18   thinkTimer=new RV_defaultUsageScenario_Workload()
       ;
19   thinkTimeRepeatedTest= new ThinkTimeRepeatedTest(
20     usageScenarioTest
21   , TestDuration.getIterations_defaultUsageScenario
       ()
22   , thinkTimer
23   );
24   workloadTest = new LoadTest(
25     thinkTimeRepeatedTest
26     , users
27   );
28
29   simultaneousLoadTestSuite.addTest(workloadTest);
30
31   TestSuite suite = new TestSuite();
32   addSetUpTests(suite);
33   suite.addTest(simultaneousLoadTestSuite);
34   addTearDownTests(suite);
35   return suite;
36 }
```

Listing 6: JUnit-Testsuite für den Testfall aus der UIMA-Fallstudie

im Komponenten-Repository spezifiziert, wie es in den Palladio-basierten Testfällen vorgesehen ist (vgl. Unterabschnitt 4.1.1).

```
1 UIMAprovides providedRole_UIMAprovides = null;
2
3 public void testScenario() {
4   try {
5     providedRole_UIMAprovides.process(
          ParameterInput.
          getValue_defaultUsageScenario_zC0H());
6   } catch (Exception e) {
7     e.printStackTrace();
8   }
9 }
```

Listing 7: Hauptmethode aus dem Testfall der UIMA-Fallstudie, in der die UIMA-Pipeline aufgerufen wird.

Der Eingabeparameter für den in Listing 7 dargestellten Aufruf der UIMA-Pipeline wird in der Klasse `ParameterInput` spezifiziert. Wie in Listing 8 zu sehen ist, wird eine Vorlage generiert, in der der Tester eigene Werte für den Eingabeparameter eingeben soll. In der „`PROTECTED REGION`“ kann der Tester Änderungen vornehmen, die bei einer wiederholten Generierung des JUnit-Testskripts nicht überschrieben werden. Wenngleich der JUnit-Testskript-Generator die stochastischen Ausdrücke des PCM übersetzen kann, werden Parameter nicht unterstützt. Insbesondere die aufwendige Umsetzung von Parametern mit komplexen Datenstrukturen fehlt in der prototypischen Implementierung. Zudem ist die Generierung von Testdaten ein schwieriges Feld, das in dieser Arbeit nicht diskutiert wird. Der Tester soll dementsprechend in der Klasse `ParameterInput` entweder vorbereitete Testdaten oder passende Generierungsstrategien für Testdaten zum Einsatz bringen.

```
1 public static JCas
     getValue_defaultUsageScenario_zC0H() {
2   /*PROTECTED REGION ID(...) ENABLED START*/
3   return null;
4   /*PROTECTED REGION END*/
5 }
```

Listing 8: Methode aus der Klasse `ParameterInput`, die den Wert für den Eingabeparameter des UIMA-Pipeline-Aufrufs zurückgibt