Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Stefan Münchow

Monitoring-Komponente zur Überwachung der Prozesse und zur statistischen Auswertung der Abläufe eines Messdatenmanagementsystems auf Basis von Werum HyperTest

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Stefan Münchow

Monitoring-Komponente zur Überwachung der Prozesse und zur statistischen Auswertung der Abläufe eines Messdatenmanagementsystems auf Basis von Werum HyperTest

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Stefan Sarstedt
Zweitgutachter : Prof. Dr. Olaf Zukunft

Abgegeben am 01. August 2011

Stefan Münchow

Thema der Bachelorarbeit

Monitoring-Komponente zur Überwachung der Prozesse und zur statistischen Auswertung der Abläufe eines Messdatenmanagementsystems auf Basis von Werum HyperTest

Stichworte

Verteiltes System, Messdatenmanagementsystem, Monitoring, Statistik, J2EE

Kurzzusammenfassung

Das Messdatenmanagementsystem Werum HyperTest ist eine verteilte Unternehmensanwendung auf Basis von J2EE (Java 2 Platform Enterprise Edition). In dieser Arbeit wird eine Monitoring-Komponente für das System entwickelt. Sie ermöglicht sowohl eine Überwachung von verteilten Prozessen als auch die Erhebung von Statistiken innerhalb des zentralen Servers. Wichtige Designziele sind dabei die leichte Erweiterbarkeit und Unabhängigkeit vom verwendeten Anwendungsserver.

Das Hauptaugenmerk liegt auf der Entwicklung einer allgemeinen Monitoring-Infrastruktur, die auf einfache Weise zur Erfassung zusätzlicher Informationen genutzt werden kann. Zur Konfiguration der Komponente und zur Anzeige der gewonnenen Daten wird eine webbasierte grafische Oberfläche realisiert. Abschließend wird die Auswirkung der Statistikerhebung auf die Performanz des Systems gemessen.

Stefan Münchow

Title of the paper

Monitoring component for process observation and statistical analysis of procedures within a test data management system based on Werum HyperTest

Keywords

Distributed system, test data management system, monitoring, statistics, J2EE

Abstract

The test data management system Werum HyperTest is a distributed enterprise application based on J2EE (Java 2 Platform Enterprise Edition). In this paper a monitoring component for the system is developed. It allows an observation of distributed processes as well as the gathering of statistics within the central server. Important goals of design are extensibility and independence of the employed application server.

The main focus is on the development of a monitoring infrastructure which can easily be used for collecting additional information. A web based graphical user interface for configuring the component and displaying the obtained data is realized. Finally, the influence of the statistics collection on the performance of the system is measured.

Inhaltsverzeichnis

1. Einführung **1**
1.1. Motivation . . . 1
1.2. Zielsetzung . . . 2
1.3. Aufbau der Arbeit . . . 3

2. Grundlagen **4**
2.1. Werum HyperTest . . . 4
2.1.1. Funktionsweise . . . 4
2.1.2. Systemarchitektur . . . 5
2.2. J2EE (Java 2 Platform Enterprise Edition) . . . 7
2.2.1. EJB (Enterprise Java Beans) . . . 9
2.2.2. JMS (Java Message Service) . . . 12
2.2.3. JMX (Java Management eXtensions) . . . 13
2.3. Interzeptor-Entwurfsmuster . . . 14
2.3.1. Struktur . . . 15
2.3.2. Vor- und Nachteile . . . 16
2.3.3. Realisierungen . . . 17
2.4. AOP (Aspektorientierte Programmierung) . . . 18
2.4.1. Grundbegriffe . . . 18
2.4.2. Technische Umsetzung . . . 19
2.4.3. Vor- und Nachteile . . . 20
2.4.4. Realisierungen . . . 21
2.5. GWT (Google Web Toolkit) . . . 22

3. Analyse **23**
3.1. Anforderungen . . . 23
3.1.1. Funktionale Anforderungen . . . 23
3.1.2. Nichtfunktionale Anforderungen . . . 25
3.1.3. Technische Rahmenbedingungen . . . 25
3.2. Designziele . . . 26
3.3. Vorhandene Lösungen & Produkte . . . 27
3.3.1. Glassbox . . . 27
3.3.2. JBoss StatisticsCollector . . . 28

3.3.3. RHQ / JBossON . . . 29
3.3.4. Zusammenfassung . . . 30

4. Konzeption **32**
4.1. Architektur der Komponente . . . 32
4.1.1. Komponenten & Schnittstellen . . . 32
4.1.2. Verteilungssicht . . . 34
4.2. Prozessüberwachung . . . 36
4.2.1. Lösungsansätze . . . 36
4.2.2. Feinentwurf der Watchdog-Subkomponente . . . 39
4.2.3. Ablauf einer Zustandsabfrage . . . 42
4.3. Statistikerhebung . . . 44
4.3.1. Feinentwurf der Statistics-Subkomponente . . . 44
4.3.2. Datenmodell für statistische Daten . . . 47
4.3.3. Filter für Parameter und Rückgabewerte . . . 48
4.3.4. Integration in das bestehende System . . . 50
4.3.5. Ablauf eines entfernten Aufrufs . . . 52
4.4. Grafische Oberfläche . . . 54

5. Realisierung **56**
5.1. Prozessüberwachung . . . 56
5.1.1. Zyklischer Versand von Anfragenachrichten . . . 56
5.1.2. Aufbau der JMS-Nachrichten . . . 58
5.1.3. Datenbanküberwachung . . . 59
5.1.4. Programmierschnittstelle der Subjekte . . . 59
5.2. Statistikerhebung . . . 60
5.2.1. Actions zur Erhebung von Daten . . . 60
5.2.2. Zyklische Erhebung und Speicherung von Daten . . . 61
5.2.3. Aspekt zur Erhebung von Methodenaufrufstatistiken . . . 61
5.2.4. Syntax der Filter-Ausdrücke . . . 64
5.3. Grafische Oberfläche . . . 65

6. Messung des Zeitverhaltens **68**
6.1. Testumgebung und Durchführung . . . 68
6.2. Auswertung der Ergebnisse . . . 69

7. Zusammenfassung **71**
7.1. Fazit . . . 71
7.2. Ausblick . . . 72

A. Layout der grafischen Oberfläche **74**

B. Parser für Filter-Ausdrücke 78

C. Inhalt der beiliegenden CD 80

Abkürzungsverzeichnis 81

Abbildungsverzeichnis 82

Quellcodeverzeichnis 84

Literaturverzeichnis 85

1. Einführung

In diesem Kapitel werden die Motivation und das Ziel dieser Arbeit erläutert. Außerdem wird der Aufbau der Arbeit vorgestellt.

1.1. Motivation

An moderne Unternehmensanwendungen wird eine Vielzahl von unterschiedlichen Anforderungen gestellt. Sie sollen die zentralen Geschäftsprozesse eines Unternehmens unterstützen, hochverfügbar und skalierbar sein. Die Realisierung solcher Anwendungen ist üblicherweise komplex und die einzelnen Systemkomponenten sind über mehrere physikalische Computer verteilt. Zusätzlich sollen derartige Systeme auf unterschiedlichen Plattformen lauffähig sein. Um all diese Anforderungen besser umsetzen zu können, verwenden viele Unternehmenslösungen die J2EE-Plattform (Java 2 Platform Enterprise Edition), welche eine Softwarearchitektur mit einigen standardisierten Diensten darstellt.

Neben den Ansprüchen, die von den Benutzern einer Anwendung gestellt werden, existieren auch Anforderungen an die Administration und Wartung eines solchen Systems. Das Verhalten verteilter Systeme ist häufig schwer nachvollziehbar und die Fehlersuche aufwändig. Insbesondere Fehler, die nicht offensichtlich sind, wie der Ausfall von verteilten Prozessen oder Performanzprobleme, sind schwierig zu erkennen. Vom Server erstellte Logdateien können zwar Hinweise geben, müssen jedoch vorher durch einen Entwickler ausgewertet werden und enthalten häufig nicht die gesuchte Information. So entsteht der Bedarf nach Monitoring-Werkzeugen, die Administratoren und Anwendungsentwicklern Informationen über den Zustand des Systems geben. Das können Informationen über die Verfügbarkeit verteilter Komponenten sein, insbesondere aber auch über Vorgänge innerhalb zentraler Server.

Das Messdatenmanagementsystem HyperTest der Firma Werum Software & Systems AG ist eine solche komplexe Unternehmensanwendung. Sie hat das Ziel, die Verwaltung und effiziente Nutzung großer Mengen von Messdaten und begleitender Dokumente zu vereinfachen. Zu diesem Zweck werden Messdaten verschiedener Dateiformate in HyperTest verwaltet, mit relevanten Dokumenten verknüpft und mit Metainformationen versehen. Dadurch können Daten einfacher aufgefunden werden. Außerdem sind Zusammenhänge zwischen gespeicherten Daten erkennbar und ihre Entstehung nachvollziehbar. Die Praxis hat gezeigt,

dass auch HyperTest ein Monitoring-Werkzeug benötigt, um auftretende Probleme einfacher erkennen und beheben zu können.

1.2. Zielsetzung

In dieser Arbeit sollen die Möglichkeiten untersucht werden, eine verteilte Anwendung auf Basis von J2EE zu überwachen. Das Ziel besteht darin, eine Monitoring-Komponente zu entwickeln, die Informationen über das System erhebt und bereitstellt. Das wird im Rahmen dieser Arbeit für das Messdatenmanagementsystem HyperTest durchgeführt. Das Monitoring soll dabei zwei Funktionen erfüllen:

Prozessüberwachung
: Das System besteht aus mehreren über verschiedene physikalische Computer verteilten Prozessen. In der Regel existieren ein zentraler Server, eine Datenbank und mehrere Klienten für spezielle Aufgaben, wie z. B. die automatisierte Übernahme von Messdaten und begleitenden Dokumenten in das System oder die automatisierte Auswertung ausgewählter Daten. Es hat starke Auswirkungen auf das Gesamtsystem, wenn eine dieser Komponenten ausfällt. Daher soll es möglich sein, den aktuellen Zustand der einzelnen Prozesse zur Laufzeit zu ermitteln.

Statistikerhebung
: Der HyperTest-Server ist die zentrale Komponente des Systems. Hier fallen Performanzprobleme am stärksten ins Gewicht. Daher sollen innerhalb des Servers statistische Daten erhoben werden, mit deren Hilfe Aussagen über das Zeitverhalten und die Auslastung des Systems getroffen werden können. Interessant sind hier z. B. der Speicherbedarf des Servers, die Anzahl aktuell angemeldeter Benutzer und die Ausführungsdauer von Operationen.

Diese beiden Funktionen sollen zur Laufzeit zu- und abschaltbar sein. Außerdem soll das Monitoring die Performanz des Systems möglichst wenig belasten. Alle relevanten Informationen sollen auf einer grafischen Oberfläche visualisiert werden, um einen schnellen Überblick über die Daten zu ermöglichen.

Die Lösung wird anhand der aktuellen Version 2.10.6 von HyperTest entwickelt. Natürlich sind dabei einige Spezifika des Systems zu beachten. Die zu entwickelnde Lösung soll jedoch so allgemein konzipiert werden, dass sie sich möglichst einfach auf ähnliche Anwendungen übertragen lässt. Das Hauptaugenmerk dieser Arbeit liegt auf der Entwicklung einer Monitoring-Infrastruktur, die Anwendungsentwickler einfach für die Erfassung weiterer Informationen verwenden können.

1.3. Aufbau der Arbeit

Diese Ausarbeitung besteht neben der Einleitung aus sechs weiteren Kapiteln. In Kapitel 2 werden zunächst einige Grundlagen erläutert, die für das weitere Verständnis dieser Arbeit notwendig sind. Hier werden die Funktionsweise und Struktur von HyperTest kurz vorgestellt. Als nächstes werden die verwendeten Technologien J2EE und EJB (Enterprise Java Beans) erläutert. Abschließend werden mit dem Interzeptor-Entwurfsmuster und der aspektorientierten Programmierung zwei Möglichkeiten erläutert, Dienste transparent zu einem bestehenden System hinzuzufügen.

Kapitel 3 beschäftigt sich mit der genauen Analyse der Problemstellung. Neben der Anforderungsanalyse werden Designziele definiert, die später zur Bewertung von Lösungsansätzen herangezogen werden. Zuletzt werden mehrere vorhandene Monitoring-Lösungen betrachtet und bewertet.

Kapitel 4 enthält mit der Konzeption der Monitoring-Komponente den Kern dieser Arbeit. Hier werden verschiedene Lösungsmöglichkeiten entwickelt und miteinander verglichen. Sie werden im Hinblick auf die Anforderungen und Designziele aus Kapitel 3 bewertet. Anschließend wird eine geeignete Lösung ausgewählt. In Kapitel 5 werden einige ausgewählte Aspekte der Realisierung vorgestellt.

In Kapitel 6 wird das Zeitverhalten der realisierten Statistikerhebung gemessen. Dazu wird zunächst eine Testumgebung definiert. Danach werden die ermittelten Messergebnisse ausgewertet.

Im abschließenden Kapitel 7 werden das Vorgehen und die erzielten Ergebnisse zusammengefasst. Außerdem wird eine Bewertung der Ergebnisse vorgenommen und ein Ausblick auf mögliche Erweiterungen und Verbesserungen der Monitoring-Komponente gegeben.

2. Grundlagen

In diesem Kapitel werden die Grundlagen erläutert, die für das Verständnis dieser Arbeit notwendig sind. Zunächst werden die Funktionsweise und Struktur von HyperTest beschrieben. Danach werden die verwendeten Technologien J2EE und EJB vorgestellt. Zuletzt werden das Interzeptor-Entwurfsmuster und das Konzept der aspektorientierten Programmierung erläutert.

2.1. Werum HyperTest

In diesem Abschnitt wird das zugrunde liegende System HyperTest erläutert. Die exemplarische Realisierung der entwickelten Lösung wird später für dieses System durchgeführt.

2.1.1. Funktionsweise

Zur Erhebung und Auswertung von Messdaten werden häufig verschiedene bewährte Werkzeuge, wie z. B. DIAdem von National Instruments, verwendet. Diese legen Daten in spezifischen Dateiformaten und Verzeichnisstrukturen ab, wodurch eine gezielte Suche nach Informationen erschwert wird. HyperTest ermöglicht die Verwaltung von Dateien verschiedener Dateiformate zusammen mit Informationen über den Kontext, in dem sie entstanden sind, in einem einzigen System. Dafür werden die Dateien nach HyperTest importiert und anschließend über das System verwaltet. So können Benutzer auf eine einheitliche Weise auf alle gespeicherten Daten zugreifen. Die tatsächliche Ablage der Dateien auf dem Server ist dem Benutzer dabei unbekannt. Zum Öffnen von Dateien aus HyperTest heraus werden geeignete Anwendungen, wie z. B. Microsoft Office, verwendet (vgl. Werum 2011, S. 10 f., 53).

Alle Daten werden in einer Navigationsstruktur, ähnlich einem Dateisystem, organisiert. Sie werden als Objekte eines bestimmten Objekttyps gespeichert. Dieser bestimmt die Metadaten eines Objekts und die Operationen, die auf ihm ausgeführt werden können. Ein Objekt ist entweder ein Container, der weitere Objekte enthalten kann, oder eine Datei. Objekte können mit anderen verknüpft werden, um so Zusammenhänge zwischen Daten nachvollziehbar zu machen. Dabei bestimmt der Objekttyp die erlaubten Beziehungen (vgl. Werum

2011, S. 32 f.). Ein Beispiel für einen Objekttyp sind Prüfaufträge, die angelegt werden, bevor Messdaten erhoben werden. Die Messdaten werden später nach HyperTest importiert und den jeweiligen Prüfaufträgen zugeordnet.

Messdaten und begleitende Dokumente, wie z. B. Bilder oder Dokumente zur Beschreibung des Testaufbaus, können automatisiert nach HyperTest importiert werden. Dabei werden Informationen aus Datei- und Verzeichnisnamen oder aus den Messdatendateien als Metainformationen in den erzeugten Objekten gespeichert oder zur Zuordnung der Daten zu einem bestimmten Objekt verwendet. Zusätzlich können Benutzer Kommentare und Berichte hinzufügen. So können Messdaten mit zusätzlichen Angaben zu Prüfstandskonfigurationen, Prüfparametern etc. versehen werden (vgl. Werum 2011, S. 13, 151 f.).

Das Auffinden von Daten wird durch Suchfunktionen unterstützt. Eine Metadatensuche durchsucht die Metainformationen von Objekten in der Datenbank nach vorgegebenen Kriterien. Zusätzlich können ausgewählte Dokumente und Metainformationen für eine Volltextsuche indiziert werden (vgl. Werum 2011, S. 80 ff.). Benutzer von HyperTest verwenden eine dialogbasierte grafische Oberfläche (im Folgenden nur *GUI-Klient* genannt), die alle Funktionen des Systems leicht zugänglich für den Benutzer zur Verfügung stellt. Außerdem existiert ein API (Application Programming Interface), mit dessen Hilfe Kunden eigene Werkzeuge entwickeln können, die HyperTest verwenden (vgl. Werum 2011, S. 13, 15 ff.).

Eine weitere Funktion von HyperTest ist die Durchführung automatisierter Auswertungen. Dafür wählt ein Benutzer die zu verwendenden Eingangsdaten und die Art der Analyse aus. HyperTest stellt die Daten für ein Auswertungswerkzeug zur Verfügung und startet den Vorgang. Anschließend überwacht das System die korrekte Ausführung der Auswertung und fügt die Ergebnisse, z. B. generierte PDF-Dateien, in den Datenbestand auf dem Server ein (vgl. Werum 2011, S. 13).

Neben den hier genannten Funktionen ermöglicht HyperTest die Verwaltung weiterer für Messungen relevanter Daten, wie z. B. Prüfstände und Messmittel. Über ein Rechtesystem wird der Zugriff auf die Daten für verschiedene Benutzer kontrolliert. Eine vollständige Beschreibung aller Funktionen von HyperTest ist in Werum (2011) zu finden.

2.1.2. Systemarchitektur

HyperTest ist als verteiltes System auf Basis von J2EE realisiert. Eine Installation besteht aus einem zentralen Server und beliebig vielen Klienten, die sich mit dem Server verbinden können. Dabei kann zwischen Klienten für spezielle Aufgaben, wie z. B. der automatisierten Auswertung von Daten, und den GUI-Klienten unterschieden werden. Eine mögliche Verteilung der Prozesse ist in Abbildung 2.1 dargestellt.

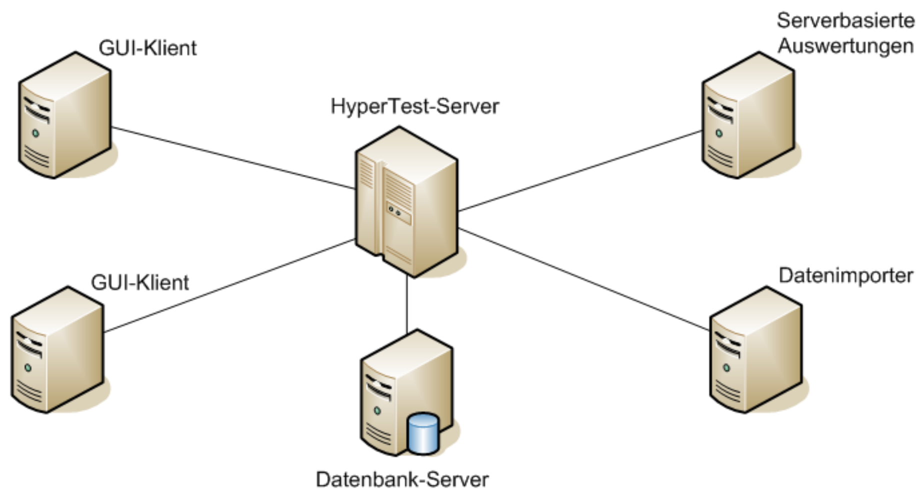

Abbildung 2.1.: Übersicht über das HyperTest-System

Benutzer von HyperTest müssen keine Kenntnis darüber besitzen, dass Server, Datenbank und weitere Prozesse auf verschiedene physikalische Computer verteilt sind. Das System ist also *ortstransparent* (vgl. Tanenbaum und van Steen 2008, S. 21 f.). Die wichtigsten Bestandteile des Systems werden im Folgenden erläutert (vgl. Werum 2011, S. 12 f.):

HyperTest-Server Der zentrale Server enthält die gesamte geschäftliche Logik des Systems. Die Architektur basiert auf J2EE Version 1.4 und EJB Version 2.1. Beide Technologien werden im folgenden Abschnitt vorgestellt. Alle Zugriffe auf gespeicherte Daten erfolgen über den Server.

GUI-Klient Der GUI-Klient stellt die grafische Benutzerschnittstelle für das System dar. Er ist als Java-Anwendung mit Swing realisiert. Die Verteilung erfolgt über Java WebStart.

Serverbasierte Auswertungen Diese Komponente dient dazu, automatisierte Auswertungen auf vorher ausgewählten Daten durchzuführen. Der HyperTest-Server stellt dem Prozess die benötigten Daten zu Verfügung, welche dieser an das entsprechende Auswertungswerkzeug übergibt. Die Ergebnisse, z. B. Berichte im PDF-Format, überträgt der Prozess an den HyperTest-Server. Dort werden sie an geeigneter Stelle zu den bestehenden Daten hinzugefügt (vgl. Werum 2011, S. 13).

Datenimporter Diese Komponente dient zum automatisierten Import von Dateien in das HyperTest-System. Sie verwendet Konverter für verschiedene Dateiformate. Diese extrahieren zunächst Metainformationen aus den Dateien und generieren daraus eine vom Dateiformat unabhängige XML-Datei (eXtensible Markup Language). Der Datenimporter überträgt die Dateien auf den HyperTest-Server und nutzt die Informationen aus der XML-Datei zur Erzeugung der entsprechenden Objekte im System (vgl. Werum 2011, S. 151 f.).

2.2. J2EE (Java 2 Platform Enterprise Edition)

Die J2EE-Plattform definiert eine Standardarchitektur zur Entwicklung von mehrschichtigen, verteilten Java-Anwendungen. Sie legt den grundsätzlichen Aufbau einer Anwendung fest. Außerdem definiert sie Dienste, die innerhalb einer Anwendung verwendet werden können. Diese Dienste werden durch klar definierte Schnittstellen beschrieben, die von Herstellern verschiedener Produkte realisiert werden können. Dies garantiert die Austauschbarkeit verschiedener Komponenten eines Systems. So realisieren z. B. Datenbanken verschiedener Hersteller eine JDBC-Schnittstelle (Java Database Connectivity), wodurch der Anwendungscode unabhängig von einem konkreten Produkt bleibt.

Eine J2EE-Anwendung läuft üblicherweise innerhalb eines *Anwendungsservers*. Ein solcher Anwendungsserver ist eine Realisierung der J2EE-Spezifikation und stellt die Infrastruktur und die Standarddienste für Anwendungen bereit (vgl. Sun Microsystems 2003b, S. 5 ff.).

Eine Übersicht über den Aufbau der J2EE-Architektur wird in Abbildung 2.2 gezeigt. Die Anwendung wird in logische *Container* aufgeteilt, von denen jeder eine Laufzeitumgebung für einen bestimmten Typ von Anwendungskomponenten darstellt. Dies können z. B. Enterprise Java Beans, Servlets oder Applets sein. Anwendungskomponenten kommunizieren dabei niemals direkt miteinander, sondern ausschließlich über Methoden des jeweiligen Containers. Dadurch kann der Anwendungsserver transparent Dienste, wie Benutzerauthentifikation und Transaktionssteuerung, zu den Aufrufen hinzufügen. Eine J2EE-Anwendung besteht aus vier Arten von Containern (vgl. Sun Microsystems 2003b, S. 5-8):

Applet-Container Enthält Applets, welche in einem Browser ablaufen und als grafische Oberfläche für eine J2EE-Anwendung fungieren können.

Application-Container Enthält die Anwendungsklienten. Diese sind meist J2SE-Anwendungen (Java 2 Platform Standard Edition), welche auf einem Arbeitsplatz-Computer ausgeführt werden. Sie werden von den Benutzern des Systems verwendet und ermöglichen den Zugriff auf die Geschäftslogik im Server.

EJB-Container Enthält die Geschäftslogik der Anwendung. Diese wird mithilfe von Enterprise Java Beans modelliert. EJBs werden in Abschnitt 2.2.1 genauer erläutert.

Web-Container Enthält alle Anwendungskomponenten, die üblicherweise von Webbrowsern aus angesprochen werden. Dies können Servlets, JSP-Seiten (Java Server Pages) und andere Komponenten sein, die HTML (HyperText Markup Language) oder XML generieren und zum Klienten übertragen.

Jedes J2EE-konforme Produkt muss die in der Spezifikation festgelegten Dienste bzw. Schnittstellen implementieren. Einige dieser Dienste werden im Folgenden kurz genannt (vgl. Sun Microsystems 2003b, S. 9-14).

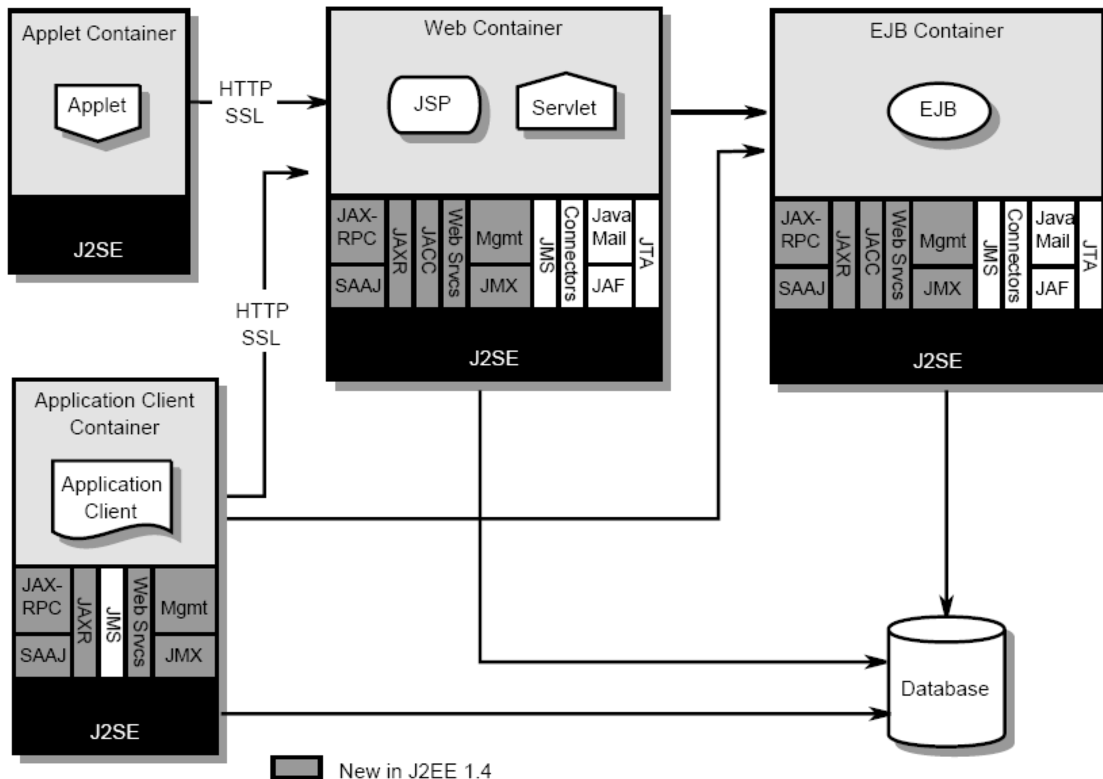

Abbildung 2.2.: Übersicht über die J2EE-Architektur (Sun Microsystems 2003b, S. 6)

JTA (Java Transaction API)
Dienst zur Verwaltung von (verteilten) Transaktionen.

JDBC (Java DataBase Connectivity)
Einheitliche Schnittstelle zum Zugriff auf (relationale) Datenbanken. Diese wird von verschiedenen Datenbankherstellern realisiert.

JMS (Java Message Service)
Dienst zur asynchronen Kommunikation über eine nachrichtenorientierte Middleware. Dieser wird in Abschnitt 2.2.2 genauer erläutert.

JNDI (Java Naming and Directory Interface)
Programmierschnittstelle zum Zugriff auf Namens- und Verzeichnisdienste. Definiert außerdem einen eigenen Namensdienst.

JMX (Java Management eXtensions)
Programmierschnittstelle für das Management und die Überwachung von Java-Anwendungen. JMX wird in Abschnitt 2.2.3 genauer erläutert.

Eine komplette Übersicht der Standarddienste ist in Sun Microsystems (2003b, S. 9-14) zu finden.

2.2.1. EJB (Enterprise Java Beans)

EJB ist eine komponentenbasierte Architektur zur Realisierung von verteilten Geschäftsanwendungen in Java. Sie legt den Aufbau von Komponenten, genannt *Java Beans*, genau fest. Durch die Standardisierung können diese theoretisch auf jedem J2EE-konformen Anwendungsserver betrieben werden (vgl. Sun Microsystems 2003a, S. 27). Mit der Standardisierung von EJB werden u. a. folgende Ziele verfolgt (vgl. Sun Microsystems 2003a, S. 31 f., S. 414 f.):

- Entwickler werden entlastet, damit sie sich auf die Umsetzung der Geschäftslogik konzentrieren können. „Low-Level"-Funktionen wie Transaktionssteuerung, Persistenz und die Synchronisation nebenläufiger Zugriffe werden als Dienste vom EJB-Container zur Verfügung gestellt.
- EJB-Komponenten sind auf verschiedenen Anwendungsservern lauffähig, ohne sie verändern oder neu kompilieren zu müssen.
- Beans können mithilfe von Werkzeugen verschiedener Hersteller entwickelt und eingesetzt werden. Komponenten, die mit verschiedenen Werkzeugen erstellt wurden, können zur Laufzeit über Standardschnittstellen zusammenarbeiten.
- Die Zusammenarbeit von EJB-Komponenten mit Nicht-Java-Komponenten wird durch die Verwendung eines IIOP-basierten (Internet Inter-ORB Protocol) Protokolls ermöglicht. Außerdem wird die Entwicklung und der Einsatz von Webservices erleichtert.

EJB Version 2.1

In EJB Version 2.1 werden drei Arten von EJBs unterschieden (vgl. Sun Microsystems 2003a, S. 49 ff.):

Entity-Beans

Sie repräsentieren *Entitäten*, die in der Datenbank persistiert werden. Änderungen werden transaktional ausgeführt und bei einem Serverabsturz wird der letzte Stand wiederhergestellt. Eine Entity-Bean entspricht einer Tabellenzeile einer relationalen Datenbank.

Session-Beans
: Sie enthalten entitätenübergreifende Geschäftslogik. Man unterscheidet zwischen *zustandslosen* und *zustandsbehafteten* Session-Beans. Zustandsbehaftete Session-Beans können einen Konversationszustand für einen Klienten verwalten, zustandslose Session-Beans hingegen speichern keine Informationen. Beide werden nicht in der Datenbank persistiert.

Message-Driven-Beans
: Sie werden durch asynchrone Nachrichten über JMS aufgerufen. Ansonsten entsprechen sie zustandslosen Session-Beans. JMS wird in Abschnitt 2.2.2 genauer vorgestellt.

Enterprise-Beans implementieren festgelegte Callback-Methoden, die vom Anwendungsserver beim Eintreten bestimmter Lebenszyklus-Ereignisse, wie z. B. der Persistierung einer Entity-Bean in der Datenbank, aufgerufen werden. Diese können dazu genutzt werden, bei bestimmten Ereignissen zusätzliche Logik auszuführen.

Jede Session- oder Entity-Bean besteht aus mehreren Klassen und Schnittstellen. Diese müssen von speziellen Schnittstellen, wie z. B. *javax.ejb.EJBObject* oder *javax.ejb.EJBHome*, abgeleitet sein. Die Bestandteile einer Session- oder Entity-Bean sind folgende (vgl. Monson-Haefel 2002, S. 29 ff.):

Entferntes / lokales Interface
: Hier sind die Geschäftsmethoden der Komponente definiert. Durch das entfernte und lokale Interface wird unterschieden, von wo Methoden aufgerufen werden können. Methoden im entfernten Interface können von außerhalb des EJB-Containers über entfernte Aufrufe angesprochen werden. Dagegen können Methoden im lokalen Interface nur innerhalb des gleichen EJB-Containers aufgerufen werden.

Entferntes / lokales Home-Interface
: Hier sind Methoden zur Verwaltung, d. h. zum Erzeugen, Suchen und Löschen von Beans definiert. Auch dabei wird zwischen lokalen und entfernten Aufrufen unterschieden.

Bean-Klasse
: In dieser Klasse befindet sich die konkrete Implementierung einer Bean. Sie implementiert keine der o. g. Schnittstellen direkt, muss jedoch Methoden enthalten, deren Signaturen denen der Methoden im entfernten und lokalen (Home-)Interface entsprechen.

Primärschlüssel-Klasse
: Optional kann für Entity-Beans eine Primärschlüsselklasse definiert werden. Sie dient als systemweit eindeutiger Schlüssel für die Entität und muss serialisierbar sein.

Eine Message-Driven-Bean benötigt keine Komponentenschnittstellen, da sie nicht von außen aufgerufen wird. Stattdessen implementiert sie die Schnittstelle *MessageListener*. Diese enthält eine `onMessage`-Methode, die durch den EJB-Container beim Eintreffen einer Nachricht aufgerufen wird.

Für den Einsatz der Komponenten ist ein sogenannter Deployment-Deskriptor zu erstellen. Dieser ist üblicherweise eine deklarative Beschreibung einer oder mehrerer EJBs im XML-Format. Er beschreibt die Beans und konfiguriert die von ihnen verwendeten Dienste. Auf diese Weise bleibt die fachliche Logik einer Komponente unabhängig von der Umgebung, in der sie eingesetzt wird (vgl. Sun Microsystems 2003a, S. 501 ff.).

Da dieses Modell die Definition vieler Klassen und Schnittstellen vorschreibt, die nur indirekt miteinander in Verbindung stehen, wirkt es überaus komplex. Daher wurde die Architektur in den folgenden EJB-Versionen stark vereinfacht.

EJB Version 3

In EJB Version 3 wurde die Architektur vollständig überarbeitet. Die wichtigsten Neuerungen werden im Folgenden genannt (vgl. Sun Microsystems 2006, S. 25 f.):

- Es werden Java-Annotationen verwendet, um die Anzahl benötigter Klassen und Schnittstellen zu verringern und die Deployment-Deskriptoren zu vereinfachen.
- Home-Interfaces sind nicht mehr erforderlich. Komponentenschnittstellen müssen keine besonderen Schnittstellen, wie z. B. *EJBObject*, mehr erweitern.
- Statt Entity-Beans werden einfache Java-Objekte und das JPA (Java Persistence API) verwendet.
- Callback-Methoden in den Beans müssen nur noch bei Bedarf implementiert werden.
- Es wurden standardisierte Interzeptoren für Beans eingeführt. Interzeptoren werden in Abschnitt 2.3 genauer vorgestellt.

Timer-Service

In der EJB-Spezifikation Version 2.1 wird ein Timer-Dienst eingeführt. Dieser erlaubt die zeitgesteuerte Ausführung von Operationen innerhalb von Enterprise-Beans. Dafür werden die Schnittstellen *Timer*, *TimerService*, *TimerHandle* und *TimedObject* definiert.

Die ersten drei Schnittstellen müssen vom EJB-Container implementiert werden. Die *Timer*-Schnittstelle bietet Methoden an, um Informationen über einen Timer zu ermitteln oder diesen zu stoppen. Die angebotene `getInfo`-Methode liefert ein beliebiges serialisierbares

Objekt zurück, welches dem Timer bei der Erzeugung übergeben wird. Es kann dazu genutzt werden, Informationen zu einem Timer zu speichern. Ein *TimerHandle* ist lediglich eine Referenz auf einen Timer. Die *TimerService*-Schnittstelle bietet Methoden an, um einmalig oder periodisch ablaufende Timer zu erzeugen oder alle vorhandenen Timer zu ermitteln.

Eine Enterprise-Bean, die Gebrauch vom Timer-Dienst machen möchte, muss die Schnittstelle *TimedObject* implementieren. Diese definiert eine `ejbTimeout`-Methode. Sie wird bei Ablauf eines Timers aufgerufen und bekommt den entsprechenden Timer als Parameter übergeben. Standardmäßig wird beim Aufruf der `ejbTimeout`-Methode eine neue Transaktion gestartet, dies kann jedoch bei Bedarf geändert werden (vgl. Sun Microsystems 2003a, S. 493 ff.).

2.2.2. JMS (Java Message Service)

Das Java Message Service API beschreibt eine Menge von standardisierten Schnittstellen, über die ein Java-Programm auf eine MOM (Message Oriented Middleware) zugreifen kann. Eine MOM ermöglicht asynchrone und verlässliche Kommunikation zwischen verschiedenen Prozessen über einen Nachrichtenkanal. Die Prozesse, die JMS verwenden, werden als *JMS-Klienten* bezeichnet. Die Realisierung der JMS-Schnittstellen wird durch einen sogenannten *JMS-Provider* bereitgestellt (vgl. Sun Microsystems 2002b, S. 13 f.).

Ein Nachrichtenkanal, über den Nachrichten versendet werden, wird als *JMS-Destination* bezeichnet. Es sind zwei verschiedene Arten von Nachrichtenkanälen definiert, die *Queue* und das *Topic*. JMS sieht zwei verschiedene Kommunikationsmodelle, sogenannte *JMS-Domains*, vor:

Punkt-zu-Punkt
: Bei der Punkt-zu-Punkt-Kommunikation übermittelt ein Sender eine Nachricht an eine Queue. Die Nachricht wird von einem einzigen Empfänger entgegengenommen. Ist zum Zeitpunkt der Zustellung kein Empfänger verfügbar, so bleibt die Nachricht solange in der Queue, bis sich ein Empfänger anmeldet und sie abruft (vgl. Sun Microsystems 2002b, S. 75-78).

Publish / Subscribe
: Hier sendet ein Herausgeber eine Nachricht an ein Topic, welche daraufhin von allen am Topic angemeldeten Abonnenten empfangen wird. Normalerweise werden Nachrichten hier nicht zwischengespeichert, wodurch eine gesendete Nachricht u.U. von keinem Abonnenten empfangen wird. Eine Ausnahme bilden sogenannte Durable Subscriptions, bei denen die Nachricht zwischengespeichert wird (vgl. Sun Microsystems 2002b, S. 79-85).

In Sun Microsystems (2003b, S. 103 f.) ist definiert, dass jeder J2EE-kompatible Anwendungsserver einen JMS-Provider, d. h. eine Realisierung des JMS-Standards, bereitstellen muss. Im Falle des JBoss-Anwendungsservers ist dies z. B. das JBoss Messaging (auch JBossMQ genannt).

2.2.3. JMX (Java Management eXtensions)

JMX ist eine Spezifikation zur Verwaltung und Überwachung von Ressourcen in Java-Anwendungen. Eine Ressource kann dabei ein beliebiges Artefakt sein, das in Java entwickelt wurde oder für das zumindest eine Java-Schnittstelle existiert (vgl. Sun Microsystems 2002a, S. 21 f.). Laut Langner und Reiberg war ein wichtiger Grund für die Einführung von JMX, dass Konfigurationsparameter vorher an verschiedenen Stellen eines Systems, wie Deployment-Deskriptoren und der Datenbank, abgelegt wurden. Dadurch waren viele Systeme schwer administrierbar und konnten zur Laufzeit nicht konfiguriert werden (vgl. Langner und Reiberg 2006, S. 39). JMX versucht dies zu vereinheitlichen. Die JMX-Architektur ist in drei Ebenen aufgeteilt:

Instrumentations-Ebene
: Zu dieser Ebene gehören die verwalteten Ressourcen. Sie werden durch sogenannte *MBeans* (Managed Beans) repräsentiert. Eine MBean ist eine einfache Java-Klasse, die eine Management-Schnittstelle implementiert (vgl. Sun Microsystems 2002a, S. 21 f.). Diese enthält Methoden für alle Attribute und Operationen, die eine verwaltete Ressource bereitstellt. Es werden verschiedene Typen von MBeans unterschieden, die sich im Wesentlichen dadurch unterscheiden, wie ihre Management-Schnittstellen definiert werden. Genauere Informationen dazu sind in Sun Microsystems (2002a, S. 26) nachzulesen.

Agenten-Ebene
: Ein JMX-Agent enthält einen *MBeanServer*, eine zentrale Instanz zur Verwaltung aller von diesem Agenten verwalteten MBeans. Zusätzlich bietet er einige Dienste an, welche die Arbeit mit MBeans vereinfachen sollen. Alle MBeans werden beim *MBeanServer* unter einem eindeutigen Namen registriert und können dort mithilfe des Namens abgerufen werden (vgl. Sun Microsystems 2002a, S. 22 f.).

Ebene der verteilten Dienste
: Diese Ebene stellt mithilfe von Konnektoren eine einheitliche Schnittstelle für den Zugriff auf die Agenten-Ebene bereit (vgl. Sun Microsystems 2002a, S. 23 f.). Standardmäßig existiert z. B. ein Konnektor für den Zugriff über HTTP (HyperText Transfer Protocol).

MBeans können zur Laufzeit des Systems über den *MBeanServer* abgerufen werden. Außerdem können die Methoden ihrer Management-Schnittstelle aufgerufen werden. Dies ermöglicht die Konfiguration von MBeans zur Laufzeit, indem z. B. ein Attribut über eine Setter-Methode verändert wird. Neben der Verwaltung von Ressourcen bietet JMX einige standardisierte Dienste. Dazu gehören ein Timer-Dienst, der zeitgesteuerte Operationen auf MBeans auslösen, und ein Beziehungsdienst, der Beziehungen zwischen MBeans verwalten kann. Durch diesen ist es möglich, Abhängigkeiten zwischen MBeans zu definieren. Zudem existiert ein Monitor-Dienst, der die Überwachung einzelner Attribute von MBeans ermöglicht (vgl. Kreger u. a. 2003, S. 69). Monitore werden in Kreger u. a. (2003, S. 264 ff.) genau beschrieben.

Ein wichtiger Unterschied zwischen MBeans und EJBs besteht darin, dass MBeans während des Startvorgangs des Anwendungsservers erzeugt und initialisiert werden. EJBs hingegen werden erst aktiv, sobald ein Klient eine ihrer entfernten Methoden aufruft. Deshalb können MBeans für Aufgaben genutzt werden, für die EJBs ungeeignet wären, wie z. B. die Initialisierung von Datenbanktabellen oder die Realisierung eines einfachen Zwischenspeichers für bestimmte Objekte (vgl. Langner und Reiberg 2006, S. 40 ff., 45 ff.).

Die JMX-Realisierung des JBoss Anwendungsservers erlaubt die Konfiguration von MBeans über eine XML-Datei. Dabei können Initialwerte für die Attribute der MBean deklarativ angegeben werden. Sie werden dann beim Start des Servers gesetzt.

2.3. Interzeptor-Entwurfsmuster

Die meisten Frameworks sollen *universell*, d. h. für eine Vielzahl verschiedenartiger Anwendungen, einsetzbar sein. Vor allem im Middleware-Bereich ist dieser Anspruch sehr ausgeprägt. Allerdings benötigen Anwendungen, die eine solche Middleware verwenden, speziell ausgewählte Dienste, wie z. B. Transaktionssteuerung, Sicherheitsprüfungen, Logging oder Monitoring.

In Schmidt (2000, S. 109 f.) werden zwei mögliche Ansätze beschrieben, nach denen Anbieter einer Middleware ihr Produkt konzipieren können, um dieser Anforderung gerecht zu werden:

- Alle Dienste, die von einer Anwendung benötigt werden könnten, werden in die Middleware integriert. Diese wird dadurch jedoch u.U. schwer wartbar und langsam. Zudem ist es nicht möglich, alle von Anwendungen benötigten Dienste zur Entwicklungszeit eines Frameworks vorauszusehen.
- Die Middleware wird möglichst leichtgewichtig konzipiert und bietet standardmäßig keinerlei Dienste an. Infolgedessen müssen Anwendungsprogrammierer alle benötigten

Dienste selbst im Anwendungscode implementieren. Dieser wird dadurch komplexer und enthält Logik, die keinen direkten Bezug zur fachlichen Logik besitzt. Der Zugriff auf Middleware-Interna durch solche Dienste ist nur schwer zu realisieren.

Eine Lösung für die o. g. Probleme bietet das Interzeptor-Entwurfsmuster. Es ermöglicht die transparente Erweiterung eines Frameworks um zusätzliche Dienste, die in Form von Interzeptoren bereitgestellt werden. Diese implementieren eine spezielle Schnittstelle des Frameworks. Beim Auftreten von Ereignissen benachrichtigt das Framework die Dienste über diese Schnittstelle. Zusätzlich kann die Middleware interne Informationen an die Dienste übergeben, wodurch diese auf Teile der Abarbeitung zugreifen oder diese beeinflussen können (vgl. Schmidt 2000, S. 111).

2.3.1. Struktur

Eine Übersicht über die Struktur des Interzeptor-Entwurfsmusters ist in Abbildung 2.3 zu finden. Für eine Gruppe von Ereignissen wird eine Interzeptor-Schnittstelle mit Callback-Methoden definiert. Diese wird von den konkreten Interzeptoren implementiert. Für jeden Interzeptor-Typ wird außerdem eine Dispatcher-Klasse vorgesehen, die eine Registrierung der Interzeptoren beim Framework ermöglicht. Sobald ein Ereignis eintritt, benachrichtigt das Framework den zuständigen Dispatcher. Dieser ruft daraufhin alle an ihm registrierten Interzeptoren auf. Für den Zugriff auf interne Informationen des Frameworks können Kontext-Klassen definiert werden, welche den Interzeptoren übergeben werden. Auf diese Weise können Interzeptoren das Verhalten des Frameworks verändern (vgl. Schmidt 2000, S. 112-116).

Bei der Definition der Kontext-Klassen bestehen grundsätzlich zwei Möglichkeiten (vgl. Schmidt 2000, S. 122):

- Es wird eine allgemeine Kontext-Klasse definiert, die stets alle verfügbaren Informationen enthält. Sie ist komplexer in der Verwendung, allerdings muss der Entwickler nur eine Klasse kennen.
- Es werden spezialisierte Kontext-Klassen für jedes Ereignis definiert. Sie enthalten jeweils nur die für das Ereignis relevanten Informationen und sind dadurch einfach in der Verwendung. Allerdings muss der Entwickler mehrere Klassen kennen und verstehen.

Kontext-Objekte können einem Interzeptor bei seiner Registrierung am Dispatcher übergeben werden. In diesem Fall enthalten sie nur statische Informationen, welche sich im Lebenszyklus des Interzeptors nicht mehr verändern. Die Übergabe eines neu erzeugten Kontext-Objekts bei Eintritt eines Ereignisses bietet detailliertere Informationen über das Ereignis und den aktuellen Zustand des Frameworks, bringt jedoch ein gewisses Maß an zusätzlichem Overhead mit sich (vgl. Schmidt 2000, S. 123).

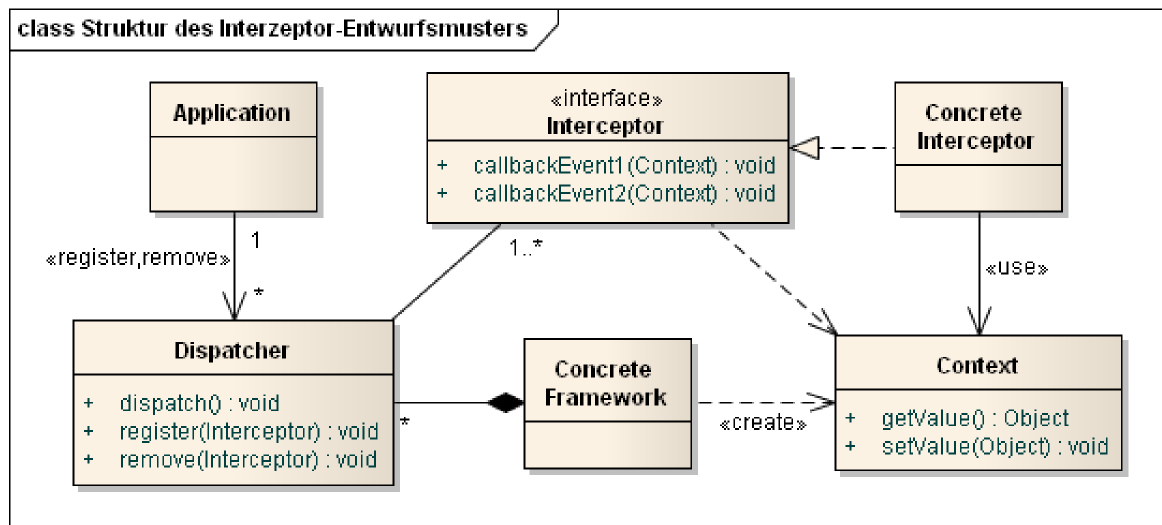

Abbildung 2.3.: Struktur des Interzeptor-Entwurfsmusters (vgl. Schmidt 2000, S. 115)

2.3.2. Vor- und Nachteile

Hier werden die wichtigsten Vor- und Nachteile des Interzeptor-Entwurfsmusters gegenübergestellt.

Vorteile

Flexibilität Neue Dienste können durch Standardschnittstellen hinzugefügt, geändert und entfernt werden, ohne die Architektur oder die Implementierung des Frameworks zu ändern.

Trennung der Belange Infrastruktur-Dienste und fachlicher Code sind voneinander getrennt und können von verschiedenen Entwicklern unabhängig voneinander implementiert werden.

Wiederverwendbarkeit Da der Interzeptor-Code vom Anwendungscode getrennt ist, können die Interzeptoren in verschiedenen Anwendungen wiederverwendet werden.

Nachteile

Komplexes Design Um das Framework möglichst flexibel zu konzipieren, müssen u.U. viele verschiedene Dispatcher angeboten werden. Dadurch kann das System komplex und ineffizient werden. Durch viele verschiedene Interzeptor-Typen wird der Code

schwerer verständlich. Andererseits führen generische Interzeptoren und Dispatcher zu umfangreichen und komplexen Schnittstellen.

Fehleranfälligkeit Fehlerhafte Interzeptoren können den Ablauf des gesamten Frameworks blockieren. Einige Frameworks beenden die Ausführung eines Interzeptors nach einem festgelegten Timeout, um dies zu verhindern. Ebenso können Interzeptoren Laufzeitfehler verursachen.

Hier seien nur die wichtigsten Vor- und Nachteile genannt, eine vollständige Übersicht ist in Schmidt (2000, S. 136 ff.) zu finden.

2.3.3. Realisierungen

Viele verbreitete J2EE-Anwendungsserver realisieren das Interzeptor-Entwurfsmuster, um Dienste zu EJBs hinzuzufügen. Es werden EJB-Container verwendet, welche die EJB-Komponenten von direkten Aufrufen und der Laufzeitumgebung isolieren. Der Container kann vor und nach EJB-Aufrufen vorkonfigurierte Interzeptoren aufrufen. So müssen Dienste, wie z. B. Transaktionssteuerung und Sicherheitsprüfungen, nicht von den Komponenten selbst implementiert werden (vgl. Schmidt 2000, S. 132 f.).

Der JBoss Anwendungsserver verwendet Interzeptoren auf Klienten- und Serverseite, um Dienste transparent zu EJBs hinzuzufügen. Dabei wird ein Methodenaufruf auf der Klientenseite in ein *Invocation*-Objekt umgewandelt, welches den Aufruf repräsentiert. Dieses Objekt wird den klientenseitigen Interzeptoren nacheinander übergeben, wodurch diese das Objekt auslesen und verändern können. Anschließend wird es über das Netzwerk an den Server übertragen und dort dem EJB-Container übergeben. Auch hier wird das Objekt nacheinander den konfigurierten Interzeptoren übergeben, bevor der letzte Interzeptor in der Kette die tatsächliche EJB-Methode aufruft (vgl. Fleury und Reverbel 2003, S. 361-363).

In der EJB-Spezifikation Version 3 wurden Interzeptoren für Java Beans standardisiert. Ein Interzeptor kann für alle oder einzelne ausgewählte Methoden einer Bean definiert werden. Die Konfiguration der Interzeptoren geschieht durch Annotationen oder im Deployment-Deskriptor. Eine Klasse oder Methode kann mit beliebig vielen Interzeptoren versehen werden. Der Kontext wird bei jedem Aufruf als *InvocationContext*-Objekt an die Methoden des Interzeptors übergeben.

Eine Interzeptor-Klasse wird als einfache Java-Klasse realisiert, wobei die Methoden eine festgelegte Signatur aufweisen müssen. Die Namen der Methoden sind beliebig. Die Verbindung zwischen Bean-Methoden und den zu verwendenden Interzeptor-Methoden geschieht allein durch Annotationen an den Bean-Methoden, der Bean-Klasse oder durch Konfiguration im Deployment-Deskriptor (vgl. Sun Microsystems 2006, S. 301-314).

2.4. AOP (Aspektorientierte Programmierung)

Üblicherweise wird die Gesamtfunktionalität eines Softwaresystems während des Entwurfs in funktionale Einheiten aufgeteilt. Dies können, je nach verwendeter Programmiersprache, Klassen, Funktionen oder Prozeduren sein. Verbreitete Programmiersprachen bieten entsprechende Abstraktions- und Kompositionsmechanismen, mit denen sich Systeme in Einheiten zerlegen und aus diesen zusammenfügen lassen (vgl. Kiczales u. a. 1997, S. 3 f.). Es ergeben sich Komponenten, die eine klar definierte und in sich abgeschlossene Funktionalität zur Verfügung stellen. Beispiele sind ein Bankkonto, eine Benutzerverwaltung oder ein Widget[1] einer grafischen Oberfläche.

Andererseits existieren häufig Anforderungen, die sich keiner bestimmten Komponente eindeutig zuordnen lassen und nicht unmittelbarer Bestandteil der fachlichen Funktionalität des Systems sind. Diese nennt man *Crosscutting Concerns* (deutsch: Querschnittsbelange), da sie über das gesamte System verteilt sind. Beispiele hierfür sind die Synchronisierung nebenläufiger Zugriffe, Logging und Fehlerbehandlung (vgl. Kiczales u. a. 1997, S. 8 f.).

2.4.1. Grundbegriffe

Aspekte und Advices

Aspekte fassen die Querschnittsbelange des Systems zusammen. Sie kapseln Funktionalität, die sonst innerhalb mehrerer verschiedener Komponenten des Systems implementiert werden müsste. So könnte z. B. das Logging durch einen Aspekt realisiert werden. Eine klare Trennung zwischen Komponenten- und Aspekt-Code ist eines der Hauptziele der aspektorientierten Programmierung (vgl. Kiczales u. a. 1997, S. 8).

Bei der Verwendung von AOP in Java werden Aspekte zumeist als einfache Java-Klassen modelliert. Die Methoden dieser Klassen werden *Advices* genannt und enthalten den tatsächlich ausgeführten Programmcode. Eine Aspektklasse kann dabei mehrere Advices enthalten, die beim Erreichen bestimmter Stellen im Programm, genannt *Joinpoints*, ausgeführt werden (vgl. Böhm 2006, S. 25). Aspekte können auch Instanzvariablen enthalten.

Joinpoints und Pointcuts

Joinpoints sind wohldefinierte Stellen im Programm, an denen Aspektcode ausgeführt werden kann. Sie sind meist implizit durch die Programmiersprache gegeben (vgl. Kiczales u. a.

[1] Ein Widget ist ein Steuerelement einer grafischen Oberfläche, wie z. B. ein Button oder ein Textfeld.

1997, S. 17). In Java entsprechen z. B. der Aufruf von Methoden oder der Zugriff auf Felder solchen Joinpoints (vgl. Böhm 2006, S. 25).

Ein *Pointcut* definiert, an welchen Joinpoints eine Advice-Methode eines Aspekts ausgeführt wird. Er selektiert also gezielt Joinpoints aus der Menge aller Joinpoints und verbindet diese mit einer Advice-Methode (vgl. Böhm 2006, S. 25 f.). Beispiele für solche Pointcuts sind die Ausführung einer Methode mit einem festgelegten Namen einer festgelegten Klasse oder Aufrufe von Konstruktoren in einem bestimmten Paket.

2.4.2. Technische Umsetzung

Da Aspekte und fachliche Komponenten unabhängig voneinander implementiert werden und anschließend getrennt voneinander vorliegen, ist ein weiterer Schritt erforderlich, um beide zu einem Programm zusammenzufügen. Dies geschieht durch das sogenannte *Weaving* (deutsch: Weben). Dabei wird der Aspektcode in den Anwendungscode eingefügt. Das Gesamtergebnis entspricht in etwa dem, als wären die Querschnittsbelange im gesamten System verteilt implementiert worden. Im Fall von Java erzeugt der Weaver als Ergebnis den aus den Aspekten und fachlichen Klassen zusammengefügten Bytecode. Dieser Vorgang wird auch *Bytecode-Instrumentierung* genannt. Ein einfaches Beispiel ist in Abbildung 2.4 dargestellt.

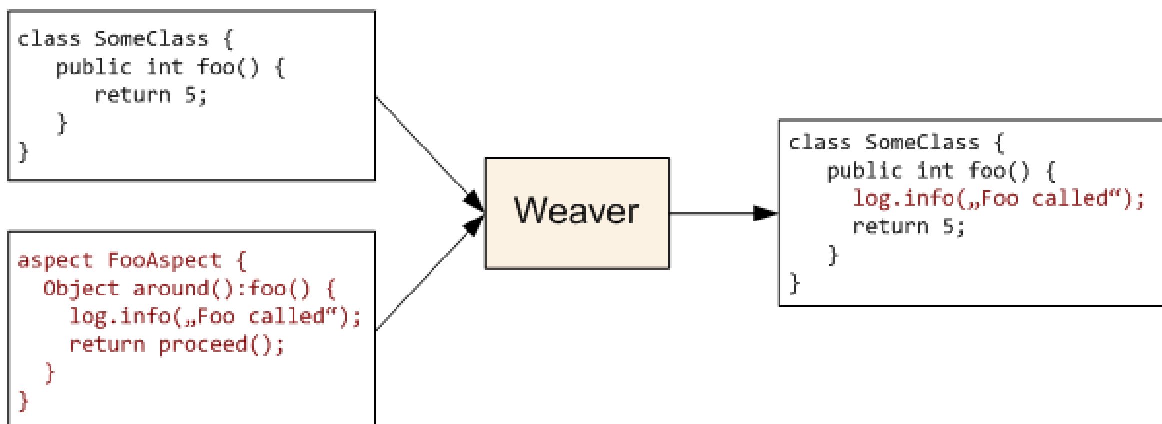

Abbildung 2.4.: Funktionsweise eines Aspekt-Weavers

Es gibt verschiedene Zeitpunkte, zu denen das Weaving durchgeführt werden kann (Böhm 2006; Kersten 2005b):

Compile-Time-Weaving Aspekt- und Anwendungs-Code werden durch einen speziellen Compiler zusammengefügt. Das Ergebnis sind Klassen, in denen die Aspekte bereits enthalten sind. Diese Möglichkeit beansprucht mehr Zeit beim Kompilieren eines

Systems, ist jedoch zur Laufzeit äußerst performant, da keine zusätzlichen Objekte benötigt werden.

Load-Time-Weaving Die Aspekte werden beim Laden der Klassen in die JVM (Java Virtual Machine) in die Klassen eingefügt. Dies verlangsamt den Start des Systems, dafür ist danach kein Unterschied zum Compile-Time-Weaving feststellbar. Bei der Ausführung muss der JVM eine spezielle Bibliothek (genannt *Agent*) als Parameter übergeben werden, welche die Bytecode-Instrumentierung vornimmt.

Runtime-Weaving Hier werden die Aspekte zur Laufzeit erst bei Bedarf mit den Klassen verbunden. Meist werden dazu dynamische Proxies verwendet. Sie werden im folgenden Abschnitt genauer erklärt. Diese Option wirkt sich am negativsten auf die Performanz des Systems zur Laufzeit aus.

Dynamische Proxies in Java

In Java können seit Version 1.3 Proxyobjekte für bestehende Schnittstellen zur Laufzeit erzeugt werden. Diese werden *dynamische Proxies* genannt. Ein Proxy ist ein Stellvertreter-Objekt, welches die gleichen Methoden wie das tatsächliche Objekt implementiert und an dessen Stelle verwendet wird. Dies ermöglicht die Ausführung zusätzlicher Logik vor und nach den Aufrufen der tatsächlichen Methoden (vgl. Gamma 2008, S. 254 ff.).

Bei der Verwendung von Proxies ist die Performanzeinbuße größer als bei Klassen, die vorher durch Bytecode-Instrumentierung angepasst wurden. Der Grund dafür ist, dass ein zusätzliches Objekt erzeugt werden muss, welches das tatsächliche Objekt per Reflection aufruft. Der Vorteil besteht darin, dass kein zusätzlicher Weaver oder Compiler notwendig ist.

Weitere Informationen zu dynamischen Proxies sind in (vgl. Oracle 1999) zu finden.

2.4.3. Vor- und Nachteile

Hier werden die wichtigsten Vor- und Nachteile der aspektorientierten Programmierung gegenübergestellt.

Vorteile

- Durch Modularisierung der Querschnittsbelange zu Aspekten ist der Quellcode besser strukturiert und leichter nachvollziehbar. Verantwortlichkeiten der Komponenten sind klar getrennt und Redundanz wird verringert.

- Designentscheidungen können in späteren Phasen der Entwicklung getroffen werden. Es ist nachträglich möglich, Querschnittsbelange im System als Aspekte zu implementieren.
- Die Weiterentwicklung des Systems wird vereinfacht. Bestehende Aspekte, z. B. ein Logging-Aspekt, können für neu entwickelte fachliche Komponenten unverändert angewandt werden. Umgekehrt können neue Aspekte für alle bestehenden Komponenten hinzugefügt werden.

(vgl. Böhm 2006, S. 23)

Nachteile

- Der Einsatz zu vieler Aspekte macht das System schwer verständlich. Der Kontrollfluss ist an einer Stelle im Quellcode nicht mehr unmittelbar erkennbar. Falls Wechselwirkungen zwischen verschiedenen Aspekten existieren, wird dieser Effekt noch verstärkt.
- Die Kapselung von Komponenten kann durchbrochen werden. Es können neue Felder und Methoden in Klassen eingefügt werden. Dadurch sind unsaubere Programmierpraktiken möglich, die dringend vermieden werden sollten.

(vgl. Böhm 2006, S. 361 f.)

2.4.4. Realisierungen

Es existieren für Java mehrere AOP-Frameworks. Die wichtigsten sind:

AspectJ Dies ist das älteste AOP-Framework. Es ermöglicht die Realisierung von Aspekten über eine Spracherweiterung oder als einfache Java-Klassen, die mit Annotationen versehen werden. Das Framework enthält einen speziellen Compiler und eine Bibliothek zur Anwendung von Load-Time-Weaving. AspectJ verwendet Bytecode-Instrumentierung (vgl. Xerox Corporation 2003–2005).

JBoss AOP Dieses Framework wurde im Umfeld des JBoss Anwendungsservers entwickelt. Laut Fleury u. a. (2006, S. 581 ff.) ist es jedoch auch unabhängig von dem Anwendungsserver in beliebigen Java-Anwendungen einsetzbar. Es unterstützt Runtime-Weaving. Die Aspekte werden als einfache Java-Klassen realisiert und in einer XML-Datei konfiguriert (vgl. Fleury u. a. 2006, S. 581 ff.).

Spring AOP Dieses Framework verwendet dynamische Proxies und ist für die Verwendung zusammen mit dem Spring Dependency-Injection-Container entwickelt worden. Spring erlaubt zusätzlich die Einbindung von bereits vorhandenen AspectJ-Aspekten (vgl. Johnson u. a. 2008).

2.5. GWT (Google Web Toolkit)

Mithilfe des Google Web Toolkits können dynamische Webanwendungen in Java entwickelt werden. Diese machen starken Gebrauch von Ajax (Asynchronous JavaScript and XML). Ajax bietet die Möglichkeit, asynchron Aufrufe an einen Webserver zu senden. Als Austauschformat für Nachrichten dient hier meist XML, jedoch sind auch andere Formate möglich. Ajax bietet den Vorteil, dass Webseiten nicht vollständig neu geladen werden müssen, sondern auch einzelne Teile einer Seite aktualisiert werden können. Dadurch ist die Entwicklung dynamischer Webanwendungen möglich. Da Ajax technisch auf JavaScript basiert, ist für die Ausführung von Ajax-basierten Webanwendungen keine zusätzliche Webbrowser-Erweiterung notwendig.

In GWT wird die komplette Anwendung, d. h. klientenseitiger und serverseitiger Code zunächst in Java realisiert. Der klientenseitige Code wird anschließend von einem Java-zu-JavaScript-Compiler in JavaScript übersetzt. Am resultierenden JavaScript-Code ist nicht erkennbar, dass die Anwendung mithilfe von GWT entwickelt wurde. Da der Entwickler die Weboberfläche mithilfe von Java entwickelt, muss er nicht mit den Besonderheiten einzelner Webbrowser oder JavaScript vertraut sein. Der GWT-Compiler erzeugt den spezifischen JavaScript-Code für verschiedene Browser aus den Java-Klassen. Die Produktivität des Entwicklers kann dadurch erhöht werden (vgl. Google 2011a).

Mit Sencha Ext GWT existiert eine Bibliothek, die eine große Anzahl von zusätzlichen Widgets, Layouts und ähnlichen Erweiterungen für GWT enthält. Diese erleichtert die Entwicklung zusätzlich, da viele häufig auftretende Anforderungen hier bereits umgesetzt wurden (vgl. Sencha 2011).

3. Analyse

In diesem Kapitel wird die Problemstellung analysiert. Dafür werden zunächst alle Anforderungen an die Monitoring-Komponente erläutert und Designziele definiert, die während der Konzeption berücksichtigt werden. Zuletzt werden mehrere vorhandene Monitoring-Lösungen vorgestellt und bewertet.

3.1. Anforderungen

In diesem Abschnitt werden alle Anforderungen (A) und Prämissen (P) an die zu entwickelnde Monitoring-Komponente erfasst. Sie wird im Folgenden nur *Monitoring*, das HyperTest-Gesamtsystem nur *System* genannt.

3.1.1. Funktionale Anforderungen

Prozessüberwachung

A1. Das Monitoring überwacht die Verfügbarkeit der zum System gehörenden verteilten Prozesse. Dazu gehören die Datenbank, der Datenimporter und die Komponente für serverbasierte Auswertungen. Eventuell vorhandene kundenspezifische Prozesse können ebenfalls überwacht werden.

A2. Der Benutzer kann die zu überwachenden Prozesse zusammen mit einem Anzeigenamen konfigurieren.

A3. Der Benutzer kann eine Zykluszeit für jeden überwachten Prozess konfigurieren. Diese gibt an, wie häufig das Monitoring den Zustand des Prozesses überprüft.

A4. Der Benutzer kann die Überwachung der verteilten Prozesse zur Laufzeit zu- und abschalten.

P1. Das Monitoring hat keinen direkten Zugriff auf verteilte Prozesse, wie die Datenbank, den Datenimporter und die Komponente für serverbasierte Auswertungen. Die Kommunikation mit diesen Prozessen erfolgt ausschließlich über den HyperTest-Server.

Statistikerhebung

A5. Das Monitoring erhebt statistische Daten innerhalb des HyperTest-Servers. Alle Daten werden einer Kategorie (z. B. Benutzerstatistik) zugeordnet. Jede Kategorie besitzt einen Namen, der sie systemweit eindeutig identifiziert.

A6. Die Daten für eine Kategorie werden periodisch erhoben und gespeichert.

A7. Das Monitoring legt die erhobenen Daten in der HyperTest-Datenbank ab.

A8. Der Benutzer kann das Zeitintervall für die Erhebung von Daten für jede Kategorie einzeln konfigurieren.

A9. Der Benutzer kann die Erhebung der statistischen Daten für jede Kategorie zur Laufzeit einzeln zu- und abschalten.

P2. Standardmäßig vorhandene Kategorien sind Hauptspeicher-, Benutzer- und Methodenaufrufstatistik.

A10. Das Monitoring ermittelt im Rahmen der Hauptspeicherstatistik den belegten und insgesamt verfügbaren Hauptspeicher des HyperTest-Servers.

A11. Das Monitoring ermittelt im Rahmen der Benutzerstatistik die Anzahl der am HyperTest-Server angemeldeten Benutzer.

A12. Das Monitoring erhebt im Rahmen der Methodenaufrufstatistik Daten über vorher vom Benutzer festgelegte Methoden innerhalb des HyperTest-Servers. Das sind:

- Die minimale, maximale und durchschnittliche Ausführungsdauer einer Methode in einem festgelegten Zeitintervall.
- Die Anzahl der Aufrufe einer Methode in einem festgelegten Zeitintervall.

A13. Der Benutzer kann die Erhebung von Methodenaufrufstatistiken durch Filter über die Parameter und den Rückgabewert auf Aufrufe einschränken, die bestimmten Bedingungen genügen.

A14. Filter können atomar sein oder durch logische Operatoren (Konjunktion, Disjunktion und Negation) aus atomaren Filtern zusammengesetzt werden.

P3. Der Benutzer gibt Filter zunächst textuell in einem genau festgelegten Format an.

Grafische Oberfläche

A15. Eine grafische Oberfläche visualisiert den aktuellen Zustand der Prozesse und die erhobenen statistischen Daten.

A16. Die grafische Oberfläche ermöglicht die Konfiguration des Monitorings. Dazu gehören die zu überwachenden Komponenten (mit Anzeigename und Zykluszeit), sowie die zu erhebenden Kategorien (mit Zykluszeit) und Methoden (mit Methodenname, Klassenname und Filter) der Statistikerhebung.

A17. Die grafische Oberfläche ermöglicht bei der Anzeige von statistischen Daten eine Eingrenzung auf ein Zeitintervall. Für die Anzeige von Methodenaufrufstatistiken kann außerdem eine Methode gezielt ausgewählt werden.

3.1.2. Nichtfunktionale Anforderungen

A18. Das Monitoring kann transparent zum bestehenden System hinzugefügt werden, d. h. es sind keine Änderungen an der bestehenden Anwendungslogik notwendig.

A19. Die Statistikerhebung verlangsamt die Ausführung von Operationen im Server so wenig wie möglich.

A20. Anwendungsentwickler können die Erhebung statistischer Daten für zusätzliche Kategorien hinzufügen, ohne Änderungen am bestehenden System vornehmen zu müssen.

3.1.3. Technische Rahmenbedingungen

A21. Das System verwendet die HyperTest-Plattform Version 2.10.6.

A22. Das System basiert auf der Java Enterprise Edition Version 1.4 und Enterprise Java Beans Version 2.1.

A23. Das System läuft auf einem JBoss Application Server Version 4.2.3 oder höher.

A24. Das System verwendet eine Oracle-Datenbank Version 9i oder höher oder eine PostgreSQL-Datenbank Version 8.4 oder höher.

3.2. Designziele

In diesem Abschnitt wird erläutert, welche Ziele bei der Entwicklung einer Lösung zusätzlich zu den Anforderungen verfolgt werden. Sie werden später zur Bewertung von verschiedenen Lösungsmöglichkeiten herangezogen.

Portabilität
Zwar wird die Lösung anhand des aktuellen HyperTest-Systems mit dem JBoss Anwendungsserver als Middleware entwickelt, jedoch bestehen Überlegungen, das System auf andere Anwendungsserver zu portieren. Um dies zu vereinfachen, sollen die Abhängigkeiten zum verwendeten Anwendungsserver minimiert werden. Es werden, soweit möglich, nur in der J2EE-Spezifikation standardisierte Dienste verwendet. Diese müssen von jedem J2EE-konformen Anwendungsserver realisiert werden.

Einfachheit
Spezialfälle und viele unterschiedliche Strukturen machen das System schwer verständlich. Daher sollen die Lösungen für einzelne Aspekte des Monitorings möglichst gleichförmig gestaltet und Spezialfälle vermieden werden. Im Falle der Prozessüberwachung sollten alle Prozesse auf die gleiche Art und Weise überwacht werden können, idealerweise auch die Datenbank. Für die Statistikerhebung sollte nur ein allgemeiner Mechanismus zur Erhebung von Daten entwickelt werden, der für unterschiedliche Kategorien verwendet werden kann.

Erweiterbarkeit
Um das Monitoring auf neu hinzukommende Anforderungen vorzubereiten, sollte es so gestaltet werden, dass es einfach erweitert werden kann. Für die Prozessüberwachung bedeutet dies, dass zukünftig entwickelte Prozesse einfach in die Überwachung aufgenommen werden können. Im Falle der Statistikerhebung sollten neue Kategorien einfach hinzugefügt werden können, ohne die bestehende Logik ändern zu müssen. Auch die grafische Oberfläche sollte nicht auf die vorhandenen Kategorien festgelegt sein, sondern sich einfach um weitere Ansichten erweitern lassen.

Zentrale Konfiguration
Sind die Konfigurationsparameter über verschiedene Prozesse verteilt, so führt dies zu einem erhöhten Konfigurationsaufwand und zu hoher Fehleranfälligkeit. Eine Inkonsistenz der Konfigurationen in verschiedenen Komponenten des Systems kann Fehler verursachen. Die Konfiguration ist aufwändig, da die Orte, an denen die Parameter gespeichert sind, erst gesucht werden müssen. Daher sollte die Konfiguration möglichst an einer zentralen Stelle abgelegt werden, um so einen *Single Point of Administration* zu schaffen.

3.3. Vorhandene Lösungen & Produkte

In diesem Abschnitt werden einige vorhandene Monitoring-Lösungen vorgestellt und ihre Funktionalität beschrieben. Am Ende wird eine kurze Zusammenfassung und Bewertung der vorgestellten Werkzeuge gegeben.

3.3.1. Glassbox

Glassbox ist ein Monitoring-Werkzeug, welches die Erkennung von Problemen in Java-Anwendungen vereinfachen soll. Es eignet sich besonders zur Erkennung von Performanz-problemen und aufgetretenen Fehlern innerhalb von Anwendungsservern.

Das Werkzeug wurde hauptsächlich mittels AOP und JMX realisiert. Performanzstatistiken über Servlet-Aufrufe und Datenbankzugriffe werden mithilfe von Aspekten erhoben. Durch die Verwendung von Load-Time-Weaving kann Glassbox zu einer bestehenden Anwendung hinzugefügt werden, ohne Änderungen am bestehenden Code zu erfordern. Die gewonne-nen Daten werden im Hauptspeicher in MBeans verwaltet. Auf diese Weise können erfasste Werte mithilfe von bestehenden Anwendungen, wie der JConsole[1], eingesehen werden. Ein-zelne Aspekte können außerdem über JMX deaktiviert werden.

Eine Komponente namens Glassbox Troubleshooter vergleicht die gewonnenen Daten mit einer Wissensbasis, die Beschreibungen für bekannte Probleme enthält. Der Benutzer kann zusätzlich konfigurieren, wie viel Zeit eine Operation maximal beanspruchen darf, bevor sie als zu langsam eingestuft wird. Auf diese Weise werden z. B. das lange Warten eines Threads auf die Freigabe eines Locks oder ein langsamer Datenbankzugriff erkannt. Die so ermittelten potentiellen Probleme werden auf einer webbasierten grafischen Oberfläche dar-gestellt. Neben einer natürlichsprachlichen Beschreibung des erkannten Problems werden relevante Informationen, wie z. B. ein Aufruf-Stack oder eine problematische SQL-Abfrage (Structured Query Language) angezeigt. Die grafische Oberfläche ist in Abbildung 3.1 zu sehen.

Weitere Informationen zum Projekt sind unter Glassbox Corporation (2008) zu finden. Unter Bodkin (2005) gibt einer der Entwickler einige Einblicke in die Realisierung von Glassbox.

[1] Die JConsole ist ein grafisches Werkzeug zur Anzeige von MBeans. Es ist Bestandteil jedes JDKs (Java Development Kit).

Abbildung 3.1.: Die grafische Oberfläche von Glassbox (Glassbox Corporation 2008)

3.3.2. JBoss StatisticsCollector

In früheren JBoss-Versionen gab es einen Dienst zur Erhebung von Statistiken über Methodenaufrufe und Datenbankabfragen, den JBoss StatisticsCollector. Dieser wurde mit JBoss Version 3.2 in ein von der JBoss-Community gepflegtes Zusatzpaket ausgelagert und ist nicht mehr in der normalen JBoss-Lieferung enthalten. Der StatisticsCollector erfasst statistische Daten über Methodenaufrufe und SQL-Abfragen. Es werden alle EJB-Aufrufe, auch lokale Aufrufe an andere EJBs, und SQL-Abfragen erfasst. Überdies wird aufgezeichnet, wie häufig die gleiche Transaktion durchgeführt wird. Die Erhebung von Werten kann über JMX gestartet werden. Ermittelte Daten werden im Hauptspeicher durch MBeans verwaltet.

Das Ergebnis der Erhebung sind *Reports*, auf die ebenfalls über JMX zugegriffen werden kann. Standardmäßig können zwei Arten von Reports generiert werden: Ein Table-Report mit detaillierten Informationen über ausgeführte SQL-Abfragen oder ein allgemeiner Report mit Informationen über ausgeführte Methodenaufrufe und SQL-Abfragen. Es können bei Bedarf weitere Reports realisiert werden.

Der JBoss StatisticsCollector verwendet Interzeptoren zur Erfassung der Daten. Diese wer-

den in der EJB-Container-Konfiguration des Anwendungsservers hinzugefügt. Auf diese Weise kann der StatisticsCollector transparent zu einer bestehenden Anwendung hinzugefügt werden, ohne den bestehenden Anwendungscode verändern zu müssen. Die mitgelieferten Interzeptoren sind jedoch JBoss-spezifisch und können in keinem anderen Anwendungsserver genutzt werden.

Das Projekt scheint nicht mehr gepflegt zu werden. Sowohl die notwendigen Dateien als auch die Dokumentation sind im Internet nur schwer aufzufinden. Die Projektseite innerhalb der JBoss-Community-Webseite wurde zuletzt im Juli 2008 aktualisiert. Einige Foreneinträge innerhalb der JBoss-Community lassen die Vermutung zu, dass es häufig Probleme gibt, den StatisticsCollector in Betrieb zu nehmen[2].

Weitere Informationen sind unter JBoss Community (2008) zu finden.

3.3.3. RHQ / JBossON

RHQ ist ein Open-Source Management- und Monitoring-Werkzeug, welches ursprünglich von RedHat entwickelt wurde. Es wurde mit einem vorher ebenfalls eigenständigen Monitoring-Werkzeug namens Jopr zusammengefasst. Neben RHQ existiert noch eine kommerzielle Version des Werkzeugs mit dem Namen JBossON (JBoss Operations Network).

Die Funktionsweise ist grundlegend anders als bei den beiden bisher vorgestellten Monitoring-Lösungen. RHQ ist selbst in Server und Klienten, genannt *Agents*, aufgeteilt. Der RHQ-Server benötigt einen zusätzlichen eigenständigen JBoss-Server. Auf jedem physikalischen Computer, auf dem zu verwaltende Ressourcen laufen, muss ein Agent installiert werden. Dieser sendet zyklisch Informationen an den RHQ-Server. Welcher Art diese Informationen sind, bestimmen installierte Plugins. Auf diesem Wege kann das System erweitert werden.

Das Werkzeug bietet eine umfangreiche Funktionalität durch vorhandene Plugins: So existieren bereits Plugins zur Ermittlung von allgemeinen Informationen, wie z. B. Speicherbedarf, Prozessorauslastung und Festplattenbelegung. Weiterhin gibt es Plugins zur Verwaltung vieler verschiedener Server, darunter der JBoss Anwendungsserver, Apache HTTP-Server und einige Datenbankserver. Für Anwendungs- und Webserver können die Antwortzeiten von Servlets und EJBs ermittelt werden. Beim Auftreten bestimmter Ereignisse kann RHQ Aktionen, wie den Versand einer E-Mail, veranlassen. Ein solches Ereignis kann z. B. das Erreichen eines Schwellenwertes des benutzten Hauptspeichers oder eine Verfügbarkeitsänderung einer verwalteten Ressource sein.

[2]Vgl. http://http://community.jboss.org/message/165415?tstart=0. Zugriffsdatum: 19.07.2011

RHQ enthält eine GWT-basierte grafische Oberfläche. Hier können alle notwendigen Einstellungen vorgenommen und die ermittelten Daten zur Laufzeit eingesehen werden. Die Oberfläche ist in Abbildung 3.2 zu sehen. Neben der grafischen Oberfläche wird auch ein Kommandozeilenklient zur Verfügung gestellt.

Abbildung 3.2.: Die grafische Oberfläche von RHQ

Weitere Informationen über RHQ sind unter RHQ Project (2011) zu finden.

3.3.4. Zusammenfassung

Abschließend wird hier nun bewertet, welche Konzepte der vorgestellten Lösungen sich zur Erfüllung der erhobenen Anforderungen eignen und in die Konzeption einfließen werden.

Glassbox zeigt einen möglichen Weg, um Methodenaufrufstatistiken durch den Einsatz von aspektorientierter Programmierung zu erheben. Allerdings bietet es keine Möglichkeit Statistiken zu erheben, die unabhängig von Klientenaufrufen sind. Da die Möglichkeiten zur

Konfiguration des Werkzeuges recht eingeschränkt sind, ist die Anforderung, einzelne Operationen gezielt erfassen zu können, ebenfalls nicht realisierbar. Dennoch bietet der Ansatz über aspektorientierte Programmierung einige Vorteile: Die Lösung funktioniert auf beliebigen Anwendungsservern, da diese keine Kenntnis davon besitzen, dass AOP verwendet wird. Zur Verwaltung von erhobenen Daten werden MBeans eingesetzt, wodurch diese über die standardisierte JMX-Schnittstelle abgefragt werden können. Eine Möglichkeit zur Überwachung von Prozessen bietet Glassbox nicht.

Der JBoss StatisticsCollector verfolgt einen ähnlich Ansatz wie Glassbox, allerdings unter Verwendung von Interzeptoren. Für die Verwaltung von Daten werden ebenfalls MBeans verwendet. Die Verwendung von Interzeptoren hat den Nachteil, dass diese vor EJB Version 3 nicht standardisiert und damit abhängig vom jeweiligen Anwendungsserver sind. Der StatisticsCollector ist auf die Erhebung von Methodenaufrufstatistiken beschränkt.

RHQ unterscheidet sich in der Arbeitsweise stark von den vorher vorgestellten Werkzeugen, da es eigene Prozesse zur Verwaltung von Ressourcen verwendet. Es ist das einzige Werkzeug, welches eine Prozessüberwachung ermöglicht. Auch die Erhebung von Statistiken ist grundsätzlich möglich. Durch seinen Aufbau ist das Werkzeug zwar flexibel erweiterbar, allerdings für HyperTest nicht nutzbar. Bei einigen Kunden existieren strenge Regelungen für die Freigabe von verwendbaren Netzwerkports. Daher ist in den Anforderungen festgelegt, dass die Monitoring-Komponente ausschließlich über den HyperTest-Server mit den verteilten Prozessen kommunizieren darf.

Ein von allen Lösungen verwendeter Ansatz besteht darin, die erhobenen statistischen Daten und die Konfiguration mithilfe von MBeans zu verwalten. Dies bietet den Vorteil, dass die Daten über die standardisierte JMX-Schnittstelle abgefragt werden können. Außerdem ist dadurch eine Konfiguration der Komponenten zur Laufzeit möglich. Zur Erhebung von Methodenaufrufstatistiken können grundsätzlich Interzeptoren oder AOP verwendet werden. Welche Möglichkeit sich hier als vorteilhafter erweist, wird im späteren Verlauf dieser Arbeit diskutiert.

4. Konzeption

Dieses Kapitel bildet den eigentlichen Kern dieser Arbeit. Hier werden verschiedene Lösungsansätze für das Monitoring entwickelt und gemäß den Anforderungen und Designzielen aus Kapitel 3 bewertet. Auf diese Weise wird ein geeigneter Lösungsweg ausgewählt und weiterentwickelt.

4.1. Architektur der Komponente

In diesem Abschnitt werden die grobe Architektur der Monitoring-Komponente und ihre mögliche Verteilung auf mehrere physikalische Computer erläutert.

4.1.1. Komponenten & Schnittstellen

Ziel dieser Arbeit ist die Entwicklung einer Monitoring-Komponente, die sowohl eine Prozessüberwachung als auch eine Statistikerhebung leistet. Dies spiegelt sich im Komponentenschnitt dadurch wider, dass die Komponente in zwei Subkomponenten aufgeteilt wird:

Watchdog Diese Subkomponente stellt die Funktionalität zur Überwachung der verteilten Prozesse bereit. Sie bietet Schnittstellen an, um die Überwachung zu konfigurieren und den aktuellen Systemzustand abzufragen. Außerdem existiert eine weitere Schnittstelle, die von den überwachten Prozessen genutzt wird. Die überwachten Prozesse werden im Folgenden nur *Subjekte*, die serverseitigen Teile der Subkomponente nur *Watchdog* genannt.

Statistics Diese Subkomponente stellt die Funktionalität zur Erhebung von Statistiken bereit. Sie bietet Schnittstellen an, um die Statistikerhebung zu konfigurieren und erhobene Daten abzufragen. Außerdem werden weitere Schnittstellen bereitgestellt, über die Daten zur Statistiksammlung hinzugefügt werden können. Es wird eine allgemeine Schnittstelle zur Erfassung von statistischen Daten und eine spezielle Schnittstelle zur Erfassung von Methodenaufrufstatistiken angeboten.

Der Aufbau der Komponente ist in Abbildung 4.1 dargestellt. Ein Klient ist eine beliebige Anwendung, welche die Monitoring-Komponente verwendet. Die im Rahmen dieser Arbeit entwickelte grafische Oberfläche zur Konfiguration des Monitorings und zur Abfrage der erhobenen Daten ist ein solcher Klient. In der Abbildung ist außerdem sichtbar, dass die gesamte Monitoring-Komponente die HyperTest-Plattform benutzt. Dabei handelt es sich um eine Programmbibliothek mit Klassen und Schnittstellen, die innerhalb eines HyperTest-Systems häufig verwendet werden. Fast alle Komponenten verwenden diese als Grundlage.

Die Schnittstelle *StatisticsCollection* dient zum Einfügen von Daten in die Statistiksammlung. Sie wird von der Komponente angeboten, damit in Zukunft neben der Benutzer-, Speicher- und Methodenaufrufstatistik weitere Daten zur Statistiksammlung hinzugefügt werden können. Die Schnittstelle *MethodStatisticsCollection* dient zur Erfassung von Methodenaufrufen. Sie enthält Methoden, mit denen Methodenaufrufe und ihre Ausführungszeiten erfasst werden können.

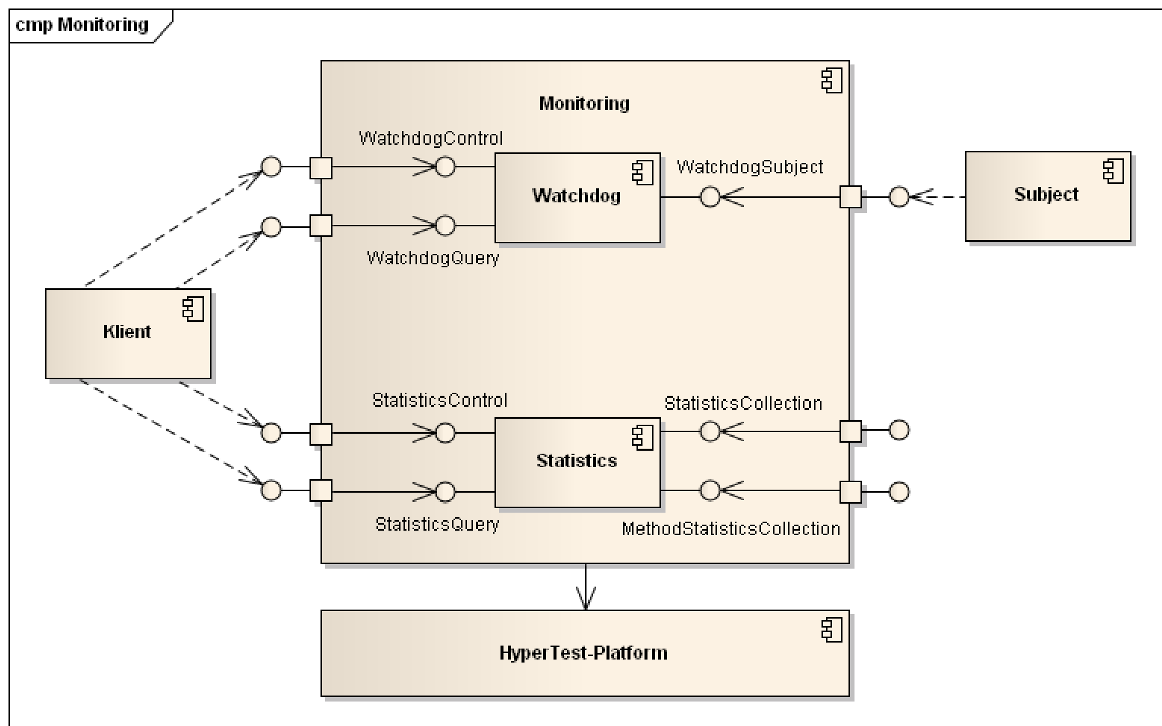

Abbildung 4.1.: Struktur der Monitoring-Komponente

Es ist erkennbar, dass die Watchdog- und Statistics-Subkomponenten völlig unabhängig voneinander verwendbar sind. Benötigt ein Klient lediglich die Prozessüberwachung, so verwendet er nur die Schnittstellen der Watchdog-Subkomponente, unabhängig von der Statistikerhebung.

4.1.2. Verteilungssicht

Da es sich bei HyperTest um ein verteiltes System handelt, sind Komponenten über verschiedene physikalische Computer verteilt. Das gilt auch für die Monitoring-Komponente. Um die benötigten Teile der Komponente optimal verteilen zu können, ergibt sich folgende Zerlegung in Programmbibliotheken:

Watchdog- und Statistics-Client
: Diese Bibliotheken enthalten die klientenseitigen Bestandteile der beiden Subkomponenten. Das sind jeweils eine Konfigurations- und eine Abfrage-Schnittstelle. Außerdem existiert jeweils eine Klasse, die diese Schnittstellen implementiert und die Kommunikation mit dem Server kapselt.

Watchdog- und Statistics-Server
: Diese Bibliotheken enthalten die serverseitigen Bestandteile der beiden Subkomponenten. Für die Prozessüberwachung ist dies die gesamte Logik zur zyklischen Überprüfung der Verfügbarkeit der Prozesse, für die Statistikerhebung die Logik zur Erhebung und Speicherung von statistischen Daten.

Watchdog-Subject
: In den überwachten Subjekten ist zusätzliche Logik notwendig, die eine Überwachung der jeweiligen Prozesse durch den Watchdog ermöglicht. Diese ist in der Subjektbibliothek enthalten.

Statistics-Interceptor
: Da für die Statistik transparent Methodenaufrufstatistiken erhoben werden sollen, muss in den Aufrufmechanismus eingegriffen werden. Die erforderliche Logik in Form eines Interzeptors oder Aspekts ist in dieser Bibliothek enthalten.

Monitoring-Common
: Enthält allgemein genutzte Schnittstellen und Klassen der Monitoring-Komponente, wie z. B. Transportobjekte für die Übertragung von Daten zwischen Server und Klienten.

Monitoring-WebGUI
: Enthält alle benötigten Klassen und Schnittstellen der grafischen Oberfläche. Dieser Programmteil gehört nicht unmittelbar zur Monitoring-Komponente, sondern ist ein Klient, der die Komponente verwendet. Dazu verwendet die WebGUI die o. g. Klientenbibliotheken.

Durch diese feingranulare Aufteilung können die Bestandteile der Komponente gezielt dort eingesetzt werden, wo sie benötigt werden. Eine mögliche Verteilung der Artefakte auf mehrere physikalische Computer ist in Abbildung 4.2 dargestellt.

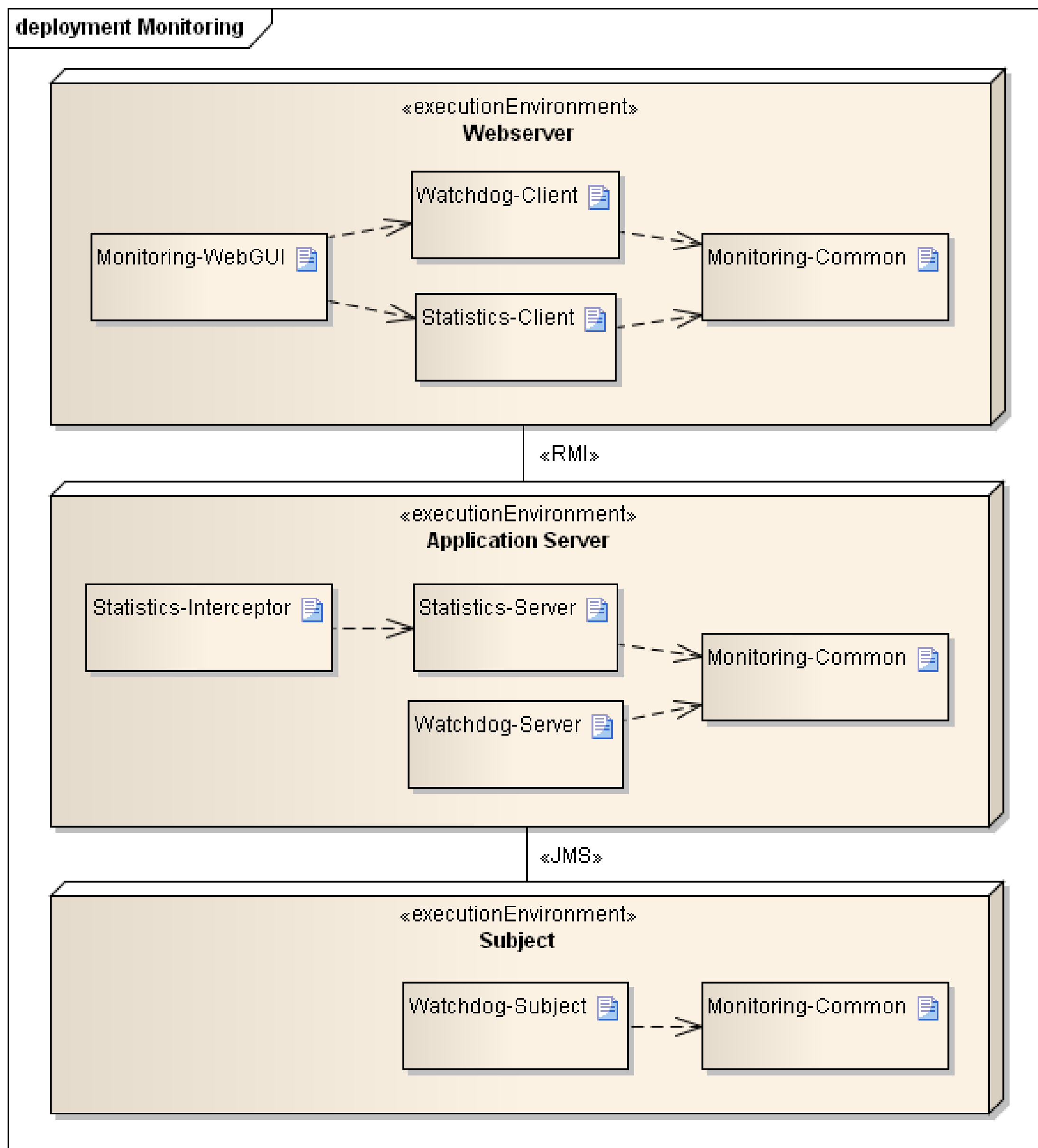

Abbildung 4.2.: Verteilung der Artefakte auf physikalische Computer

4.2. Prozessüberwachung

Im Folgenden wird die Watchdog-Subkomponente entworfen. Dafür werden verschiedene Lösungsmöglichkeiten entwickelt und bewertet. Am Ende dieses Abschnitts wird der Feinentwurf der Subkomponente vorgestellt.

4.2.1. Lösungsansätze

Zyklische Meldung von überwachten Subjekten

Um Portabilität zu erreichen, sollen standardisierte Dienste verwendet werden, soweit dies möglich ist. Dieser Lösungsansatz verwendet JMS zur Kommunikation des Watchdogs mit den Subjekten. JMS ermöglicht lose Kopplung zwischen Komponenten, indem diese sich mit einem Nachrichtenkanal verbinden, anstatt direkt miteinander zu kommunizieren. Dadurch wird die Abhängigkeit zwischen den Komponenten minimiert. Es scheint sich auf den ersten Blick hervorragend für die Entwicklung einer Prozessüberwachung zu eignen.

Im Server wird eine Watchdog-Komponente erstellt. Diese meldet sich bei einer für das Monitoring erstellten JMS-Queue als Empfänger an. Alle Subjekte melden sich bei dieser Queue als Sender an. Sie senden periodisch eine Nachricht, um dem Watchdog mitzuteilen, dass sie aktiv sind. Dieser empfängt alle Nachrichten und aktualisiert so seine Wissensbasis. Sendet ein Prozess über eine vorher festgelegte Zeitspanne keine Nachricht, so kann davon ausgegangen werden, dass er nicht mehr verfügbar ist.

Da der Datenbankserver nicht erweitert werden kann, ist eine Instanz erforderlich, die stellvertretend periodisch eine Nachricht an die Queue sendet. Dazu könnte eine Session-Bean im Server zyklisch den Zustand der Datenbank ermitteln und, wie die anderen Prozesse, die entsprechende Antwortnachricht versenden. Die Struktur dieser Lösung wird in Abbildung 4.3 gezeigt.

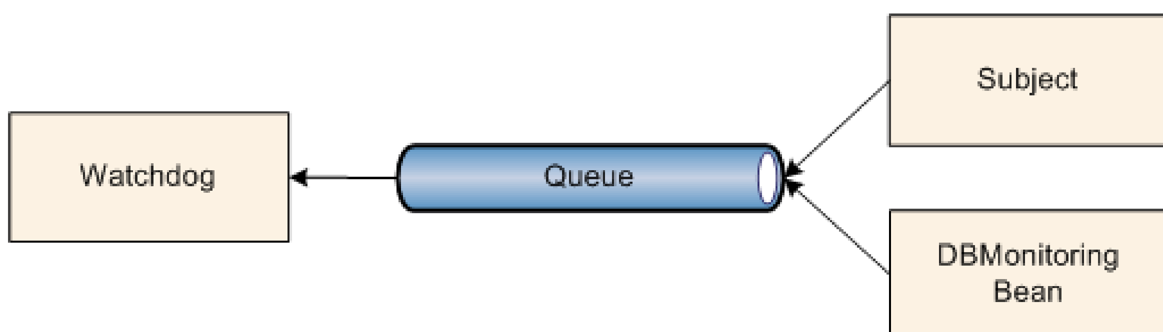

Abbildung 4.3.: Überwachung von Prozessen über eine JMS-Queue

Die Vorteile dieser Lösung bestehen darin, dass jeder beliebige Prozess überwacht werden kann. Es ist lediglich erforderlich, dass der Prozess einen Thread erzeugt, der periodisch eine Nachricht an die Queue versendet. Die Erweiterbarkeit ist damit gewährleistet. Durch die Verwendung von JMS wird außerdem Portabilität sichergestellt, die Lösung würde also auch auf anderen Anwendungsservern funktionieren.

Jedoch beinhaltet dieser Ansatz einige Probleme. Die Subjekte müssen laut den Anforderungen mit einem Anzeigenamen und einer Zykluszeit konfiguriert werden können. Zumindest die Zykluszeit muss den Subjekten bekannt sein, da sie selbst zyklisch Nachrichten versenden. Der Anzeigename kann entweder den Subjekten selbst bekannt sein oder sie verwenden einen internen Namen, der vom Watchdog auf den jeweiligen Anzeigenamen abgebildet wird. In beiden Fällen liegen die Konfigurationsparameter an mehreren Stellen verteilt. Dies widerspricht den angestrebten Designprinzipien.

Weiterhin verlangen die Anforderungen, dass die Prozessüberwachung zur Laufzeit zu- und abschaltbar sein soll. Ohne eine Erweiterung des Konzeptes wäre dies nicht möglich, da die Kommunikation über die Warteschlange nur in eine Richtung funktioniert. Diese Anforderung könnte über ein zusätzliches JMS-Topic gelöst werden, über das der Watchdog Befehle an die Subjekte versendet. Dadurch wird die Lösung jedoch weitaus komplexer. Der folgende Ansatz vereinfacht dieses Konzept durch die Verwendung eines einzigen JMS-Topics.

Anfrage-Antwort-Zyklus über Topic

Um die Nachteile des zuletzt vorgestellten Lösungsansatzes zu beheben, müsste die Konfiguration für die Überwachung an eine zentrale Stelle verlegt werden. Außerdem wäre es von Vorteil, wenn die Subjekte *passiv* agierten, d. h. nur auf vom Watchdog ausgelöste Ereignisse reagierten. Der Einsatz von JMS scheint sinnvoll, da er wenige Änderungen an den Prozessen und am Server erfordert und Portabilität ermöglicht.

Dieser Ansatz orientiert sich am ICMP-Protokoll (Internet Control Message Protocol). Dies wird von jedem Netzwerk-Host, also jedem Computer und Router in einem Netzwerk, realisiert und dient zum Austausch von Diagnoseinformationen. Eine Anwendung des Protokolls besteht darin, die Verfügbarkeit eines Hosts zu überprüfen. Dazu wird eine EchoRequest-Nachricht an den betreffenden Host gesendet, welcher daraufhin mit einer EchoReply-Nachricht antwortet. Antwortet der Host über eine bestimmte Zeitspanne nicht, wird davon ausgegangen, dass er nicht erreichbar ist (vgl. Postel 1981). Das Kommandozeilenprogramm Ping verwendet ICMP. Das Prinzip der Echo-Nachrichten lässt sich auf die hier vorliegende Problemstellung übertragen.

Dazu wird ein JMS-Topic erstellt, an dem sich sowohl der Watchdog, als auch die Subjekte anmelden. Der Watchdog sendet zyklisch eine EchoRequest-Nachricht an das Topic, die von

allen Subjekten empfangen wird. Diese antworten daraufhin über das selbe Topic mit einer EchoReply-Nachricht. Der Watchdog empfängt alle Antwortnachrichten und aktualisiert so seine Wissensbasis. Antwortet ein Prozess über eine bestimmte Zeitspanne nicht, so wird davon ausgegangen, dass er nicht mehr verfügbar ist.

Durch die Verwendung einer Message-Driven-Bean kann die Datenbank mit einer ähnlichen Logik wie die Subjekte überwacht werden. Die Message-Driven-Bean abonniert das Watchdog-Topic und prüft bei Empfang einer EchoRequest-Nachricht die Verfügbarkeit der Datenbank. Ist die Datenbank ansprechbar, so wird eine Antwortnachricht gesendet. Ansonsten wird keine Antwortnachricht gesendet und auf die nächste Anfrage des Watchdogs gewartet. Die Struktur dieser Lösung ist in Abbildung 4.4 dargestellt.

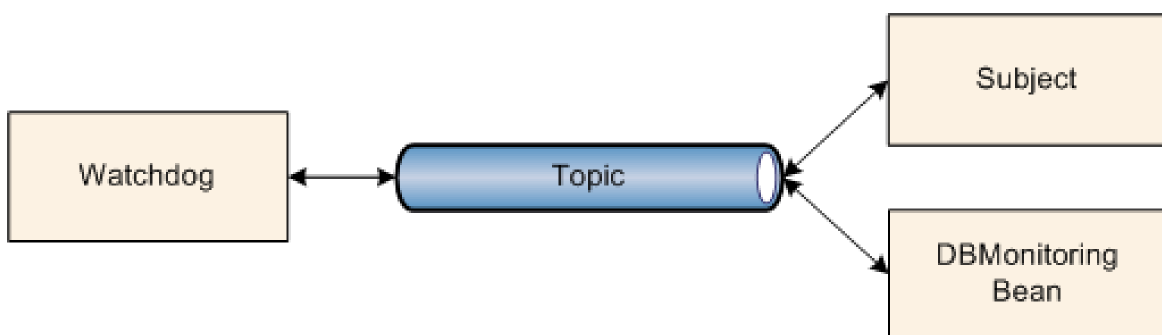

Abbildung 4.4.: Request-Reply-Zyklus über JMS-Topic

Die Vorteile dieses Lösungsansatzes bestehen darin, dass durch die Verwendung von JMS die angestrebte Portabilität gewährleistet ist. Die Überwachung von Subjekten und der Datenbank verläuft auf die gleiche Art und Weise, wodurch die Lösung leicht verständlich ist. Kommen neue Subjekte hinzu, so können sie einfach in die Überwachung aufgenommen werden. Die Klassen und Schnittstellen zur Verwendung durch überwachte Subjekte werden in einer Bibliothek bereitgestellt. Zudem wird die Konfiguration der Überwachung zentral im Server verwaltet, die Subjekte reagieren lediglich auf empfangene Nachrichten. Dadurch ist, außer den Verbindungsinformationen für das JMS-Topic und einem internen Namen, keinerlei Konfiguration in den Subjekten notwendig. Die restliche Konfiguration geschieht vollständig im HyperTest-Server.

Zusammenfassung

Es wurden zwei Lösungsansätze für die Realisierung der Prozessüberwachung dargestellt. Abschließend soll hier eine kurze Bewertung stattfinden.

Der erste Ansatz verwendet eine JMS-Queue, über die Subjekte periodisch eine Nachricht an den Server senden. Dieser Ansatz begünstigt die Portabilität auf andere Anwendungsserver. Allerdings ist die Konfiguration und Steuerung der Überwachung in diesem Fall komplex. Der zunächst simpel scheinende Ansatz müsste erweitert werden, um alle Anforderungen erfüllen zu können, wodurch er seine Einfachheit verliert.

Der zweite Ansatz verwendet ein JMS-Topic für die Kommunikation und orientiert sich an den Echo-Nachrichten des ICMP-Protokolls. Mit diesem Ansatz lassen sich alle Anforderungen erfüllen. Auch die angestrebten Designprinzipien werden durchgängig eingehalten. Dieser Ansatz ist für die Umsetzung geeignet und wird daher im Folgenden Verwendung finden.

4.2.2. Feinentwurf der Watchdog-Subkomponente

Für die Verwendung des Watchdog durch einen Klienten werden die Schnittstellen *WatchdogControl* und *WatchdogQuery* und ein ServiceProvider, welcher diese beiden Schnittstellen implementiert, bereitgestellt. Die Control-Schnittstelle enthält Operationen zur Konfiguration der Watchdog-Subkomponente. Dazu gehören das Hinzufügen und die Konfiguration von Subjekten sowie das Starten und Stoppen der Überwachung. Die Query-Schnittstelle dient allein zur Abfrage des Zustands einzelner oder aller Subjekte. Der ServiceProvider kapselt die Kommunikation mit dem Server und ruft dort eine entsprechende Session-Bean, die *WatchdogServiceBean*, auf.

Server

Die Klassen und Schnittstellen, die den Kern der Watchdog-Subkomponente bilden, sind in Abbildung 4.5 dargestellt. Die *WatchdogServiceBean* dient als serverseitige Fassade für die Komponente und leitet Aufrufe von Klienten an die entsprechenden Klassen weiter. Sie implementiert außerdem die *TimedObject*-Schnittstelle, wodurch sie den standardisierten Timer-Dienst des Anwendungsservers benutzen kann. Der Timer-Dienst wird verwendet, um periodisch Nachrichten an die überwachten Subjekte zu senden. Dazu wird für jedes Subjekt ein periodisch ablaufender Timer erzeugt und gestartet. Ihm wird der Name des jeweiligen Subjekts bei der Erzeugung übergeben, um eine korrekte Zuordnung zu ermöglichen. Der Name eines Subjekts fungiert als systemweit eindeutiger Schlüssel für einen überwachten Prozess.

Die Klasse *WatchdogTopicPublisher* kapselt die Logik zum Erstellen und Versenden von Nachrichten an das JMS-Topic. Läuft der Timer für ein Subjekt ab, so wird der Subjektname ermittelt und mithilfe einer *WatchdogTopicPublisher*-Instanz eine Nachricht an das JMS-Topic versandt. Gleichzeitig wird bei Ablauf des Timers der aktuelle Zustand des Subjekts in der Wissensbasis aktualisiert. Es wird geprüft, ob die Differenz zwischen der aktuellen

Systemzeit und der letzten Antwortzeit des Subjekts größer als die Zykluszeit für die Überwachung des Subjekts ist. Ist dies der Fall, so hat das Subjekt seit einer längeren Zeit als der konfigurierten Zykluszeit keine Antwort gesendet und es wird davon ausgegangen, dass es nicht mehr verfügbar ist.

Für den Empfang der Antwortnachrichten von den Subjekten existiert eine Message-Driven-Bean, die *SubjectReplyBean*. Bei Ankunft einer Nachricht vom JMS-Topic ruft der Anwendungsserver die `onMessage`-Methode der Bean auf. Sie verwendet einen *MessageHandler*, welcher eine `handleMessage`-Methode bereitstellt und die Logik zur Verarbeitung einer Nachricht kapselt. In diesem Fall ist die implementierende Klasse der *SubjectStateManager*, eine MBean zur Speicherung des aktuellen Systemzustands. Wird eine Nachricht empfangen, so wird sie an den *SubjectStateManager* weitergegeben und dort zur Aktualisierung des gespeicherten Subjektzustandes verwendet.

Fragt ein Klient den Zustand eines Subjekts ab, so wird ein *SubjectState*-Objekt vom *SubjectStateManager* zurückgegeben. Dieses enthält den Namen des Subjekts, seine Verfügbarkeit und den Zeitpunkt der zuletzt empfangenen Antwortnachricht.

Die Beschreibungen der Subjekte, mit internem Namen, Anzeigenamen und Zykluszeit werden in Entity-Beans verwaltet und in der Datenbank gespeichert. Auf diese Weise müssen die Subjekte nur ein einziges Mal konfiguriert werden und gehen bei einem Neustart des Systems nicht verloren. Die Verfügbarkeit der Subjekte wird zunächst nicht persistiert und es wird keine Historie über den Systemzustand gespeichert.

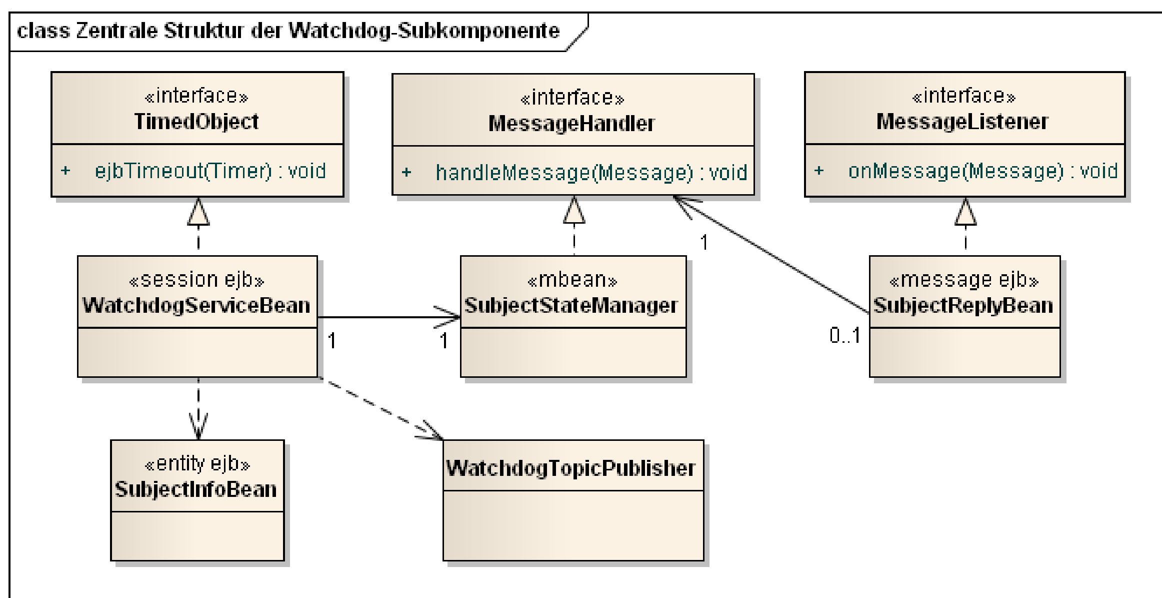

Abbildung 4.5.: Zentrale Struktur der Watchdog-Subkomponente

Subjekte

Die von den überwachten Subjekten verwendeten Klassen und Schnittstellen sind in Abbildung 4.6 dargestellt. Die zentrale Abstraktion ist hier die *WatchdogSubject*-Schnittstelle, welche nur Methoden zum Starten und Stoppen des überwachbaren Subjekts enthält. Die implementierende Klasse *WatchdogSubjectImpl* bekommt den Namen des Subjekts bei der Erzeugung übergeben und fügt ihn in jede Antwortnachricht an den Server ein. Dadurch können die Antwortnachrichten im Server dem jeweiligen Subjekt zugeordnet werden.

Die Klasse *WatchdogSubjectImpl* verwendet intern einen *WatchdogTopicSubscriber*, um Nachrichten vom JMS-Topic zu empfangen. Dieser fungiert als *MessageListener* und dient dazu, den Empfang von Nachrichten vom JMS-Topic zu kapseln. Auch hier wird die Nachricht direkt an eine *MessageHandler*-Implementierung weitergegeben, den *WatchdogRequestHandler*. Dieser verwendet wiederum einen *WatchdogTopicPublisher*, um eine Antwortnachricht an das JMS-Topic zu versenden.

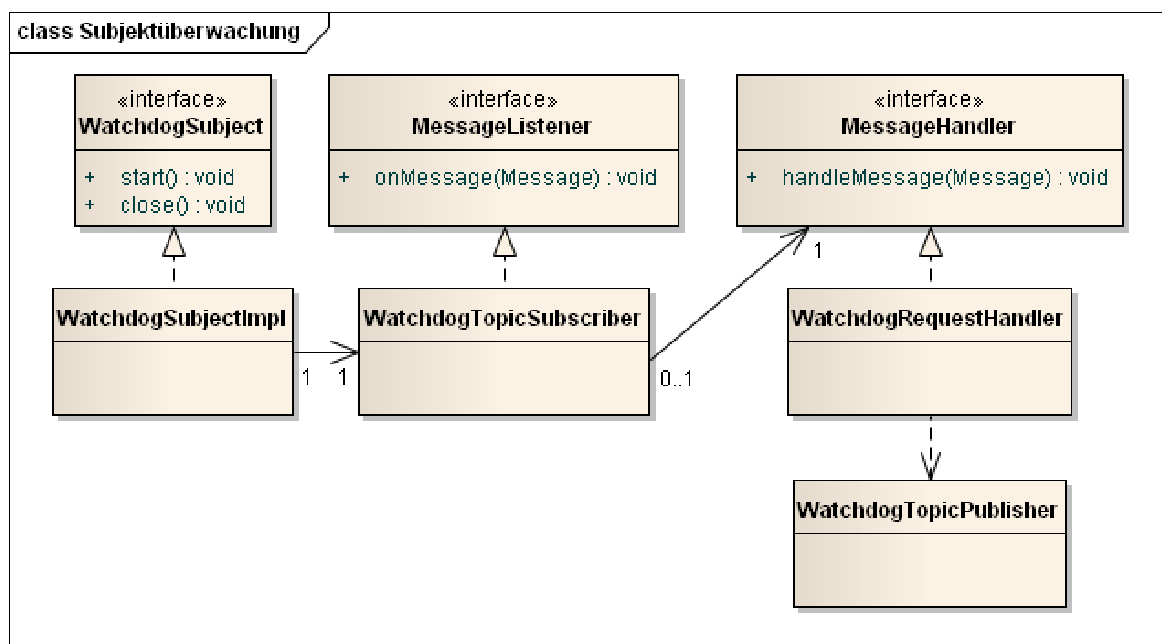

Abbildung 4.6.: Beteiligte Klassen der Subjektüberwachung

Datenbank

In Abbildung 4.7 wird die Struktur der Datenbanküberwachung gezeigt. Sie unterscheidet sich nur geringfügig von der Subjekt-Überwachung. Der Empfang von Anfragenachrichten geschieht hier allerdings durch die Message-Driven-Bean *DatabaseMonitoringBean* anstatt

durch eine *WatchdogTopicSubscriber*-Instanz. Die Klasse *DatabaseMonitoringConfig* stellt über die `getValidationQuery`-Methode eine einfache SQL-Abfrage zur Verfügung, welche auf der Datenbank ausgeführt wird, um festzustellen, ob diese auf Anfragen reagiert. Die Abfrage wird vorher vom Administrator konfiguriert. Da es sich um eine MBean handelt, kann die Konfiguration auf einfache Weise über JMX erfolgen.

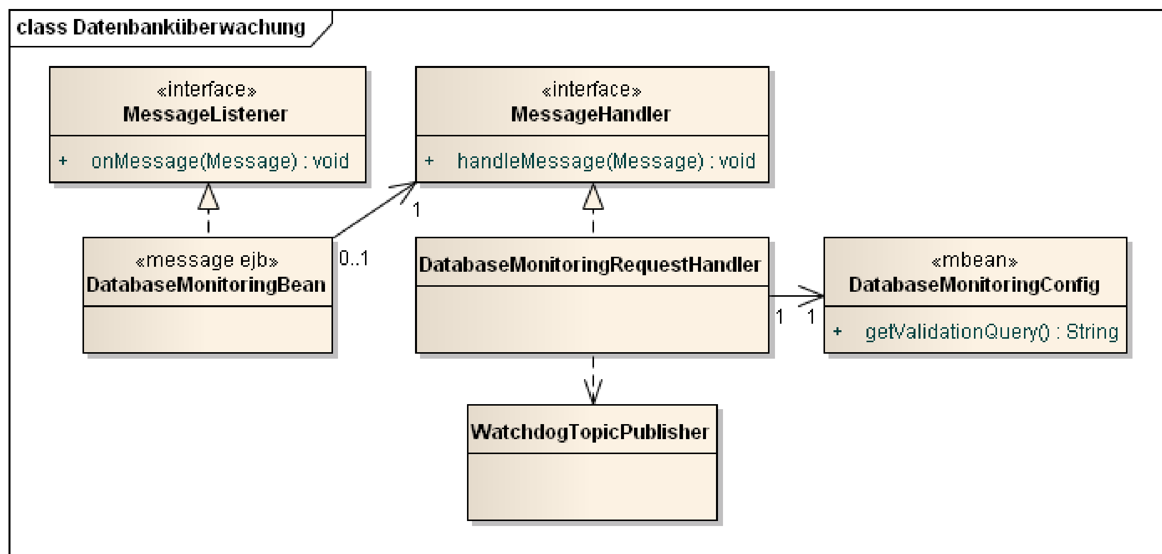

Abbildung 4.7.: Beteiligte Klassen der Datenbanküberwachung

4.2.3. Ablauf einer Zustandsabfrage

In Abbildung 4.8 ist der Ablauf einer Abfrage zur Ermittlung des Datenbankzustands dargestellt. Der Watchdog sendet über das JMS-Topic eine EchoRequest-Nachricht an die *DatabaseMonitoringBean*, welche die empfangene Nachricht direkt an den *DBMonitoringRequestHandler* weitergibt. Dieser ermittelt daraufhin die zu verwendende SQL-Testabfrage von der Klasse *DatabaseMonitoringConfig* und versucht sie auf der Datenbank auszuführen. Verläuft das erfolgreich, so wird eine EchoReply-Nachricht an das JMS-Topic gesendet. Andernfalls wird keine Antwortnachricht gesendet und auf die nächste EchoRequest-Nachricht gewartet.

Die Darstellung ist stark vereinfacht. Tatsächlich sind einige Hilfklassen, wie z. B. der *WatchdogTopicPublisher* zum Versand der Nachrichten an das JMS-Topic und die *SubjectReplyBean* für den Empfang der Antwortnachrichten vom JMS-Topic, beteiligt.

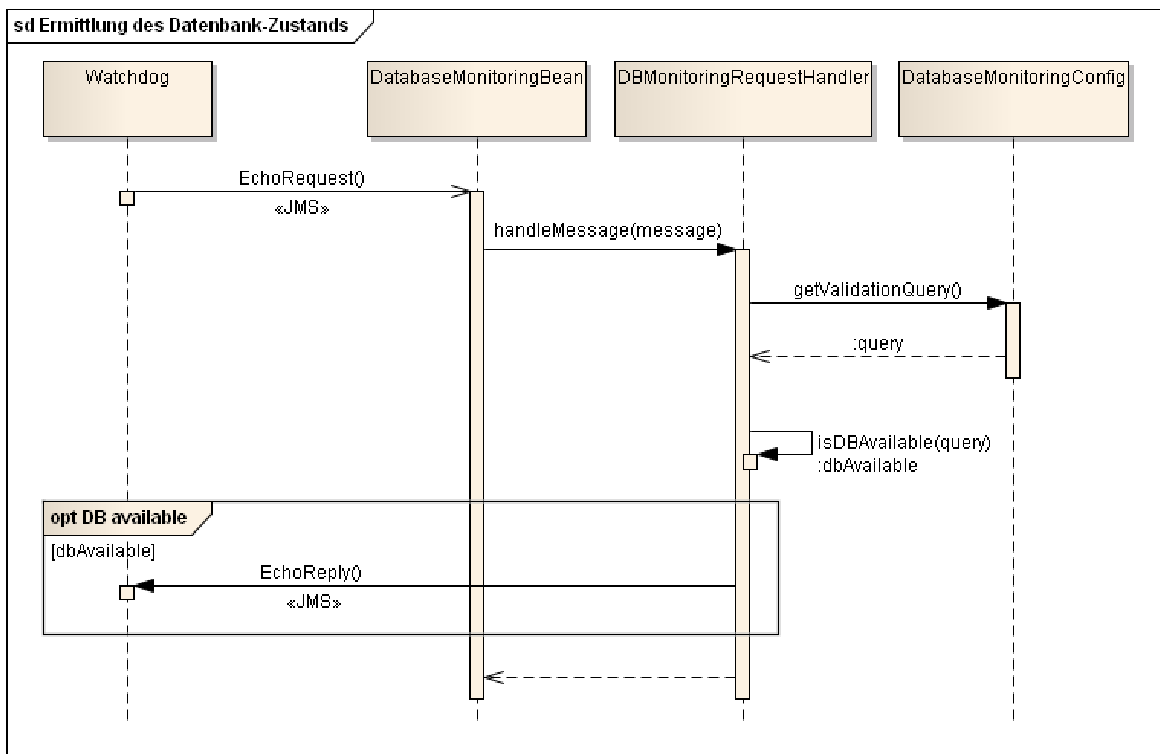

Abbildung 4.8.: Ermittlung des Datenbankzustands

4.3. Statistikerhebung

Die Statistikerhebung bildet den zweiten Aspekt der Monitoring-Komponente und wird durch die Statistics-Subkomponente umgesetzt.

Bei der Statistikerhebung besteht ein zentrales Entwurfsproblem darin, die Statistikerhebung transparent zum bestehenden System hinzuzufügen. Dabei soll der bestehende Anwendungscode nicht verändert werden. Besonders relevant ist dies bei der Erhebung von statistischen Daten über serverseitige Operationen. Die Statistics-Subkomponente muss Möglichkeiten bereitstellen, Daten in die Statistiksammlung einzufügen und diese abzufragen. Außerdem muss die Erhebung von Statistiken konfigurierbar sein. Wichtiges Ziel soll dabei sein, neue Kategorien für die Statistikerhebung einfach hinzufügen zu können.

Im Folgenden wird zunächst die Architektur der Statistics-Subkomponente erläutert. Im Anschluss daran werden verschiedene Ansätze zur Integration der Statistikerhebung in das bestehende System diskutiert.

4.3.1. Feinentwurf der Statistics-Subkomponente

Im Rahmen der Statistikerhebung werden die verschiedenen zu erhebenden Daten in Kategorien eingeteilt. Die in dieser Arbeit zu realisierenden Kategorien sind zunächst die Benutzer-, Hauptspeicher- und Methodenaufrufstatistik. Außerdem soll es laut den Anforderungen einfach möglich sein, neue Kategorien hinzuzufügen, ohne die bestehende Statistikerhebung verändern zu müssen.

Der grundsätzliche Aufbau der Statistics-Subkomponente ist dem der Watchdog-Subkomponente sehr ähnlich. Er wird in Abbildung 4.9 gezeigt. Die Schnittstellen *StatisticsControl* und *StatisticsQuery* und ein ServiceProvider, der diese beiden Schnittstellen implementiert, erlauben den Klienten die Benutzung der Statistikerhebung. Eine Session-Bean, die *StatisticsServiceBean*, dient als serverseitige Fassade für die Subkomponente und zur Verwaltung von Timern. Da die erhobenen Daten periodisch in bestimmten Zeitabständen gespeichert werden müssen, kann hier ebenfalls der standardisierte Timer-Dienst des Anwendungsservers verwendet werden. Die Timer bekommen bei der Initialisierung den jeweiligen Kategorienamen übergeben. Läuft ein Timer ab, so kann der Kategoriename ermittelt und verwendet werden, um die Erhebung der entsprechenden Daten zu veranlassen.

Die zentrale Instanz zur Erhebung von Daten ist dabei der *StatisticsCollector*. Er verwaltet sämtliche Daten und implementiert die *StatisticsCollection*-Schnittstelle, welche das Hinzufügen von Daten zur Statistiksammlung erlaubt. Ihm wird beim Start des Anwendungsservers eine Liste von Klassen übergeben, welche die *StatisticsCollectorAction*-Schnittstelle

implementieren. Er erzeugt von jeder dieser Klassen eine Instanz und speichert sie intern. Außerdem übergibt er eine Referenz auf sich selbst über eine Setter-Methode an die neu erzeugten Action-Instanzen.

Diese können daraufhin die Methoden der Schnittstelle *StatisticsCollection* verwenden, um Werte zur Statistiksammlung hinzuzufügen. Jeder erhobene Wert wird dabei einer systemweit eindeutigen Kategorie zugeordnet. Die *StatisticsCollectorAction*-Schnittstelle enthält eine Methode `getCategoryName`, die der *StatisticsCollector* dazu verwendet, die Kategorie einer Action zu ermitteln. Pro Kategorie kann der *StatisticsCollector* nur eine einzige Action-Instanz zur Erhebung von Daten verwenden.

Läuft der Timer für eine Kategorie in der *StatisticsServiceBean* ab, so ruft diese die `collect`-Methode des *StatisticsCollectors* unter Angabe des Kategorienamens auf. Der *StatisticsCollector* ermittelt daraufhin die *StatisticsCollectorAction*-Instanz, die zu der angegebenen Kategorie gehört. Er führt sie durch einen Aufruf der `execute`-Methode aus. Die Action ermittelt im Folgenden die Daten und fügt sie über die *StatisticsCollection*-Schnittstelle zur Statistiksammlung hinzu. Nachdem der Aufruf der Action beendet ist, persistiert der *StatisticsCollector* die erhobenen Daten.

Methodenaufrufstatistiken

Die zusätzlich vorhandene *MethodStatisticsCollection*-Schnittstelle dient zur Erhebung von Methodenaufrufstatistiken. Der *MethodStatisticsCollector* implementiert diese Schnittstelle und speichert die Ausführungszeiten von Methoden. Er enthält die Konfiguration der in der Statistik zu erfassenden Methoden. Zusätzlich können einfache Filter definiert werden, die Bedingungen anhand der Aufrufparameter oder Rückgabewerte eines Methodenaufrufs prüfen. Diese Filter werden in Abschnitt 4.3.3 genauer erläutert.

Der *MethodStatisticsCollector* bietet eine `capture`-Methode zur Erfassung von Methodenaufrufen an. Diese bekommt als Parameter den Klassen- und Methodennamen, die Parameter des Aufrufs, den Rückgabewert und die Ausführungsdauer der Methode übergeben. Es wird geprüft, ob der jeweilige Aufruf erfasst werden muss. Fällt diese Prüfung positiv aus, so wird die Ausführungszeit gespeichert. Der *MethodStatisticsCollector* dient lediglich zur Aggregation der Werte. Für die Persistierung von Daten ist ausschließlich der *StatisticsCollector* verantwortlich. Daher werden die Werte am Ende einer Zykluszeit durch eine Action vom *MethodStatisticsCollector* abgefragt und über die *StatisticsCollection*-Schnittstelle zur Statistiksammlung hinzugefügt. Auf diese Weise fügt sich die Methodenaufrufstatistik in die Erhebung von Daten ein, ohne eine spezielle Behandlung zu erfordern.

Auf welche Weise Informationen über Methodenaufrufe ermittelt werden, wird in Abschnitt 4.3.4 ausführlich erläutert.

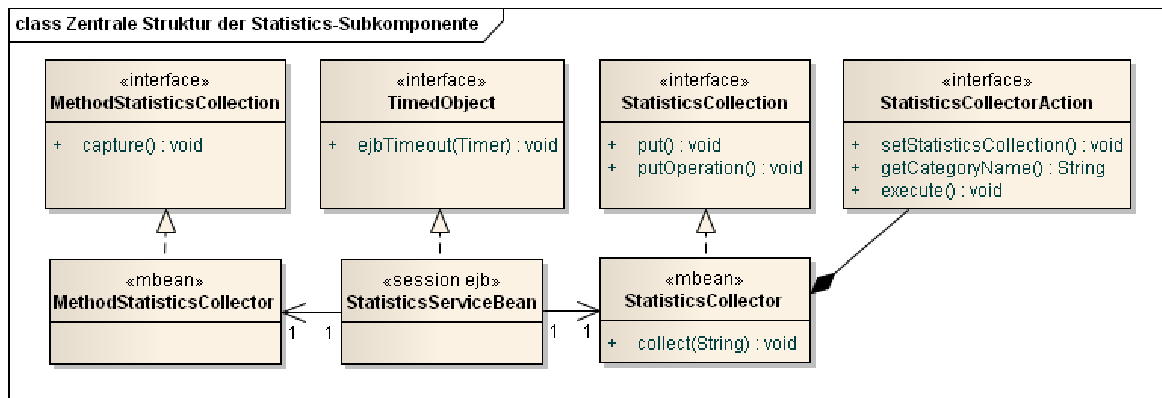

Abbildung 4.9.: Zentrale Struktur der Statistics-Subkomponente

Erhebung von Daten mithilfe von Actions

Durch die *StatisticsCollectorAction*-Schnittstelle können auf einfache Weise Daten für neue Kategorien erhoben werden. Es muss lediglich eine neue Implementierung der Schnittstelle bereitgestellt und konfiguriert werden. Dabei darf zu jeder Kategorie nur genau eine Action-Instanz vorhanden sein. Es wird außerdem unterschieden, welcher Art die erhobenen Daten sind:

Allgemeine Statistiken Diese bestehen aus einfachen Werten. Ein Beispiel ist die Hauptspeicherstatistik: Am Ende jeder Zykluszeit wird der aktuelle Zustand, z. B. der belegte und der insgesamt verfügbare Speicher im HyperTest-Server, ermittelt.

Operationenstatistiken Sie enthalten beliebig viele Werte für ein Zeitintervall. Pro Zeitintervall können mehrere verschiedene Operationen erfasst werden. Das hat zur Folge, dass ein Datensatz für jede erfasste Operation existiert. Jeder Datensatz enthält die erhobenen Werte und Metadaten, welche die jeweilige Operation beschreiben. Ein weiterer Unterschied zu allgemeinen Statistiken besteht darin, dass Operationenstatistiken üblicherweise aggregierte Werte enthalten. Würden z. B. Informationen über alle Methodenaufrufe gespeichert, wäre das Datenaufkommen zu groß. Deshalb werden hier nur die durchschnittliche, minimale und maximale Ausführungszeit und die Anzahl der Aufrufe einer Methode pro Zykluszeit gespeichert.

Allgemeine Statistiken lassen sich über die `put`-Methode der *StatisticsCollection*-Schnittstelle zur Statistiksammlung hinzufügen, während Operationenstatistiken über die `putOperation`-Methode hinzugefügt werden. Alle Action-Klassen, die Operationenstatistiken bereitstellen, müssen zur Unterscheidung die Markierungsschnittstelle *OperationStatisticsProvider* implementieren. Hier hätte auch eine Annotation verwendet werden können. Da im bestehenden Anwendungscode jedoch bereits Markierungsschnittstellen verwendet

werden, wurde dies aus Gründen der Einheitlichkeit übernommen. Eine Übersicht über die im Rahmen dieser Arbeit realisierten Actions ist in Abbildung 4.10 dargestellt.

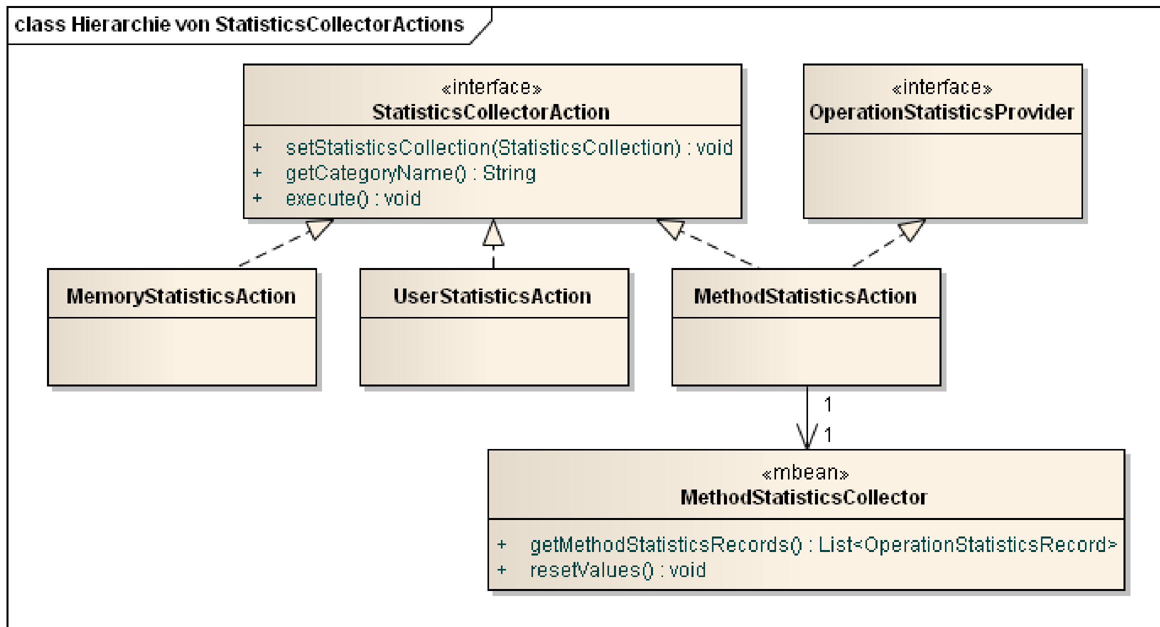

Abbildung 4.10.: Implementierungen der Schnittstelle *StatisticsCollectorAction*

Die *MethodStatisticsAction* fragt die aggregierten Werte vom *MethodStatisticsCollector* ab und setzt die dort gespeicherten Werte anschließend mithilfe der `resetValues`-Methode zurück. Der *MethodStatisticsCollector* kann daraufhin Daten für das folgende Zeitintervall sammeln. Für andere Operationenstatistiken könnten ähnliche MBeans implementiert werden.

Woher die konkreten *StatisticsCollectorAction*-Implementierungen ihre Daten beziehen, ist jeweils unterschiedlich. Die Information darüber, woher die Anzahl der angemeldeten Benutzer beschafft werden kann, ist z. B. in der *UserStatisticsAction* gekapselt. Sollte sich das in der Zukunft ändern, so kann die Implementierung ausgetauscht werden. Diese Struktur lehnt sich an das Befehls-Entwurfsmuster an (vgl. Gamma 2008, S. 273 ff.).

4.3.2. Datenmodell für statistische Daten

Für die Verwaltung und Persistierung der statistischen Daten muss ein Datenmodell entwickelt werden, das möglichst offen für zukünftige Erweiterungen ist. Es ist in Abbildung 4.11 dargestellt. Allen statistischen Daten ist gemeinsam, dass sie für eine bestimmte Zeitspanne erhoben wurden und genau einer Kategorie angehören. Diese gemeinsamen Eigenschaften

finden sich in der *StatisticsRecord*-Entität wieder. Sie repräsentiert einen Datensatz für eine Kategorie und ein Zeitintervall. Zu einem *StatisticsRecord* können beliebig viele Werte gehören.

Ein einzelner Wert wird in einer *StatisticsRecordValue*-Entität als Schlüssel-Wert-Paar gespeichert. Ein solches Paar könnte z. B. für die Hauptspeicherstatistik den Schlüssel „Used“ und als Wert die Anzahl belegter Kilobytes enthalten. Kommen in Zukunft weitere Kategorien mit anderen Werten hinzu, so können ein weiterer Kategoriename und neue Schlüssel eingeführt werden.

Statistische Daten für Operationen sind weitaus komplexer. Sie benötigen zusätzliche Informationen über die Operation, wie z. B. den Klassen- und Methodennamen eines Methodenaufrufs oder eine ausgeführte SQL-Abfrage. Diese Metadaten werden in Form von *OperationMeta*-Entitäten abgelegt. Eine *OperationMeta*-Entität beinhaltet, genau wie eine *StatisticsRecordValue*-Entität, ein Schlüssel-Wert-Paar. Ein Datensatz für eine Operation wird durch eine *OperationStatisticsRecord*-Entität repräsentiert. Ein solcher Operationenstatistik-Datensatz enthält mindestens ein Metadatum. Außerdem verweist er auf genau einen allgemeinen Datensatz, welcher die Kategorie und den Zeitraum der Erhebung enthält. Auf diese Weise werden Redundanzen vermieden.

Operationendatensätze referenzieren direkt die zugehörigen Werte. Ansonsten wäre eine Zuordnung zwischen den Werten und den Operationen, auf welche diese sich beziehen, nicht möglich. Es muss von der Anwendung sichergestellt werden, dass nur entweder die *StatisticsRecords*- oder die *OperationStatisticsRecords*-Entität Werte referenziert.

Da sowohl bei allgemeinen Statistiken, als auch bei Operationenstatistiken ein allgemeiner Datensatz erzeugt wird, muss die Anwendung sicherstellen, dass die Daten auf korrekte Weise abgefragt oder gelöscht werden. So müssen bei der Abfrage aller allgemeinen Datensätze für einen bestimmten Zeitraum alle Datensätze ausgeschlossen werden, die zu einem Operationenstatistik-Datensatz gehören. Zur Abfrage von Operationenstatistik-Datensätzen wird eine separate Methode in der *StatisticsQuery*-Schnittstelle angeboten. Beim Löschen eines Operationenstatistik-Datensatzes muss der referenzierte allgemeine Datensatz ebenfalls gelöscht werden, sofern er von keinem weiteren Operationenstatistik-Datensatz referenziert wird.

4.3.3. Filter für Parameter und Rückgabewerte

Bei der Erhebung von Methodenaufrufstatistiken werden kumulierte Werte für ein Zeitintervall gespeichert, um das Datenaufkommen möglichst gering zu halten. Dabei gehen Informationen über die einzelnen Aufrufe verloren, sodass übergebene Parameter und der Rückgabewert der Aufrufe anhand der gespeicherten Werte nicht mehr erkennbar sind.

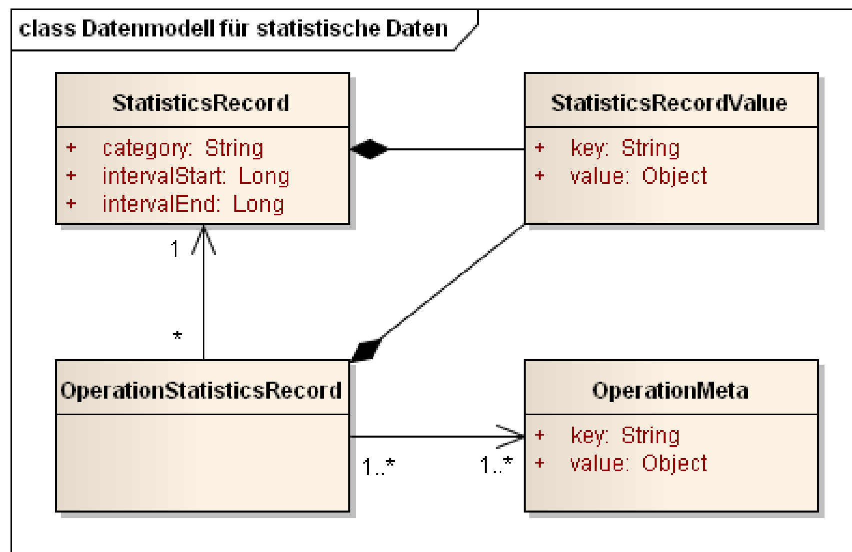

Abbildung 4.11.: Datenmodell für statistische Daten

Um eine gezieltere Erhebung von Daten zu ermöglichen, können für Methodenaufrufe Filter definiert werden. Über diese kann festgestellt werden, ob für die übergebenen Parameter oder den Rückgabewert bestimmte Bedingungen gelten. Dadurch ist es möglich, die erfassten Methodenaufrufe genauer einzugrenzen. Im Rahmen dieser Arbeit werden zunächst nur zwei einfache Filter und einige logische Operatoren (Konjunktion, Disjunktion und Negation) realisiert. Einfache Filter sind zunächst:

1. Ein *InstanceOfFilter*, der mithilfe der `instanceof`-Operation prüft, ob ein Objekt die Instanz einer vorgegebenen Klasse ist.
2. Ein *EqualsFilter*, der die textuelle Repräsentation eines Objekts mit einer vorgegebenen Zeichenkette vergleicht.

Die Klassenhierarchie wird so konzipiert, dass sie leicht erweitert werden kann. Der Aufbau entspricht dem Kompositum-Entwurfsmuster, bei dem jede Komponente wiederum aus weiteren Komponenten bestehen kann (vgl. Gamma 2008, S. 239 ff.). So referenziert die *AndFilter*-Klasse zwei weitere Filter, welche wiederum auf andere Filter verweisen können. In Abbildung 4.12 ist dies veranschaulicht.

Die baumartige Struktur ermöglicht das Speichern eines einzigen *Filter*-Objektes pro Methode, so dass es ausreicht, die `apply`-Methode der Wurzel des Baumes aufzurufen. Der Filter-Baum wird dann rekursiv durchlaufen und am Ende ein einziger Wahrheitswert zurückgegeben. Der Filter wird zur Persistierung in der Datenbank in eine String-Repräsentation überführt und innerhalb einer *OperationMeta*-Entität gespeichert.

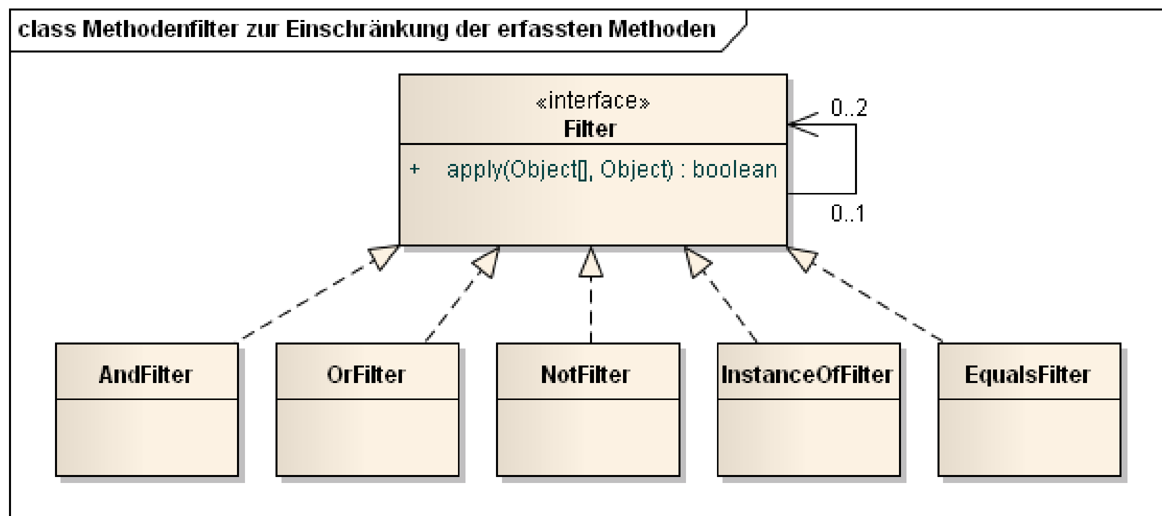

Abbildung 4.12.: Methodenfilter zur Einschränkung der erfassten Methoden

Die Filter werden vom *MethodStatisticsCollector* verwaltet. Dieser verwendet sie bei Aufrufen der `capture`-Methode um festzustellen, ob ein Methodenaufruf erfasst werden muss.

4.3.4. Integration in das bestehende System

In diesem Abschnitt werden verschiedene Möglichkeiten diskutiert, die Statistikerhebung transparent zum bestehenden System hinzuzufügen. Insbesondere bei Methodenaufrufstatistiken ist dies eine Herausforderung. Sollen in Zukunft statistische Daten über Datenbankabfragen, Servlet-Aufrufe o.ä. hinzugefügt werden, so ergibt sich das gleiche Problem.

Interzeptoren

Interzeptoren sind eine Möglichkeit, ein bestehendes Framework transparent um Dienste zu erweitern. Die grundsätzliche Funktionsweise wurde bereits in Kapitel 2 erläutert. Ein nahe liegender Lösungsansatz zur Erhebung von Methodenaufrufstatistiken besteht darin, einen serverseitigen Interzeptor zu implementieren. Dieser wird bei Bean-Aufrufen aktiv, erhebt Informationen über die Aufrufe und übergibt diese an den *MethodStatisticsCollector*. Auch für Servlet-Aufrufe wäre dies über einen ServletFilter, einer Umsetzung des Interzeptor-Konzeptes für HTTP-Requests, möglich.

Da das bestehende HyperTest-System jedoch auf EJB 2.1 basiert, wäre die Implementierung eines solchen Interzeptors nicht auf andere Anwendungsserver portabel. Wie in Kapitel 2 geschildert, wurden Interzeptoren erst in der EJB-Spezifikation Version 3 standardisiert (vgl.

Sun Microsystems 2006, S. 301 ff.). Die hier verwendete Interzeptor-Klasse würde JBoss-spezifische Schnittstellen und ein ebenfalls JBoss-spezifisches *Invocation*-Objekt benötigen. Obwohl auch andere Anwendungsserver Interzeptoren einsetzen, um Services zum EJB-Container hinzuzufügen, ist die konkrete Umsetzung bei der Verwendung von EJB 2.1 stets von dem verwendeten Produkt abhängig. Auch können Interzeptoren nur auf Bean-Aufrufe angewendet werden. Einfache Java-Objekte könnten folglich nicht berücksichtigt werden.

Aspektorientierte Programmierung

Die zweite Lösungsmöglichkeit für die Erhebung von Methodenaufrufstatistiken verwendet aspektorientierte Programmierung. Dabei wird die Logik zur Erhebung der Daten als Aspekt realisiert, mit dem die relevanten Klassen innerhalb des Servers versehen werden. Ein Around-Advice ermöglicht das Hinzufügen von Logik vor und nach einem Methodenaufruf. Auf diese Weise können Informationen über Aufrufe erhoben und an den *MethodStatistics-Collector* übergeben werden.

Die Integration der Statistikerhebung in das System mithilfe von Aspekten ist genauso transparent möglich wie bei der Verwendung von Interzeptoren. Der bestehende Anwendungscode muss nicht verändert werden. Der Anwendungsserver besitzt keine Kenntnis über das Vorhandensein eines AOP-Frameworks. Dies führt zur Unabhängigkeit von einem bestimmten Anwendungsserver und sogar der EJB-Version. Ein weiterer Vorteil besteht darin, dass Aspekte gezielter eingesetzt werden können. So könnten z. B. auch einfache Java-Objekte oder Bibliotheken der Java-Standardbibliothek mit Aspekten instrumentiert werden. Auf diese Weise wäre es u. a. möglich, eine Statistik über den Netzwerkverkehr zu erheben, indem die Java-Netzwerkbibliothek mithilfe eines Aspekts instrumentiert wird.

Die meisten AOP-Frameworks unterstützen Load-Time-Weaving, also das Zusammenfügen von Aspekt- zum Anwendungscode während des Systemstarts. Zwar verlangsamt sich dadurch der Start des Servers, zur Laufzeit ist diese Lösung jedoch performanter als Interzeptoren, da keine zusätzlichen Proxyobjekte notwendig sind. Genauere Informationen zum Performanzunterschied zwischen Proxies und Bytecode-Instrumentierung können in Arendsen (2007) nachgelesen werden.

Zusammenfassung

Interzeptoren und aspektorientierte Programmierung eignen sich gleichermaßen, um Dienste transparent zu einer bestehenden Anwendung hinzuzufügen. Allerdings haben Interzeptoren im Falle von HyperTest einige Nachteile: Durch die Verwendung von EJB-Version 2.1 ist eine Realisierung mithilfe von Interzeptoren abhängig vom Anwendungsserver. Erst in EJB

Version 3 wurden Interzeptoren standardisiert. Zudem können diese nur bei EJB-Aufrufen verwendet werden.

Aspektorientierte Programmierung kann schnell zu unverständlichen Programmen führen. Dies ist vor allem der Fall, wenn es für die Umsetzung von fachlicher Funktionalität verwendet wird. An dieser Stelle überwiegen die Vorteile aspektorienterter Programmierung jedoch: Es ist unabhängig von einem konkreten Anwendungsserver und der EJB-Version. Außerdem können Aspekte gezielt für eine Menge von Methoden angewandt werden, wodurch sich neue Möglichkeiten für die Erhebung von Statistiken eröffnen.

4.3.5. Ablauf eines entfernten Aufrufs

An dieser Stelle soll anhand eines Beispiels die Erhebung von Methodenaufrufstatistiken verdeutlicht werden. Der Ablauf ist in Abbildung 4.13 dargestellt.

Alle Bean-Methoden werden beim Start des Anwendungsservers mit einem Aspekt instrumentiert. Ein Klient des HyperTest-Systems führt einen entfernten Aufruf auf einer Enterprise-Bean, z. B. einer Session-Bean, aus. Es wird zuerst der Aspektcode ausgeführt. Dieser ermittelt die Systemzeit, bevor er den Aufruf mithilfe der `proceed`-Methode an die Bean-Methode weiterleitet. Nach der Ausführung der Bean-Methode wird im Aspektcode die Dauer des Aufrufs berechnet, indem erneut die Systemzeit ermittelt und die Differenz zu der Zeit vor dem Methodenaufruf gebildet wird.

Anschließend werden die Daten über die `capture`-Methode an den *MethodStatisticsCollector* übergeben. Dieser prüft, ob der Aufruf erfasst werden muss. Dafür werden Klassen- und Methodenname und eventuell vorhandene Filter für die Parameter und den Rückgabewert ausgewertet. Ist das Ergebnis dieser Überprüfung positiv, so wird der aktuelle Wert zur Datensammlung hinzugefügt.

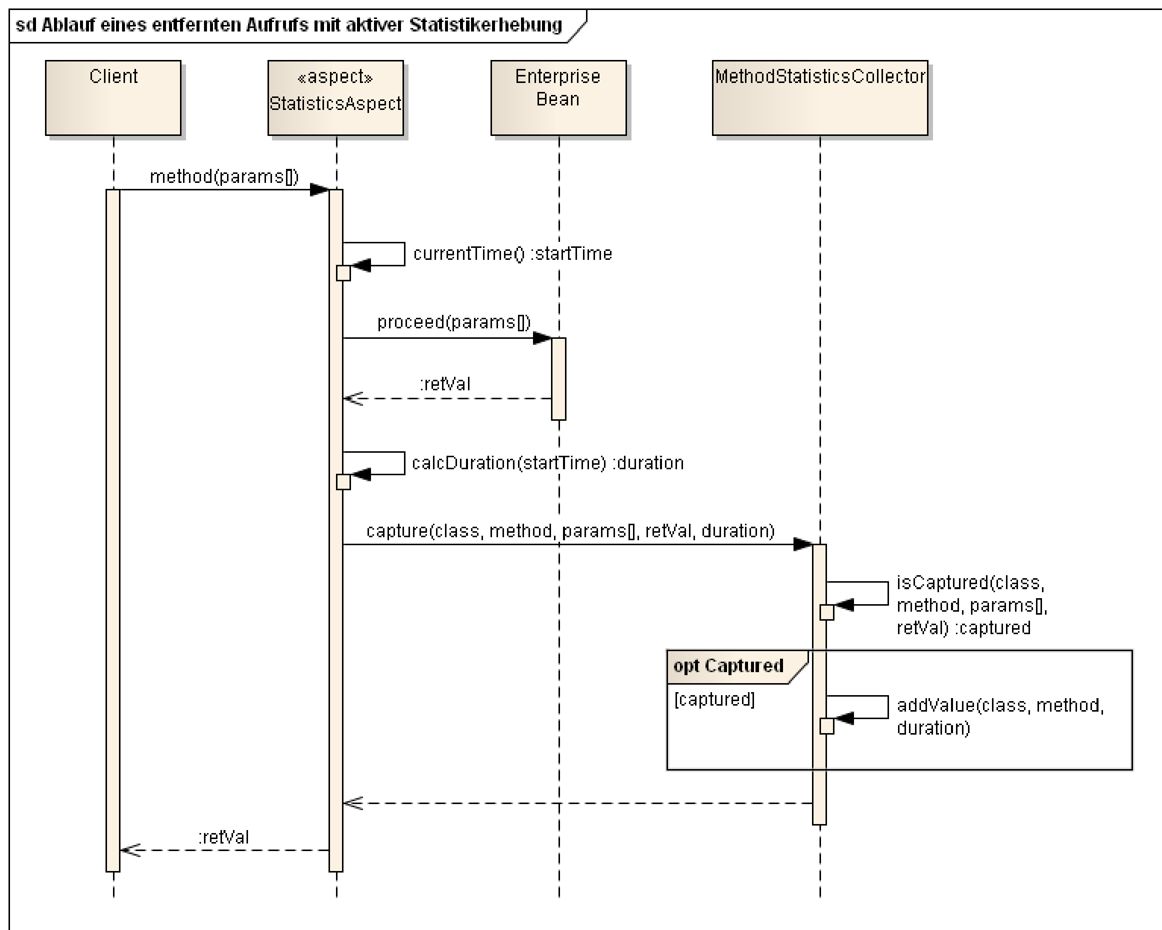

Abbildung 4.13.: Ablauf eines entfernten Aufrufs mit aktiver Statistikerhebung

4.4. Grafische Oberfläche

Die grafische Oberfläche wird so strukturiert, dass Layout und Logik voneinander getrennt sind. Dies wird erreicht, indem zunächst das gesamte Layout mithilfe vorhandener Widget-Klassen realisiert wird. Übergeordnete Widgets, wie z. B. Panels, bieten Methoden an, über die alle in ihnen enthaltenen Widgets abgefragt werden können.

Danach werden Action-Klassen für jedes Panel implementiert. Diese können über die bereitgestellten Methoden des jeweiligen Panels auf die enthaltenen Widgets zugreifen und die erforderliche Logik hinzufügen. Dadurch können Layout und Ablaufsteuerung unabhängig voneinander verändert werden. Zudem ist es möglich, übergreifende Logik für mehrere Panels oder Widgets zu implementieren. Diese Art der Strukturierung lehnt sich an das Vermittler-Entwurfsmuster an (vgl. Gamma 2008, S. 385 ff.).

In Abbildung 4.14 ist ein einfaches Beispiel dargestellt. Zunächst wird das *WatchdogPanel* implementiert. Es bietet Methoden an, um auf die enthaltenen Buttons, Tabellen und Toolbars zuzugreifen. Die *WatchdogPanelAction* bekommt bei der Erzeugung das *WatchdogPanel* übergeben und nutzt dessen Methoden, um Logik zu den Widgets hinzuzufügen. Auf diese Weise kann die *WatchdogPanelAction* z. B. über die `getButtonActivate`-Methode den Button zur Aktivierung der Prozessüberwachung abfragen und ihm einen Listener mit der erforderlichen Programmlogik hinzufügen.

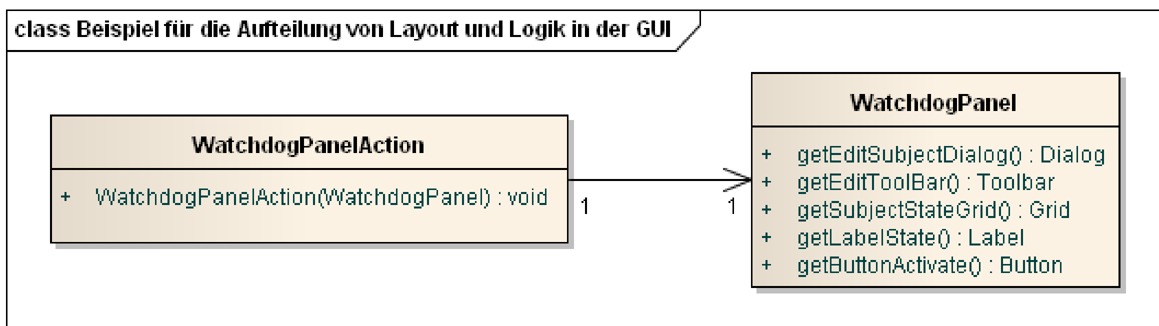

Abbildung 4.14.: Beispiel für die Aufteilung von Layout und Logik in der GUI

Der GWT-Compiler übersetzt klientenseitigen Code in JavaScript, welches dann im Webbrowser des Klienten ausgeführt wird. Auf der Serverseite muss ein *Servlet* implementiert werden, welches innerhalb eines Servlet-Containers eingesetzt wird. Die Kommunikation zwischen dem Webbrowser des Klienten und dem Servlet findet asynchron statt. Dies hat einige Auswirkungen auf das Design.

Ein Beispiel für einen Aufruf ist in Abbildung 4.15 dargestellt. Hier sollen alle verfügbaren Statistik-Kategorien ermittelt werden. Im Klienten wird ein Callback-Objekt erzeugt und bei

dem Servlet-Aufruf als Parameter übergeben. Das Servlet verwendet daraufhin die Klientenschnittstelle der Statistics-Subkomponente, um die benötigten Informationen vom Anwendungsserver abzufragen. Verläuft dies erfolgreich, so ruft das Servlet die `onSuccess`-Methode das Callback-Objektes auf und übergibt das ermittelte Ergebnis. Der klientenseitige JavaScript-Code kann dieses dann entsprechend verarbeiten.

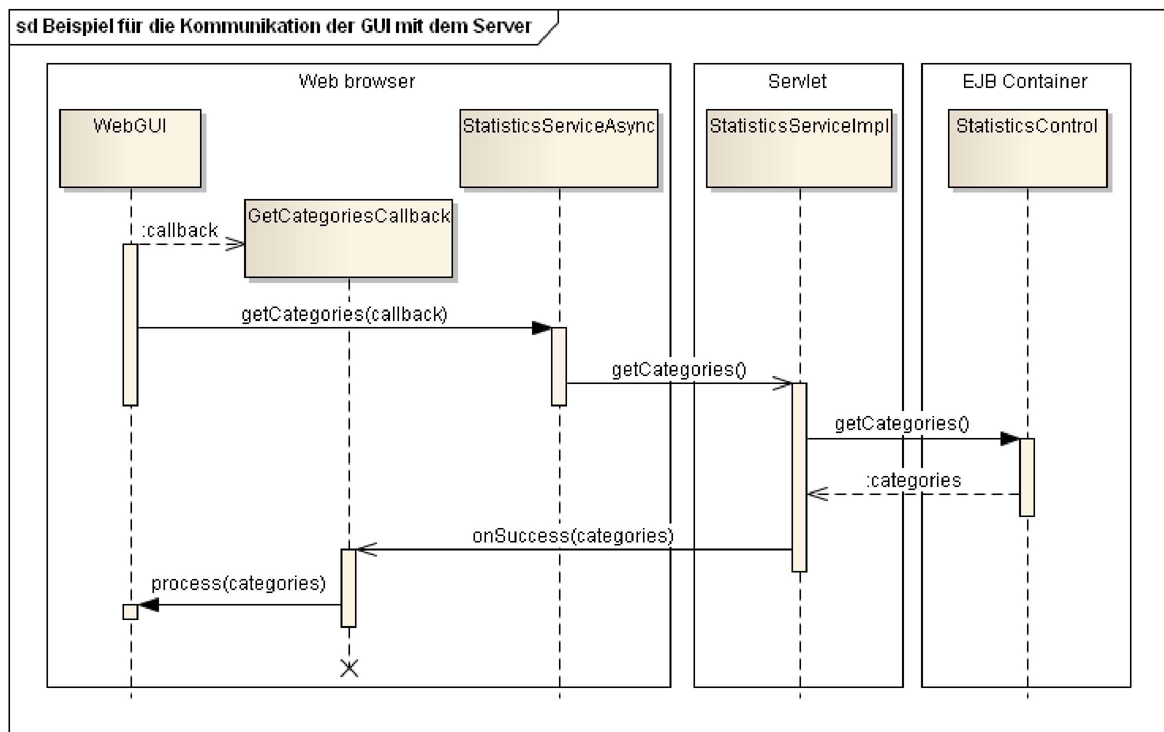

Abbildung 4.15.: Beispiel für die Kommunikation der GUI mit dem Server

Ein Entwurf des Layouts der grafischen Oberfläche ist in Anhang A zu finden.

5. Realisierung

In diesem Kapitel werden einige relevante Details der exemplarischen Realisierung vorgestellt. Der gesamte Quellcode der Monitoring-Komponente liegt dieser Arbeit auf CD bei. Daher werden nur ausgewählte Einzelheiten gezeigt.

5.1. Prozessüberwachung

5.1.1. Zyklischer Versand von Anfragenachrichten

Im HyperTest-Server werden Timer zum periodischen Versand von EchoRequest-Nachrichten an die überwachten Subjekte verwendet. Einem Timer können beliebige Daten in Form eines serialisierbaren Objektes übergeben werden. Dafür wird eine Klasse *TimerInfo* verwendet, welche einen Namen und ein Datum beliebigen Datentyps enthalten kann. Die Beziehung zwischen den Klassen *Timer* und *TimerInfo* ist in Abbildung 5.1 dargestellt. Im Namensfeld wird hier der Name des Subjekts gespeichert, im Datumsfeld die Zykluszeit für das Subjekt. Diese wird zur Aktualisierung der Wissensbasis benötigt.

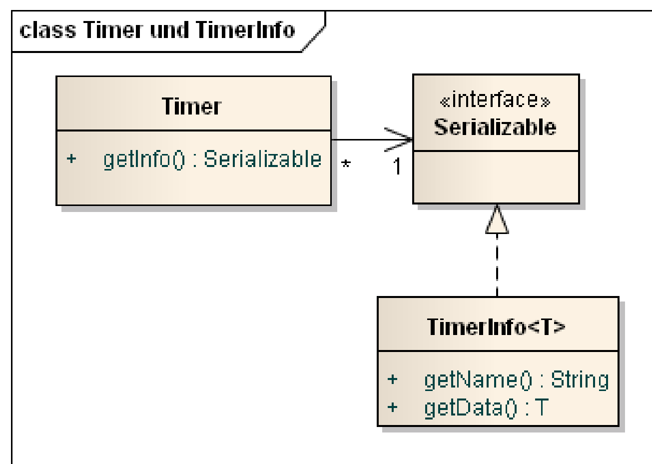

Abbildung 5.1.: *Timer* und *TimerInfo*

Die einzigen Ereignisse, die während der Überwachung innerhalb des Servers auftreten, sind der Ablauf eines Timers oder der Empfang einer Antwortnachricht vom JMS-Topic. Antwortet ein Subjekt nicht auf die Anfragen des Watchdog, so kann nur der Ablauf des Timers dazu verwendet werden, die Wissensbasis zu aktualisieren. Daher wird bei jedem Timer-Ablauf zuerst eine Nachricht versendet und anschließend der Subjektstatus überprüft. Der vollständige Ablauf ist in Abbildung 5.2 dargestellt.

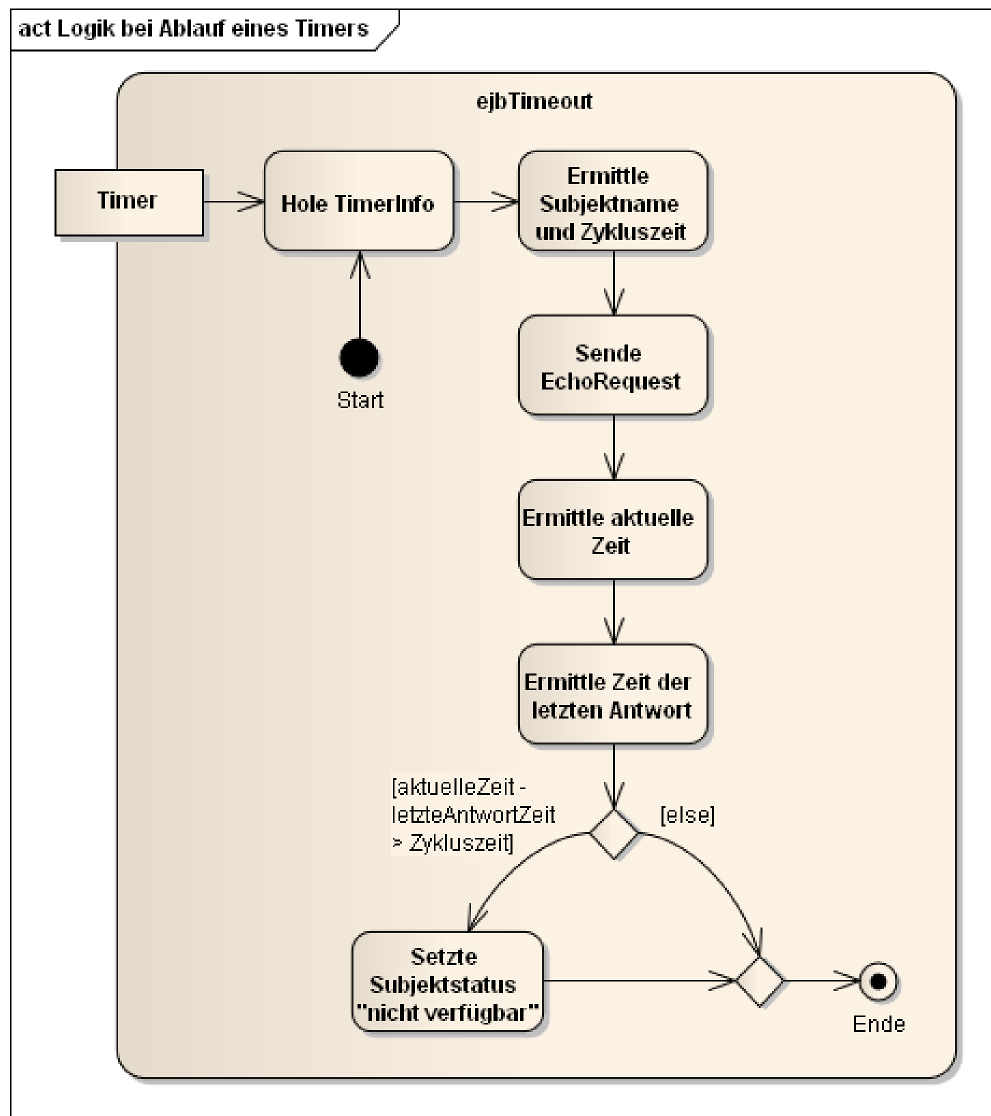

Abbildung 5.2.: Logik bei Ablauf eines Timers innerhalb der *WatchdogServiceBean*

5.1.2. Aufbau der JMS-Nachrichten

Der Aufbau der verwendeten Nachrichten ist in Abbildung 5.3 zu sehen. Sie sind vom Nachrichtentyp *MapMessage*, welcher wie eine Map mit Schlüssel-Wert-Paaren strukturiert ist. Der erste Bereich in der Grafik zeigt die Eigenschaften (Properties) der Nachricht. Diese sind die Metadaten einer Nachricht und können von Klienten zur Selektion der zu empfangenden Nachrichten verwendet werden. Der zweite Bereich zeigt den Inhalt der Nachricht. Bei einer EchoRequest-Nachricht ist der Inhalt leer, der Nachrichtentyp und der Adressat sind als Eigenschaften angegeben. Die EchoReply-Nachricht enthält als Nachrichteninhalt den Namen des Subjekts und den Zeitpunkt, zu dem die Nachricht versandt wurde. Der Nachrichtentyp ist als Eigenschaft angegeben.

EchoRequest
MessageType: „EchoRequest" Subject: „Dataimporter"

EchoReply
MessageType: „EchoReply"
Subject: „Dataimporter" Timestamp: 1308126597781

Abbildung 5.3.: EchoRequest- und EchoReply-Nachricht der Prozessüberwachung

Um zu verhindern, dass jeder JMS-Klient, der das Watchdog-Topic abonniert, alle Nachrichten empfängt, werden MessageSelector-Ausdrücke verwendet. Es handelt sich dabei um textuelle Ausdrücke, die bei der Verbindung zum JMS-Topic gesetzt werden können. Sie ermöglichen eine einfache Filterung der zu empfangenden Nachrichten für einen JMS-Klienten, wobei die Eigenschaften der Nachricht mit vorgegebenen Werten verglichen werden können. Alle Nachrichten, die nicht dem MessageSelector eines Klienten entsprechen, werden von diesem nicht empfangen.

In der Nachricht wird eine Eigenschaft gesetzt, die den Nachrichtentyp angibt. Die Subjekte verwenden als MessageSelector die Bedingung `MessageType='EchoRequest'`. Dadurch empfangen sie nur EchoRequest-Nachrichten. Beim Watchdog erfolgt dies entsprechend für die EchoReply-Nachrichten. Der vollständige MessageSelector für ein Subjekt ist in Listing 5.1 dargestellt.

```
1 MessageType='EchoRequest' AND Subject='Dataimporter'
```

Listing 5.1: Beispiel für den MessageSelector eines Subjekts

5.1.3. Datenbanküberwachung

Die Datenbank wird im Server durch eine Message-Driven-Bean repräsentiert. Diese empfängt, genau wie die anderen Subjekte, EchoRequest-Nachrichten vom JMS-Topic. Im Gegensatz zu den anderen Subjekten sendet sie nicht sofort eine Antwortnachricht, sondern überprüft vorher die Verfügbarkeit der Datenbank. Nur wenn diese Überprüfung erfolgreich verläuft, wird eine Antwort gesendet. Die Verfügbarkeit der Datenbank wird durch die Ausführung einer einfachen SQL-Abfrage geprüft. Diese wird in einer zusätzlichen MBean gespeichert und kann daher über eine XML-Datei konfiguriert werden. Die Konfiguration ist in Listing 5.2 dargestellt. In der Abfrage wird ein einfacher Wert einer Tabelle ausgelesen, die in jedem HyperTest-System vorhanden ist.

```
<mbean code="com.hypertest.monitoring.watchdog.server.DatabaseMonitoringConfig"
       name="hypertest:name=DatabaseMonitoringConfig">
    <attribute name="ValidationQuery">SELECT COUNT(*) FROM htcounter</attribute>
</mbean>
```

Listing 5.2: Konfiguration der Datenbank-Testabfrage in XML

5.1.4. Programmierschnittstelle der Subjekte

Die Subjekt-Programmierschnittstelle wurde so entworfen, dass sie möglichst einfach in den bestehenden Prozessen verwendet werden kann. Beim Start eines Prozesses wird ein *WatchdogSubjectImpl*-Objekt erzeugt und dessen `start`-Methode aufgerufen. Der Name des Subjekts wird bei der Erzeugung übergeben. Dieser muss in Server und Subjekt gleich sein, damit die Zuordnung korrekt funktioniert. In Listing 5.3 wird die Erzeugung für ein Subjekt mit dem internen Namen „SomeSubject“ gezeigt.

Intern verwendet die Klasse *WatchdogSubjectImpl* die Klasse *WatchdogTopicSubscriber*. Diese verwendet eine Implementierung der JMS-Standardschnittstelle *TopicSubscriber*, welche auf den Empfang einer Nachricht vom JMS-Topic wartet und beim Eintreffen einer Nachricht die `onMessage`-Methode des konfigurierten *MessageListener*-Objekts aufruft.

```
WatchdogSubject subject = new WatchdogSubjectImpl( "jnp://localhost:1099", "
    ConnectionFactory", "SomeSubject" );
subject.start();
```

Listing 5.3: Beispiel für die Erzeugung einer *WatchdogSubject*-Instanz

Bei der Beendigung eines Prozesses sollte das *WatchdogSubject* mit der `close`-Methode geschlossen werden, damit der intern verwendete *TopicSubscriber* korrekt vom JMS-Topic abgemeldet wird.

5.2. Statistikerhebung

5.2.1. Actions zur Erhebung von Daten

Zur Erhebung statistischer Daten verschiedener Kategorien werden Action-Klassen verwendet, welche die *StatisticsCollectorAction*-Schnittstelle implementieren. Die zu verwendende *StatisticsCollection* wird über eine entsprechende Setter-Methode gesetzt und kann daraufhin innerhalb der `execute`-Methode der Action verwendet werden. Der Kategoriename wird durch die `getCategoryName`-Methode zurückgegeben. Der Anwendungsprogrammierer muss sicherstellen, dass der Kategoriename systemweit eindeutig ist.

Da der *StatisticsCollector* als MBean realisiert wurde, kann er über eine XML-Datei konfiguriert werden. Die vollqualifizierten Klassennamen der *StatisticsCollectorAction*-Klassen werden als kommaseparierte Liste in der XML-Konfiguration der MBean angegeben. Der *StatisticsCollector* liest die Klassennamen beim Start des Anwendungsservers ein und erzeugt jeweils eine Instanz jeder Action-Klasse. Auf diese Weise ist das Hinzufügen neuer Actions einfach möglich: Sie müssen sich im Klassenpfad befinden und der Klassenname muss in der XML-Konfiguration angegeben werden. Außerdem muss jede Action-Klasse einen Standardkonstruktor besitzen.

Die Implementierung der *MemoryStatisticsAction* wird in Listing 5.4 gezeigt. Sie fügt in ihrer `execute`-Methode zwei Werte zur Statistiksammlung hinzu: Den aktuell belegten und maximal verfügbaren Hauptspeicher der JVM in Byte. Die zugehörige Kategorie trägt den internen Namen „Memory“.

```java
private StatisticsCollection statisticsCollection;

@Override
public synchronized void execute( )
{
  MemoryMXBean mxbean = ManagementFactory.getMemoryMXBean();
  MemoryUsage usage = mxbean.getHeapMemoryUsage();

  statisticsCollection.put( getCategoryName(),
                            Constants.KEY_MEMORY_USED, usage.getUsed() );
  statisticsCollection.put( getCategoryName(),
                            Constants.KEY_MEMORY_MAX, usage.getMax() );
}

@Override
public String getCategoryName( )
{
  return "Memory";
}
```

```
20
@Override
22 public void setStatisticsCollection( StatisticsCollection collection )
{
24   this.statisticsCollection = collection;
}
```

Listing 5.4: Implementierung der MemoryStatisticsAction

5.2.2. Zyklische Erhebung und Speicherung von Daten

Die Erhebung und Speicherung der statistischen Daten für eine Kategorie wird durch den Ablauf eines Timers in der *StatisticsServiceBean* veranlasst. Dieser Mechanismus ähnelt stark dem der Prozessüberwachung. Allerdings enthält das *TimerInfo*-Objekt, welches dem Timer übergeben wird, hier den Kategorienamen. Bei Ablauf eines Timers wird der Kategoriename ermittelt und die `collect`-Methode des *StatisticsCollectors* aufgerufen.

Für jede Kategorie wird während der Initialisierung ein leerer Statistik-Datensatz erzeugt. Handelt es sich um eine Kategorie, für die Operationenstatistiken erhoben werden, so wird außerdem eine Liste für Operationenstatistik-Datensätze vorbereitet. Der Datensatz erhält während der Initialisierung den Kategorienamen und beim Start der Erhebung die Startzeit des Intervalls. Innerhalb der `collect`-Methode wird er abgefragt und durch den Aufruf einer Action mit Informationen gefüllt. Aus den Daten des Datensatzes wird eine persistente Entity-Bean erzeugt. Anschließend wird er geleert und mit der Startzeit des nächsten Erhebungsintervalls versehen. Der vollständige Ablauf der `collect`-Methode ist in Abbildung 5.4 dargestellt.

Ist eine Methode in der Methodenaufrufstatistik erfasst, wird jedoch innerhalb eines Erhebungsintervalls nicht aufgerufen, so wird dennoch ein Datensatz für die Methode in der Datenbank erzeugt. Die Werte des Datensatzes (Anzahl der Aufrufe, minimale, maximale und durchschnittliche Ausführungsdauer) sind in diesem Fall 0. Ansonsten könnte nicht unterschieden werden, ob die Methode während des Erhebungsintervalls nicht in der Statistik erfasst wurde oder ob sie zwar in der Statistik erfasst, jedoch nicht aufgerufen wurde.

5.2.3. Aspekt zur Erhebung von Methodenaufrufstatistiken

Die Erhebung von Methodenaufrufstatistiken wird mithilfe eines Aspekts gelöst. In Abschnitt 2.4 wurden bereits einige AOP-Realisierungen vorgestellt. Für die Umsetzung muss ein AOP-Framework ausgewählt werden.

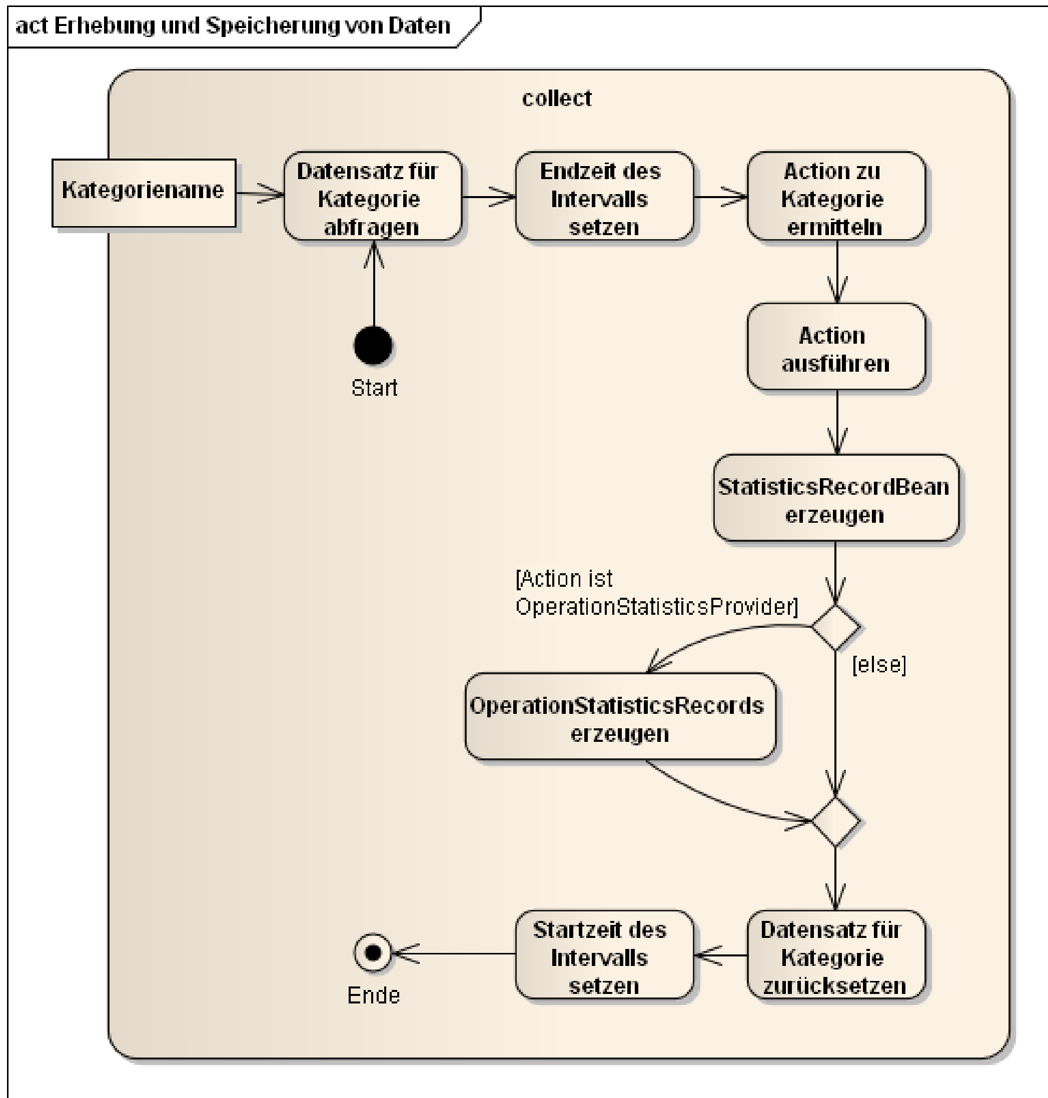

Abbildung 5.4.: Erhebung und Speicherung von Daten für eine Statistik-Kategorie

Da Spring AOP zur Verwendung zusammen mit dem Spring Dependency-Injection-Container vorgesehen ist und weit weniger Funktionalität als die anderen AOP-Frameworks bietet, wird es nicht weiter betrachtet. AspectJ und JBoss AOP unterscheiden sich bezüglich ihres Funktionsumfangs laut Kersten (2005a) und Kersten (2005b) kaum voneinander. Ein wichtiges Kriterium ist jedoch die Portierbarkeit der Realisierung auf andere Anwendungsserver. Zwar erhebt JBoss AOP den Anspruch, für beliebige Java-Anwendungen verwendbar zu sein, jedoch scheint es beim Einsatz mit anderen Anwendungsservern Probleme zu verursachen[1]. Ein Vorteil von AspectJ besteht außerdem darin, dass es von anderen Frameworks, wie z. B. Spring, unterstützt wird (vgl. Johnson u. a. 2008).

Da keine Abhängigkeiten von anderen Komponenten zu dem verwendeten Aspekt bestehen, könnte dieser einfach ausgetauscht werden, falls sich ein anderes AOP-Framework oder eine Lösung mit Interzeptoren in der Zukunft als vorteilhafter erweist. Durch die Auslieferung als eigenständiges jar-Archiv wird dies zusätzlich vereinfacht.

AspectJ bietet die Möglichkeit, Aspekte mithilfe einer Spracherweiterung zu definieren oder Annotationen zu verwenden. Hier werden Annotation verwendet, da sie jedem Java-Entwickler bekannt sind. Aspekte können als Singleton-Instanzen, pro Klasse oder sogar pro Kontrollfluss erzeugt werden. Dies hat Auswirkungen auf das Verhalten des Aspekts, insbesondere in Bezug auf gespeicherte Felder.

Es wird eine Aspekt-Instanz für jede Klasse innerhalb des Servers verwendet. Die Annotation in Listing 5.5 legt fest, dass für jede Klasse innerhalb des angegebenen Pakets eine Aspekt-Instanz erzeugt wird. So ist es möglich, jeden Aspekt einmalig zu initialisieren und den Klassennamen der instrumentierten Klasse innerhalb des Aspekts als Feld zu speichern. Da der Anwendungsserver Proxyobjekte von Bean-Klassen generiert, um Dienste zu ihren Methoden hinzuzufügen, muss der ursprüngliche Klassenname der Bean zunächst durch den Aspekt extrahiert und gespeichert werden. Dies wird aus Gründen der Performanz nur ein einziges Mal nach der Erzeugung der Bean durchgeführt.

```
1 @Aspect( "pertypewithin( com.hypertest.server..* )" )
```

Listing 5.5: Definition der Aspekt-Multiplizität

Die Annotation zur Definition des Pointcuts ist in Listing 5.6 zu sehen. Durch den `execution`-Joinpoint werden die Ausführungen aller Methoden (mit beliebigen Parameterlisten und Rückgabewerten) ausgewählt. Zusätzlich wird mit `this` die Menge der Aufrufe auf alle Enterprise-Bean-Aufrufe eingeschränkt. Dies schließt Unterklassen mit ein, erfasst also alle Aufrufe auf Entity- und Session-Beans.

[1] Vgl. http://community.jboss.org/message/279826, http://community.jboss.org/thread/88384. Zugriffsdatum: 19.07.2011

```
1 @Pointcut( "execution(* *(..)) && this(javax.ejb.EnterpriseBean)" )
```

Listing 5.6: Definition eines Pointcuts für Bean-Methoden

Die verwendete Advice-Methode ist in Listing 5.7 dargestellt. Der Joinpoint wird von AspectJ als Parameter an die Advice-Methode übergeben und enthält reflektive Informationen über den tatsächlichen Aufruf. Er wird dazu verwendet, die Parameter und den Methodennamen zu ermitteln. Der Klassenname wurde schon während der Initialisierung des Aspekts ermittelt und ist in `simpleClassName` abgelegt. Die Bean-Methode wird mithilfe der `proceed`-Methode aufgerufen und ihre Ausführungszeit gemessen. Anschließend werden die Parameter und der Methodennamen ermittelt und die Werte an den *MethodStatisticsCollector* übergeben. Dieser prüft intern, ob er die Ausführungszeit des Aufrufs speichern muss oder nicht.

```
1  @Around( " beanCall( ) " )
   public Object captureMethodInvocation( ProceedingJoinPoint thisJoinPoint )
3    throws Throwable
   {
5    long startTime = System.nanoTime();
     Object returnValue = thisJoinPoint.proceed();
7    long elapsed = System.nanoTime() - startTime;

9    Object[] arguments = thisJoinPoint.getArgs();
     MethodSignature signature = ( MethodSignature ) thisJoinPoint.getSignature();
11   String methodName = signature.getMethod().getName();

13   methodStatsCollector.capture( simpleClassName, methodName, arguments,
         returnValue, elapsed );

15   return returnValue;
   }
```

Listing 5.7: Around-Advice zur Erfassung von Methodenaufrufen

5.2.4. Syntax der Filter-Ausdrücke

Es besteht die Möglichkeit, Filterausdrücke für Parameter und Rückgabewerte von Methoden zu definieren. Diese werden zunächst textuell angegeben. In der Datenbank werden sie innerhalb einer *OperationMeta*-Entität gespeichert.

Um *Filter*-Hierarchien aus einer textuellen Repräsentation zu erzeugen, wurde ein einfacher Parser mit ANTLR (Parr 2010) realisiert. Die Syntax ist in Abbildung 5.5 dargestellt. Die Sprache für die Ausdrücke verwendet eine Präfix-Notation. Die Parameter für einen Operator

werden in Klammern, getrennt durch Kommata, angegeben. Zur Auswahl eines Parameters wird der Index innerhalb der Parameterliste verwendet, wobei der erste Parameter den Index 0 besitzt. Für den Rückgabewert existiert ein eigenes Schlüsselwort `return`.

Die vollständige ANTLR-Grammatik und die Verwendung der generierten Klassen innerhalb der Monitoring-Komponente ist in Anhang B beschrieben.

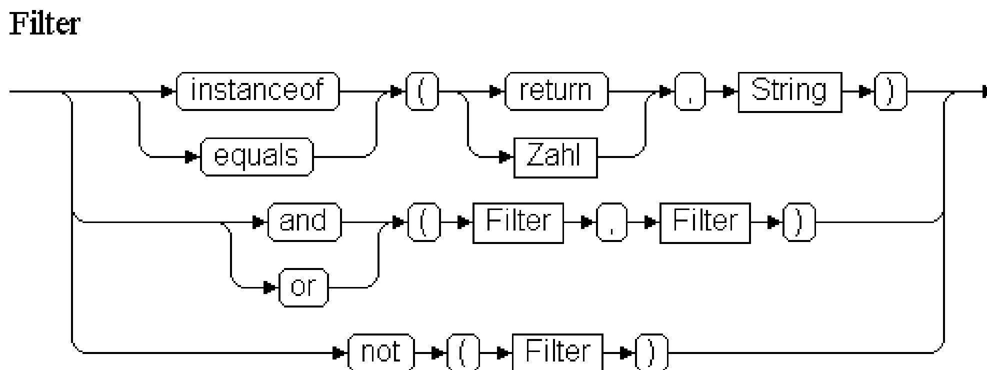

Abbildung 5.5.: Syntax-Diagramm für einfache Methodenfilter

Für einfache Filter, wie den *InstanceOfFilter* und den *EqualsFilter*, wurde eine abstrakte Basisklasse realisiert, welche den zu vergleichenden String und den Index des zu nutzenden Parameters im Konstruktor übergeben bekommt. Sie implementiert die *Filter*-Schnittstelle. Allerdings wird in der `apply`-Methode lediglich das zu vergleichende Objekt ermittelt und an eine abstrakte `applyInternal`-Methode übergeben. Konkrete Unterklassen überschreiben diese Methode. Dadurch befindet sich die Logik zur Auswahl des korrekten Parameters an einer einzigen Stelle und muss nicht in jeder konkreten Filter-Klasse erneut implementiert werden.

5.3. Grafische Oberfläche

Die grafische Oberfläche wurde gemäß des Konzeptes aus Kapitel 4 umgesetzt. Für die Realisierung der Grafiken wurde das Google Chart Tools API verwendet, eine zusätzliche Bibliothek mit der JavaScript-basierte Grafiken aus GWT heraus erzeugt werden können (vgl. Google 2011b). Ext Gwt bietet ebenfalls die Möglichkeit, Grafiken zu erzeugen, diese basieren jedoch auf Adobe Flash, was eine zusätzliche Webbrowser-Erweiterung benötigt. Ein Screenshot der Ansicht zur Anzeige von Methodenaufrufstatistiken mit einer erzeugten

Grafik ist in Abbildung 5.6 zu sehen. Weitere Screenshots der grafischen Oberfläche befinden sich auf der beiliegenden CD.

Eine Grafik wird mithilfe eines *DataTable*-Objekts erzeugt. Dieses repräsentiert eine einfache Tabelle, in der zuerst alle Spaltennamen und -typen gesetzt werden. Anschließend werden Zellen über ihre Zeilen- und Spalten-Indizes adressiert und mit einem Wert versehen. Die Umwandlung eines *StatisticsRecord* in einen solchen *DataTable* wird auf der Klientenseite mithilfe einer Utility-Klasse durchgeführt. Bisher existiert ausschließlich für allgemeine Kategorien eine generische Lösung. Die Umwandlung eines Methodenstatistik-Datensatzes ist statisch umgesetzt.

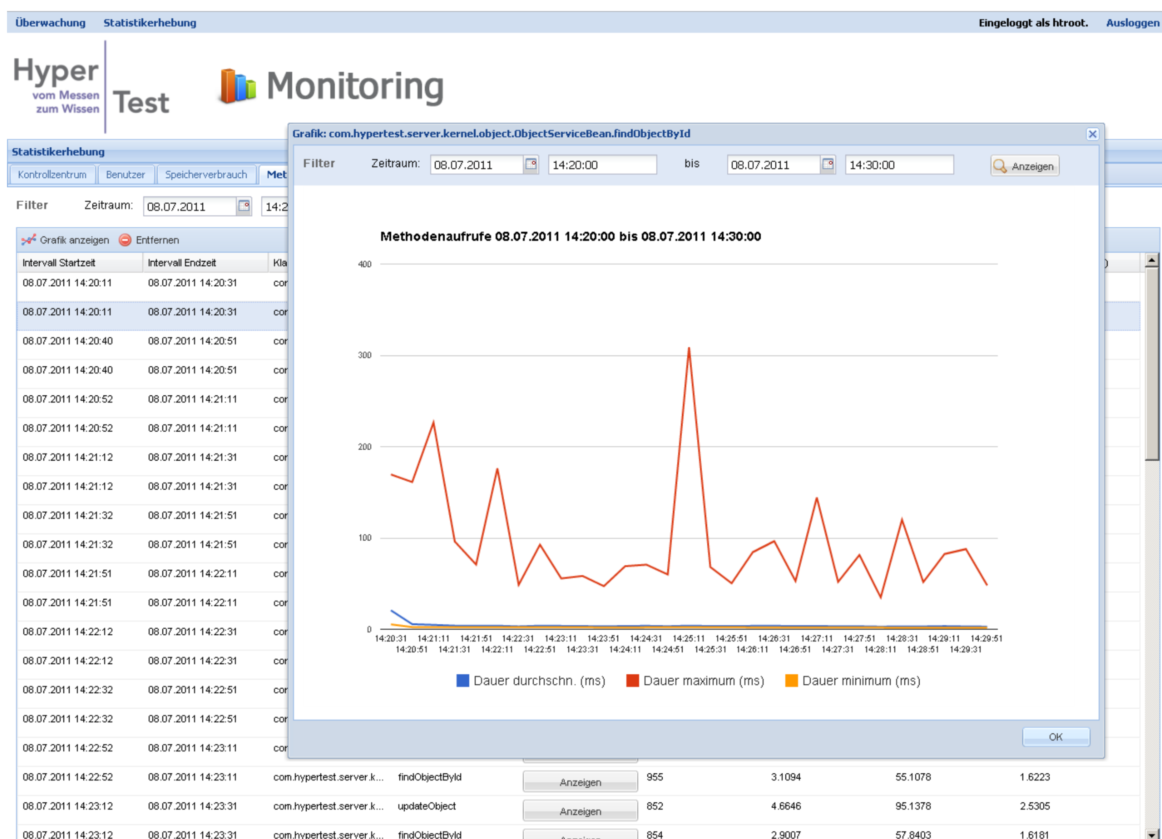

Abbildung 5.6.: Screenshot: Ansicht zur Anzeige von statistischen Daten

Die Ansicht zur Konfiguration der Statistikerhebung und zur Anzeige der erhobenen Daten ist in mehrere Tabs aufgeteilt: Es existiert je ein Tab pro Statistik-Kategorie, der eine Ansicht zur Anzeige der Daten enthält. Außerdem existiert ein Kontrollzentrum-Tab, der eine Ansicht enthält, auf der alle Statistik-Kategorien konfiguriert werden können. Während der Initialisierung ermittelt die grafische Oberfläche über einen asynchronen Service die vorhandenen Statistik-Kategorien. Für alle allgemeinen Kategorien werden die Ansichten, d. h. Tabs

und Einträge im Kontrollzentrum-Tab, generisch erzeugt. Ansichten für Operationenstatistiken müssen zurzeit manuell hinzugefügt werden. Der Grund dafür besteht darin, dass sich die Konfiguration und die anzuzeigenden Daten bei verschiedenen Operationen-Typen stark unterscheiden können. Der Ablauf der generischen Erzeugung der Ansichten für allgemeine Kategorien ist in Abbildung 5.7 dargestellt.

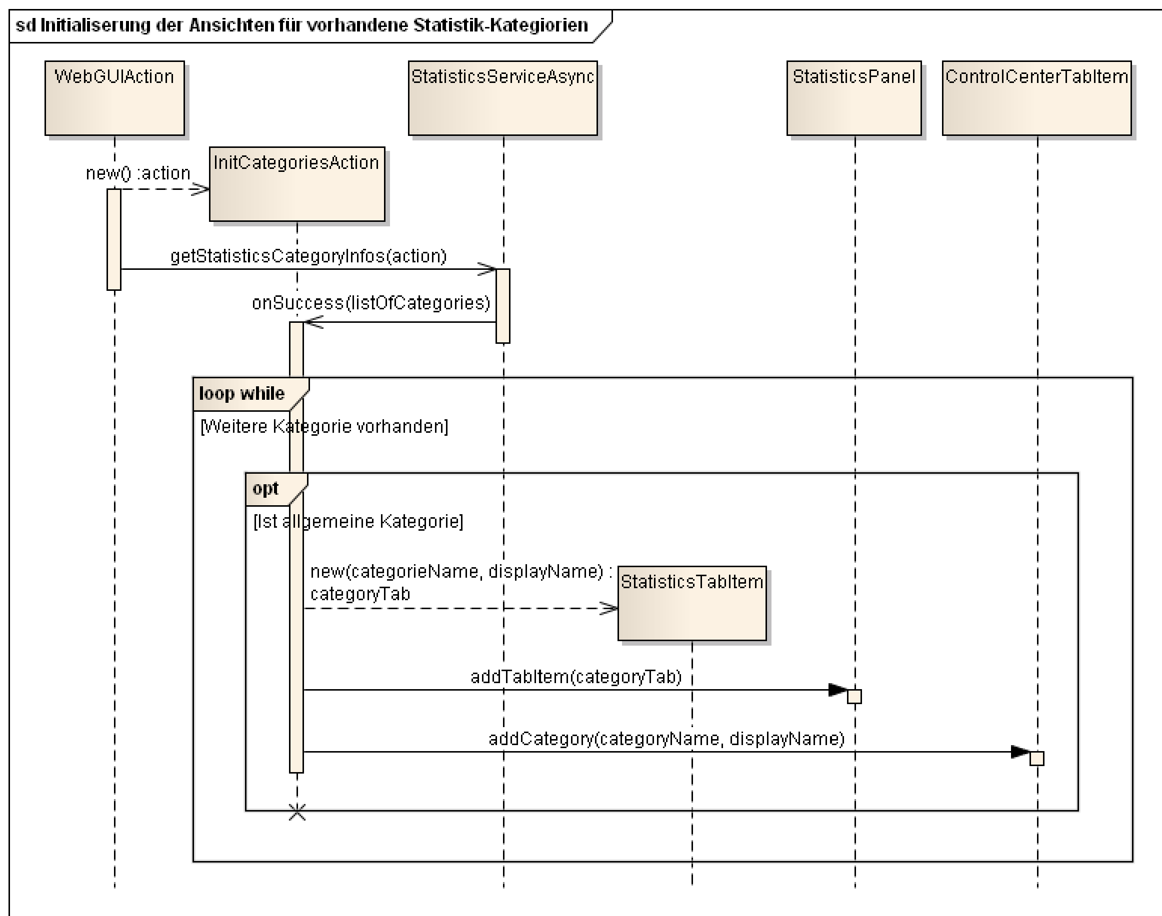

Abbildung 5.7.: Initialiserung der Ansichten für vorhandene Statistik-Kategiorien auf der GUI

6. Messung des Zeitverhaltens

In diesem Abschnitt wird das Zeitverhalten des HyperTest-Systems mit und ohne Statistikerhebung gemessen, um die Auswirkung der Statistikerhebung auf die Performanz des Systems zu ermitteln. Dazu wird die Ausführungszeit von ausgewählten Operationen zunächst im Ausgangssystem und anschließend unter Verwendung der im Rahmen dieser Arbeit realisierten Erweiterungen gemessen.

Aussagekräftige Performanzmessungen durchzuführen ist ein komplexes Thema, welches eigene wissenschaftliche Arbeiten füllt. Daher erheben die hier durchgeführten Messungen nicht den Anspruch, absolut exakt zu sein. Die ermittelte prozentuale Abweichung der beiden Messungen soll lediglich einen Anhaltspunkt dafür geben, wie stark sich die Statistikerhebung auf die Performanz des Gesamtsystems auswirkt.

6.1. Testumgebung und Durchführung

Da es bei dieser Messung darum geht, den Performanzunterschied einiger Operationen mit und ohne Statistikerhebung zu ermitteln, wird ein HyperTest-System mit einer zunächst leeren Datenbank verwendet. Lediglich unbedingt notwendige Daten, wie Benutzerkennungen und Objekttypen werden eingefügt. Man könnte die Datenbank vorher mit einer Menge von Daten füllen, wodurch die einzelnen Operationen mehr Zeit beanspruchen würden. Jedoch sollte dies die prozentuale Differenz der gemessenen Ausführungszeiten von Operationen mit und ohne Statistikerhebung nicht beeinflussen.

Für die Messung wird ein einfacher Simulator entwickelt. Dieser legt während der Initialisierung zunächst einen Container (einen einfachen Ordner) im System an und startet im Anschluss eine festgelegte Menge an Klienten. Jeder Klient baut eine eigene Verbindung zum Server auf und legt ein einfaches Testobjekt in dem zuvor erzeugten Container an. Das Testobjekt ist eine Textdatei mit einfachen Metadaten, wie z. B. einem Datum und einer Beschreibung. Anschließend führt jeder Klient hundertmal folgende Operationen aus:

- Laden des angelegten Testobjekts
- Schreiben einer zufällig generierten Zeichenkette in die Metadaten des Objekts

- Aktualisierung des veränderten Objekts im Server

Die beiden Operationen (Objekt lesen, Objekt aktualisieren) wurden stellvertretend für die Gesamtmenge aller Operationen innerhalb eines HyperTest-Systems ausgewählt, da sie in einem realen HyperTest-System häufig ausgeführt werden. Welche konkreten Operationen betrachtet werden, ist jedoch irrelevant, da der Mechanismus zur Ermittlung von Aufrufstatistiken unabhängig von der jeweiligen Operation ist.

Die Anzahl der zu verwendenden Klienten kann beim Start des Simulators angegeben werden. Der Test wird zunächst ohne und anschließend mit aktiver Statistikerhebung mit jeweils 25, 10 und einem Klienten durchgeführt. Jede Messung wird dabei zehnmal wiederholt. Für jeden Klienten werden die durchschnittlichen Ausführungszeiten für die Operationen gebildet. Aus den Werten aller Klienten werden die Mittelwerte errechnet. Dadurch erhält man für jeden Testlauf und jede Operation einen Wert. Zuletzt werden die Durchschnittswerte aller Testläufe nochmals gemittelt.

Vor Beginn der Messung wird der Ablauf einmal mit 10 Klienten durchgeführt und das Ergebnis verworfen. So soll verhindert werden, dass eventuelle Optimierungen, die die JVM nach dem Start des Systems am Bytecode vornimmt, das Ergebnis verfälschen. Außerdem werden die benötigten Datenbankverbindungen aufgebaut und in einem Verbindungspool abgelegt.

Der Computer, auf dem alle Messungen durchgeführt werden, verwendet eine Intel CPU mit 2x 2,66 GHz Taktfrequenz und 4 GB Hauptspeicher. Bei Messungen mit aktiver Statistikerhebung werden die Benutzer-, Hauptspeicher- und Methodenaufrufstatistik mit einem Erhebungsintervall von 20 Sekunden aktiviert. Für die Methodenaufrufstatistik werden zwei Fälle unterschieden:

- Die Methoden, deren Ausführungszeiten gemessen werden, sind nicht in der Statistik erfasst. Der verwendete Aspekt prüft, ob ein Methodenaufruf erfasst werden muss. Diese Überprüfung fällt negativ aus und das Programm fährt fort.

- Die Methoden, deren Ausführungszeiten gemessen werden, sind selbst in der Statistik erfasst. Die ermittelte Ausführungszeit muss in diesem Fall also gespeichert werden.

6.2. Auswertung der Ergebnisse

In Tabelle 6.1 sind die durchschnittlichen Ausführungszeiten für die Operationen ohne und mit Statistikerhebung gegenübergestellt. Hier sind die Methoden selbst nicht in der Statistik erfasst.

Klienten	Operation	Ø ms ohne Statistikerhebung	Ø ms mit Statistikerhebung	Abweichung (%)
1	Lesen	1,8865	2,1403	13,45
1	Aktualisieren	3,9851	4,2370	6,32
10	Lesen	6,6843	7,7973	16,65
10	Aktualisieren	14,4312	15,6424	8,39
25	Lesen	12,2711	13,2841	8,25
25	Aktualisieren	35,4546	40,8608	15,25
Durchschnittliche Abweichung (%)				**11,39**

Tabelle 6.1.: Messergebnisse: Performanz des HyperTest-Systems ohne und mit Statistikerhebung (Methoden nicht statistisch erfasst)

Tabelle 6.2 zeigt die Messergebnisse ohne und mit Statistikerhebung, wobei die Methoden selbst in der Statistik erfasst sind. Im Vergleich zur ersten Messung müssten die Ausführungszeiten der Methoden bei aktiver Statistikerhebung um die Zeitspanne größer sein, die es benötigt einen Wert zu speichern.

Klienten	Operation	Ø ms ohne Statistikerhebung	Ø ms mit Statistikerhebung	Abweichung (%)
1	Lesen	1,8865	2,0642	9,41
1	Aktualisieren	3,9851	4,2116	5,68
10	Lesen	6,6843	7,8957	18,12
10	Aktualisieren	14,4312	16,0192	11,00
25	Lesen	12,2711	12,8341	4,59
25	Aktualisieren	35,4546	40,3102	13,70
Durchschnittliche Abweichung (%)				**10,42**

Tabelle 6.2.: Messergebnisse: Performanz des HyperTest-Systems ohne und mit Statistikerhebung (Methoden statistisch erfasst)

Der Unterschied zwischen Methoden, die in der Statistikerhebung erfasst sind und solchen, die es nicht sind, fällt äußerst gering aus. Dies ist vermutlich darauf zurückzuführen, dass das Speichern eines Wertes sich während der Erhebung darauf beschränkt, einen Wert in eine Datenstruktur einzufügen. Die Abweichung der in der Statistik erfassten Methoden fällt hier sogar geringer aus als die der nicht erfassten Methoden. Dies ist jedoch Zufall. In einer zweiten durchgeführten Messung war die Abweichung der erfassten Methoden größer als die der nicht erfassten Methoden.

Die Ergebnisse der Messung zeigen, dass die durchschnittliche Abweichung durch die aktive Statistikerhebung durchschnittlich 10-12% beträgt.

7. Zusammenfassung

In diesem Kapitel werden eine kurze Zusammenfassung dieser Arbeit und ein Ausblick auf mögliche Verbesserungen und Erweiterungen der entwickelten Lösung gegeben.

7.1. Fazit

In dieser Arbeit wurde eine Monitoring-Infrastruktur für eine komplexe verteilte Unternehmensanwendung entwickelt. Das Monitoring sollte dabei sowohl die Überwachung verteilter Prozesse als auch die Erhebung von statistischen Daten innerhalb des zentralen Servers ermöglichen. Die beiden Funktionen sollten unabhängig voneinander benutzbar sein und das bestehende System nicht allzu stark verlangsamen. Es war also eine Lösung gefordert, die sich transparent zum bestehenden System hinzufügen ließ. Neben den fachlichen Anforderungen wurden weitere Designziele, wie Portabilität, Einfachheit und Erweiterbarkeit verfolgt und waren im Rahmen der Konzeption eine wichtige Entscheidungsgrundlage.

Die Monitoring-Komponente wurde in zwei Subkomponenten aufgeteilt: Eine Watchdog-Subkomponente für die Prozessüberwachung und eine Statistics-Subkomponente für die Statistikerhebung. Beide Subkomponenten sind unabhängig voneinander benutzbar.

Die Prozessüberwachung wurde mithilfe von JMS entwickelt. Der Server sendet dabei zyklisch Nachrichten an die zu überwachenden Prozesse. Diese senden eine Antwortnachricht mit ihrem Namen und einem Zeitstempel. Antwortet eine Komponente über eine festgelegte Zeitspanne nicht, so wird angenommen, dass sie nicht ansprechbar ist. Durch die Verwendung von JMS wurde Portabilität auf andere Anwendungsserver erreicht. Ebenso wurden Spezialfälle minimiert und die Konfiguration im Server zentralisiert. Zukünftig hinzukommende verteilte Prozesse können einfach in die Überwachung aufgenommen werden.

Die Statistikerhebung findet vollständig innerhalb des Servers statt. Es wurde eine zentrale Instanz eingeführt, welche die Erhebung von statistischen Daten leistet. Die konkrete Ermittlung von Daten wurde in Action-Klassen gekapselt. Anwendungsentwickler können so auf einfache Weise weitere Action-Klassen zur Erhebung zusätzlicher Informationen hinzufügen. Die erhobenen Werte werden dabei genau einer Kategorie zugeordnet. Es wurde ein allgemeines Datenmodell entwickelt, bei dem statistische Daten in Schlüssel-Wert-Paaren

abgelegt werden. Dadurch können auch auf Ebene des Datenmodells auf einfache Weise weitere Daten hinzugefügt werden.

Bei den erhobenen Statistiken wird zwischen allgemeinen Statistiken und Operationenstatistiken unterschieden. Allgemeine Statistiken enthalten pro Zeitintervall nur einfache Werte, während Operationenstatistiken pro Zeitintervall viele zusammengesetzte Werte enthalten können. Für die Erhebung von Methodenaufrufstatistiken wurde ein eigener Mechanismus entwickelt. Mithilfe von aspektorientierter Programmierung werden Informationen über Aufrufe erhoben und an die Statistiksammlung übergeben. Die Ausführungszeiten der Methoden werden für jedes Zeitintervall aggregiert und so die durchschnittliche, minimale und maximale Dauer einer Methode in einem Zeitintervall ermittelt.

Für die Visualisierung wurde eine einfache Weboberfläche auf Basis von GWT entwickelt. Diese ermöglicht das Konfigurieren der Monitoring-Komponente und die Abfrage aller ermittelten Daten. Sie wurde so gestaltet, dass Ansichten für neu hinzukommende allgemeine Statistik-Kategorien generisch erzeugt werden. Für Operationenstatistiken müssen neue Ansichten zur Konfiguration und Anzeige der Daten zurzeit manuell hinzugefügt werden.

Abschließend wurde eine Performanzmessung mit und ohne aktive Statistikerhebung durchgeführt. Dabei wurde festgestellt, dass die Statistikerhebung die Ausführung von Operationen im Server um ca. 10-12% verlangsamt. Dieser Wert ist für eine erste Implementierung ohne Optimierungen sehr akzeptabel.

7.2. Ausblick

Die hier entwickelte Infrastruktur bildet eine Basis für ein Monitoring-System. Viele Aspekte können noch erweitert und verbessert werden.

Die Prozessüberwachung erfasst alle vorhandenen Prozesse, einschließlich der Datenbank. Zukünftige Java-Prozesse können ebenfalls in die Überwachung aufgenommen werden. Allerdings werden die Zustände der Prozesse zurzeit nicht persistent gespeichert. Eine Historie ist folglich nicht vorhanden. Dies könnte dahingehend erweitert werden, dass alle Zustandsänderungen gespeichert werden. Zusätzlich könnte ein Administrator benachrichtigt werden, wenn ein Prozess ausfällt. Dies könnte z. B. per Mail oder über einen Instant Messenger geschehen.

Für die Statistikerhebung wurde eine Infrastruktur geschaffen. Drei Statistik-Kategorien wurden im Rahmen dieser Arbeit realisiert. Während Benutzer- und Hauptspeicherstatistik durchaus zufriedenstellende Informationen liefern, ließe sich die Methodenaufrufstatistik noch optimieren.

Zurzeit werden Methoden, die aus einer in der Statistik erfassten Methode aufgerufen werden, nicht automatisch erfasst. Hier besteht eine denkbare Erweiterung darin, den Kontrollfluss zu analysieren und ganze Aufrufketten statistisch zu erfassen. So ließe sich einfacher feststellen, welcher Teil einer Methode den größten Zeitanteil verbraucht. Dafür müsste das Datenmodell um hierarchisch gegliederte Datensätze erweitert werden. Es müsste außerdem geprüft werden, wie sich dies auf die Performanz der Statistikerhebung auswirkt.

Statistiken werden aktuell nur innerhalb des Servers erhoben. Es wäre möglich, dass verteilt laufende Prozesse ebenfalls statistische Daten an den Server übermitteln. Auch hier könnte JMS verwendet werden. Dabei gibt es jedoch einige Probleme: Durch das asynchrone Verhalten von JMS können Daten von Subjekten nicht unmittelbar abgefragt werden. Stattdessen würden die Daten zu einem unbestimmten Zeitpunkt im Server empfangen und gespeichert werden. Es müsste ein Konzept entwickelt werden, um damit sinnvoll umzugehen.

Auch bei der Statistikerhebung könnten Administratoren in bestimmten Fällen, z. B. dem Erreichen vorher konfigurierter Grenzwerte, benachrichtigt werden.

Die grafische Oberfläche kann in vielen Punkten erweitert werden. Bei der Konfiguration von Methoden, die in der Methodenaufrufstatistik erfasst werden sollen, müssen Klassen- und Methodennamen textuell eingegeben werden. Sie müssen dem Benutzer also mit vollständigem Namen bekannt sein. Das System könnte die möglichen zu überwachenden Methoden ermitteln und sie in einem Auswahlfeld zur Verfügung stellen. Dies würde die Benutzung stark vereinfachen. Ebenso werden Filter nicht gespeichert. Es wäre möglich, Filter unter einem eindeutigen Namen zu speichern, sodass häufig verwendete Filter nicht mehrfach eingegeben werden müssen. Auch könnte die textuelle Eingabe durch einen benutzerfreundlichen, evtl. grafischen Filter-Editor ersetzt werden.

A. Layout der grafischen Oberfläche

In diesem Abschnitt wird das Layout der grafischen Oberfläche entworfen. Dafür werden die benötigten Ansichten beschrieben und mithilfe von GUI-Mockups[1] visualisiert.

Die grafische Oberfläche soll die Konfiguration des Monitorings und die Abfrage von vorhandenen Daten ermöglichen. Da das Monitoring aus zwei Subkomponenten, der Prozessüberwachung und der Statistikerhebung, besteht, sollte auch die Oberfläche zwei Bereiche aufweisen, von denen stets nur einer zur Zeit angezeigt wird.

Der Bereich der Prozessüberwachung besteht aus nur einer Ansicht. Es müssen Subjekte hinzugefügt, bearbeitet und gelöscht werden können. Außerdem kann die Überwachung gestartet und gestoppt werden. Darüber hinaus sollte der Zustand aller aktuell konfigurierten Subjekte sichtbar sein. Diese Informationen können alle innerhalb einer Tabelle dargestellt werden. Beim Hinzufügen und Bearbeiten von Subjekten öffnet sich ein einfaches Dialogfeld. Der Entwurf für das Layout ist in Abbildung A.1 zu sehen.

Weitaus komplexer ist die Ansicht für die Statistikerhebung. Hier muss zwischen allgemeinen Statistiken und Operationenstatistiken unterschieden werden. Die Konfiguration und die Anzeige der Daten kann nicht innerhalb einer Tabelle dargestellt werden, da hier die aktuelle Erhebung keinen Bezug zu den angezeigten Daten haben muss. Der Benutzer kann aktuell Daten für die Benutzerstatistik erheben, sich jedoch ältere Daten anzeigen lassen. Konfiguration und Anzeige müssen hier also getrennt werden. Zum Umschalten zwischen einzelnen Ansichten werden Tabs verwendet. Für die Konfiguration wird eine Kontrollzentrum-Ansicht eingeführt, die alle Kategorien und ihre aktuelle Konfiguration anzeigt. Die Konfiguration unterscheidet sich, je nachdem welcher Art die erhobenen Daten sind. Die Kontrollzentrum-Ansicht ist in Abbildung A.2 dargestellt.

Für allgemeine Statistiken kann eine generische Ansicht erstellt werden, die für alle Kategorien verwendet werden kann. Hier werden eine Tabelle mit den statistischen Daten und eine grafische Darstellung nebeneinander dargestellt. Außerdem soll es möglich sein die Anzeige der Daten auf einen Zeitraum einzuschränken, in dem die Daten erhoben wurden. Dies ist beispielhaft für die Benutzerstatistik in Abbildung A.3 dargestellt.

[1] GUI-Mockups sind einfache Entwürfe einer grafischen Oberfläche. Dies können sowohl einfache Skizzen als auch teilweise funktionsfähige grafische Oberflächen sein, die zu Demonstrationszwecken verwendet werden.

Prozessüberwachung | Statistikerhebung | Ausloggen

HyperTest Monitoring

Überwachung

Status: Aktiv | Deaktivieren

Hinzufügen | Bearbeiten | Löschen

Prozess	Verfügbarkeit	Letzte Antwort
Datenbank Postgres	verfügbar	17.06.2011 10:30:21
ToolStarter	verfügbar	17.06.2011 10:31:10
DatenImporter (Aufbereitung)	verfügbar	17.06.2011 10:30:56
DatenImporter (Durchführung)	nicht verfügbar	17.06.2011 10:01:02

Abbildung A.1.: GUI-Mockup: Ansicht für Prozess-Überwachung

Während für allgemeine Statistiken eine generische Lösung gefunden werden kann, ist dies im Falle der Operationenstatistiken schwer zu realisieren. Operationenstatistiken enthalten Metadaten, die sich je nach Art der Daten unterscheiden können. Außerdem enthalten sie komplexere Daten. Im Rahmen dieser Arbeit wird die Methodenaufrufstatistik als einzige Operationenstatistik-Kategorie realisiert. Die Daten dieser Kategorie werden in einer eigenen Ansicht dargestellt. In der Ansicht für Operationenstatistiken können Tabelle und Grafik aufgrund ihrer Größe nicht nebeneinander dargestellt werden. Daher wird zunächst nur die Tabelle angezeigt. Die Grafik kann in einem neuen Fenster geöffnet werden, indem ein Eintrag der Tabelle ausgewählt und der Button „Grafik anzeigen“ angeklickt wird. Die Grafik zeigt dann eine Kurve für die aktuell gewählte Methode und den aktuell ausgewählten Zeitraum. Die Methodenaufrufstatistik-Ansicht ist in Abbildung A.4 zu sehen.

Prozessüberwachung | Statistikerhebung | Ausloggen

HyperTest Monitoring

Statistikerhebung

Kontrollzentrum | Benutzer | Speicherverbrauch | Methodenaufrufe

Benutzer

Status: aktiv Zykluszeit: 10000 Zykluszeit: 10000 Setzen Aktivieren

Speicherverbrauch

Status: aktiv Zykluszeit: 10000 Zykluszeit: 10000 Setzen Aktivieren

Methodenaufrufe

Status: inaktiv Zykluszeit: 10000 Zykluszeit: 10000 Setzen Aktivieren

Hinzufügen | Bearbeiten | Entfernen

Klassenname ▲	Methodenname	Filter
com.test.Foo	findSomething	equals(return, "Foobar")
com.test.Bar	updateSomething	-

Abbildung A.2.: GUI-Mockup: Kontrollzentrum für die Statistikerhebung

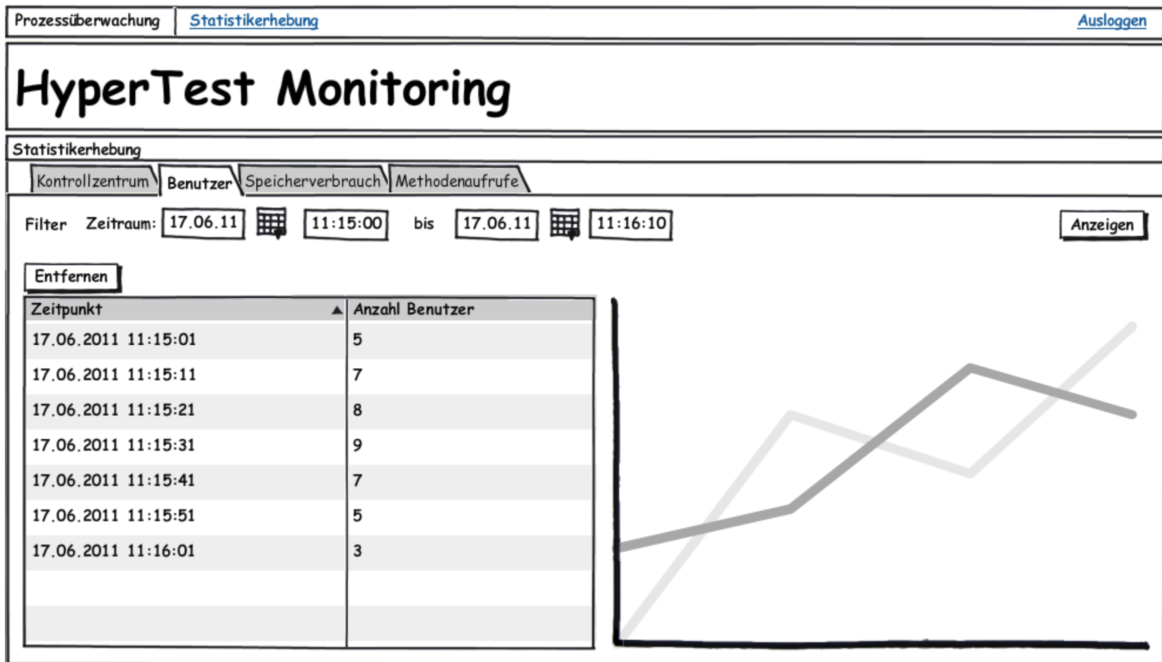

Abbildung A.3.: GUI-Mockup: Ansicht zur Anzeige allgemeiner Statistiken

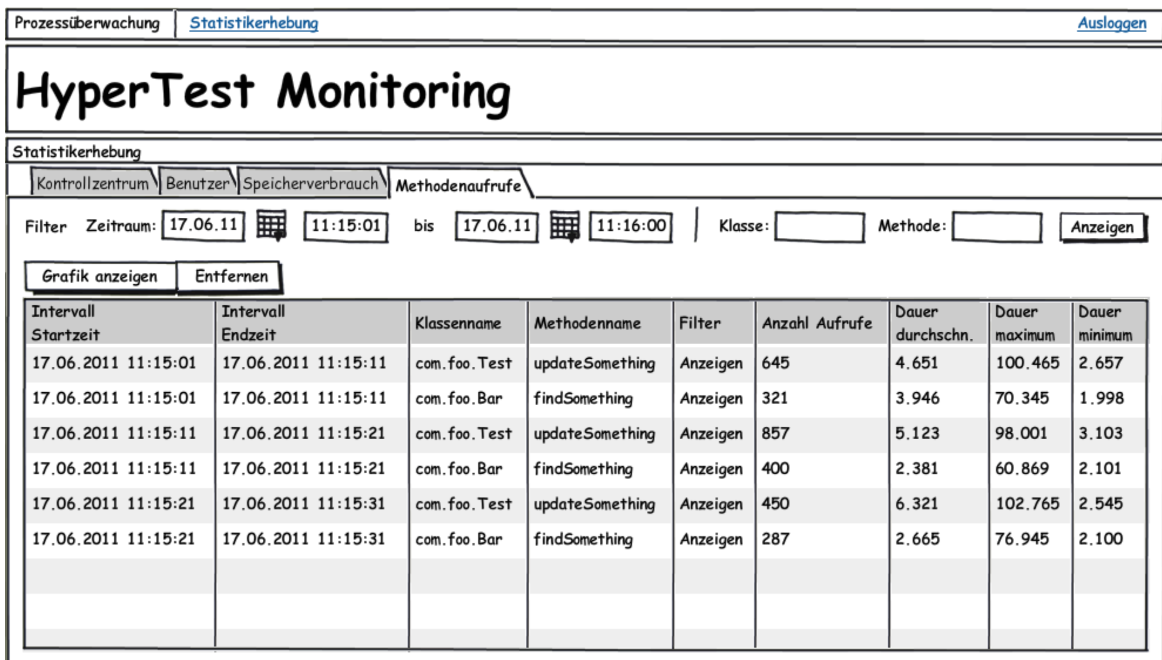

Abbildung A.4.: GUI-Mockup: Ansicht zur Anzeige von Methodenaufrufstatistiken

B. Parser für Filter-Ausdrücke

Zum Einlesen von Filter-Ausdrücken wird ein einfacher Parser verwendet, der mit Hilfe von ANTLR realisiert wurde. Die Grammatik zur Erzeugung des Parsers ist in Listing B.1 dargestellt. Der Parser verwendet eine Klasse *FilterFactory*, um die entsprechenden *Filter*-Objekte zu erzeugen.

```
grammar FilterGrammar;

options{
output=AST;
}

@header {
package com.hypertest.monitoring.statistics.server.filter.parser;

import com.hypertest.monitoring.statistics.server.filter.FilterFactory;
import com.hypertest.monitoring.statistics.server.filter.Filter;
}

@lexer::header {
package com.hypertest.monitoring.statistics.server.filter.parser;
}

@members {
FilterFactory factory = FilterFactory.getInstance();
}

/* Start rule */
start returns [Filter val]
    :  expr EOF { $val = $expr.val; };

expr returns [Filter val]
    : compExpr { $val = $compExpr.val; }
    | simpleExpr { $val = $simpleExpr.val; };

simpleExpr returns [Filter val]
    : SIMPLEOP '(' i=( INT | RETURN ) ',' STRING ')'
      { $val = factory.newSimpleFilter($SIMPLEOP.text, $i.text, $STRING.text); };
```

```
34 compExpr returns [Filter val]
       :   BINARYOP '(' e1=expr ',' e2=expr ')'
36       { $val = factory.newBinaryFilter($BINARYOP.text, $e1.val, $e2.val); }
       |   UNARYOP '(' e=expr ')'
38       { $val = factory.newUnaryFilter($UNARYOP.text, $e.val); };

40 SIMPLEOP
       : 'instanceof'
42     | 'equals';
   UNARYOP
44     : 'not';
   BINARYOP
46     : 'and'
       | 'or';
48 RETURN
       : 'return';
50 INT
       : '0'..'9'+ ;
52 STRING
       : '"' ( ~('"') )* '"';
54 WS
       : (' ' | '\t')+ { skip(); };
```

Listing B.1: ANTLR-Grammatik für einfache Filter-Ausdrücke

Das Ergebnis der Generierung des Parsers durch ANTLR sind zwei Klassen, *FilterGrammar-Lexer* und *FilterGrammarParser*. Diese werden nicht direkt verwendet, sondern von einer weiteren Klasse *FilterParser* aufgerufen. Der relevante Teil dieser Methode zum Einlesen eines Filter-Ausdrucks ist in Listing B.2 dargestellt.

```
1 ANTLRInputStream input = new ANTLRInputStream( new ByteArrayInputStream(
      filterExpression.getBytes() ) );
  FilterGrammarLexer lexer = new FilterGrammarLexer( input );
3 CommonTokenStream tokens = new CommonTokenStream( lexer );
  FilterGrammarParser parser = new FilterGrammarParser( tokens );
5
  Filter result = parser.start().val;
```

Listing B.2: Programmcode zum Einlesen eines Filter-Ausdrucks in der Klasse *FilterParser*

Verläuft das Einlesen des Filter-Ausdrucks erfolgreich, so befindet sich der vollständige Filter anschließend in der Variable `result`. Weitere Informationen zu ANTLR sind unter Parr (2010) zu finden.

C. Inhalt der beiliegenden CD

Dieser Arbeit liegt eine CD mit allen relevanten Daten bei. Sie hat folgenden Inhalt:

- **dist**/ Enthält alle im Rahmen dieser Arbeit entstandenen Bibliotheken als jar-Archive. Außerdem sind notwendige Konfigurationsdateien, Fremdbibliotheken und evtl. vorhandene Startskripte enthalten.
  - **Demo**/ Enthält die Demoklienten.
  - **Monitoring**/ Enthält alle Bibliotheken der Monitoring-Komponente.
  - **Installationshinweise.txt** Hinweise zur Einrichtung und Verwendung der einzelnen Komponenten.
- **doc**/ Enthält Dokumente, die im Rahmen der Arbeit verwendet oder erstellt wurden.
  - **Javadoc**/ Enthält die Javadoc-Dokumentation für alle relevanten Klassen und Schnittstellen der Monitoring-Komponente.
  - **Messung**/ Enthält die Logdateien und Auswertungen der Performanzmessungen.
  - **Quellen**/ Enthält verwendete Paper als PDF-Dateien.
- **screenshots**/ Enthält Screenshots der grafischen Oberfläche.
- **src**/ Enthält den im Rahmen dieser Arbeit erstellen Quellcode.
  - **Demo**/ Enthält den Quellcode der Demoklienten.
  - **Monitoring**/ Enthält den Quellcode der Monitoring-Komponente.
- **BA_MuenchowStefan.pdf** Die digitale Version dieser Arbeit im PDF-Format.

Abkürzungsverzeichnis

Ajax	Asynchronous JavaScript And XML
AOP	Aspektorientierte Programmierung
API	Application Programming Interface
EJB	Enterprise Java Beans
GWT	Google Web Toolkit
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
ICMP	Internet Control Message Protocol
IIOP	Internet Inter-ORB Protocol
J2EE	Java 2 Platform Enterprise Edition
J2SE	Java 2 Platform Standard Edition
JDK	Java Development Kit
JMS	Java Message Service
JMX	Java Management eXtensions
JPA	Java Persistence API
JRE	Java Runtime Environment
JVM	Java Virtual Machine
MBean	Managed Bean
MOM	Message Oriented Middleware
ORB	Object Request Broker
SQL	Structured Query Language
XML	eXtensible Markup Language

Abbildungsverzeichnis

2.1. Übersicht über das HyperTest-System 6
2.2. Übersicht über die J2EE-Architektur (Sun Microsystems 2003b, S. 6) 8
2.3. Struktur des Interzeptor-Entwurfsmusters (vgl. Schmidt 2000, S. 115) 16
2.4. Funktionsweise eines Aspekt-Weavers 19

3.1. Die grafische Oberfläche von Glassbox (Glassbox Corporation 2008) 28
3.2. Die grafische Oberfläche von RHQ 30

4.1. Struktur der Monitoring-Komponente 33
4.2. Verteilung der Artefakte auf physikalische Computer 35
4.3. Überwachung von Prozessen über eine JMS-Queue 36
4.4. Request-Reply-Zyklus über JMS-Topic 38
4.5. Zentrale Struktur der Watchdog-Subkomponente 40
4.6. Beteiligte Klassen der Subjektüberwachung 41
4.7. Beteiligte Klassen der Datenbanküberwachung 42
4.8. Ermittlung des Datenbankzustands 43
4.9. Zentrale Struktur der Statistics-Subkomponente 46
4.10. Implementierungen der Schnittstelle *StatisticsCollectorAction* 47
4.11. Datenmodell für statistische Daten 49
4.12. Methodenfilter zur Einschränkung der erfassten Methoden 50
4.13. Ablauf eines entfernten Aufrufs mit aktiver Statistikerhebung 53
4.14. Beispiel für die Aufteilung von Layout und Logik in der GUI 54
4.15. Beispiel für die Kommunikation der GUI mit dem Server 55

5.1. *Timer* und *TimerInfo* 56
5.2. Logik bei Ablauf eines Timers innerhalb der *WatchdogServiceBean* 57
5.3. EchoRequest- und EchoReply-Nachricht der Prozessüberwachung 58
5.4. Erhebung und Speicherung von Daten für eine Statistik-Kategorie 62
5.5. Syntax-Diagramm für einfache Methodenfilter 65
5.6. Screenshot: Ansicht zur Anzeige von statistischen Daten 66
5.7. Initialiserung der Ansichten für vorhandene Statistik-Kategiorien auf der GUI . 67

A.1. GUI-Mockup: Ansicht für Prozess-Überwachung 75

A.2. GUI-Mockup: Kontrollzentrum für die Statistikerhebung 76
A.3. GUI-Mockup: Ansicht zur Anzeige allgemeiner Statistiken 77
A.4. GUI-Mockup: Ansicht zur Anzeige von Methodenaufrufstatistiken 77

Quellcodeverzeichnis

5.1. Beispiel für den MessageSelector eines Subjekts 58
5.2. Konfiguration der Datenbank-Testabfrage in XML 59
5.3. Beispiel für die Erzeugung einer *WatchdogSubject*-Instanz 59
5.4. Implementierung der MemoryStatisticsAction 60
5.5. Definition der Aspekt-Multiplizität 63
5.6. Definition eines Pointcuts für Bean-Methoden 64
5.7. Around-Advice zur Erfassung von Methodenaufrufen 64

B.1. ANTLR-Grammatik für einfache Filter-Ausdrücke 78
B.2. Programmcode zum Einlesen eines Filter-Ausdrucks in der Klasse *FilterParser* 79

Literaturverzeichnis

[Arendsen 2007] ARENDSEN, Alef ; SPRINGSOURCE (Hrsg.): *Debunking myths: proxies impact performance*. 2007. – URL http://blog.springsource.com/2007/07/19/debunking-myths-proxies-impact-performance/. – Zugriffsdatum: 19.07.2011

[Bodkin 2005] BODKIN, Ron ; IBM DEVELOPER WORKS (Hrsg.): *Performance monitoring with AspectJ: A look inside the Glassbox Inspector with AspectJ and JMX*. 2005. – URL http://www.ibm.com/developerworks/java/library/j-aopwork10/. – Zugriffsdatum: 19.07.2011

[Böhm 2006] BÖHM, Oliver: *Aspektorientierte Programmierung mit AspectJ 5: Einsteigen in AspectJ und AOP*. 1. Heidelberg : dpunkt-Verl., 2006. – URL http://www.worldcat.org/oclc/179971980. – ISBN 3898643301

[Fleury und Reverbel 2003] FLEURY, Marc ; REVERBEL, Francisco: The JBoss Extensible Server. In: ENDLER, Markus (Hrsg.) ; SCHMIDT, Douglas (Hrsg.): *Lecture Notes in Computer Science*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003, S. 344–373. – ISBN 978-3-540-40317-3

[Fleury u. a. 2006] FLEURY, Marc ; STARK, Scott ; RICHARDS, Norman: *JBoss 4.0: Das offizielle Handbuch*. München : Addison-Wesley, 2006. – URL http://www.worldcat.org/oclc/162305035. – ISBN 3827323193

[Gamma 2008] GAMMA, Erich: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. [Nachdr.]. München [u.a.] : Addison-Wesley, 2008. – URL http://www.worldcat.org/oclc/263423344. – ISBN 9783827321992

[Glassbox Corporation 2008] GLASSBOX CORPORATION: *Glassbox Open Source*. 2008. – URL http://glassbox.sourceforge.net/glassbox/Home.html. – Zugriffsdatum: 19.07.2011

[Google 2011a] GOOGLE: *Google Web Toolkit Overview*. 2011. – URL http://code.google.com/intl/de-DE/webtoolkit/overview.html. – Zugriffsdatum: 19.07.2011

[Google 2011b] GOOGLE: *GWT Google APIs: The Official Google API Libraries for Google Web Toolkit*. 2011. – URL http://code.google.com/p/gwt-google-apis/wiki/VisualizationGettingStarted. – Zugriffsdatum: 19.07.2011

[JBoss Community 2008] JBOSS COMMUNITY: *StatisticsCollector*. 2008. – URL http://community.jboss.org/wiki/StatisticsCollector. – Zugriffsdatum: 19.07.2011

[Johnson u. a. 2008] JOHNSON, Rod ; HOELLER, Juergen ; ARENDSEN, Alef ; SAMPALEANU, Colin ; HARROP, Rob ; RISBERG, Thomas ; DAVISON, Darren ; KOPYLENKO, Dmitriy ; POLLACK, Mark ; TEMPLIER, Thierry ; VERVAET, Erwin ; TUNG, Portia ; HALE, Ben ; COLYER, Adrian ; LEWIS, John ; LEAU, Costin ; FISHER, Mark ; BRANNEN, Sam ; LADDAD, Ramnivas ; POUTSMA, Arjen: *The Spring Framework - Reference Documentation 2.5.6: Chapter 6*. 2008. – URL http://static.springsource.org/spring/docs/2.5.x/reference/aop.html. – Zugriffsdatum: 19.07.2011

[Kersten 2005a] KERSTEN, Mik: *AOP Tools Comparison, Part 1: Language Mechanisms*. 2005. – URL http://www.ibm.com/developerworks/library/j-aopwork1/. – Zugriffsdatum: 19.07.2011

[Kersten 2005b] KERSTEN, Mik: *AOP Tools Comparison, Part 2: Development environments*. 2005. – URL http://www.ibm.com/developerworks/library/j-aopwork2/. – Zugriffsdatum: 19.07.2011

[Kiczales u. a. 1997] KICZALES, Gregor ; LAMPING, John ; MENDHEKAR, Anurag ; MAEDA, Chris ; LOPES, Cristina ; LOINGTIER, Jean-Marc ; IRWIN, John: Aspect-oriented programming. In: AKSIT, Mehmet (Hrsg.) ; MATSUOKA, Satoshi (Hrsg.): *Lecture Notes in Computer Science*. Berlin/Heidelberg : Springer-Verlag, 1997, S. 220–242. – ISBN 3-540-63089-9

[Kreger u. a. 2003] KREGER, Heather ; HAROLD, Ward ; WILLIAMSON, Leigh: *Java and JMX: Building manageable systems*. Boston : Addison-Wesley, 2003. – URL http://www.worldcat.org/oclc/473295530. – ISBN 0672324083

[Langner und Reiberg 2006] LANGNER, Torsten ; REIBERG, Daniel: *J2EE und JBoss: Grundlagen und Profiwissen : verteilte Enterprise-Applikationen auf Basis von J2EE, JBoss & Eclipse*. München [u.a.] : Hanser, 2006. – URL http://www.worldcat.org/oclc/162245282. – ISBN 3446405089

[Monson-Haefel 2002] MONSON-HAEFEL, R.: *Enterprise JavaBeans*. 2. Beijing ;, Cambridge ;, Farnham ;, Köln ;, Paris ;, Sebastopol ;, Taipei ;, Tokyo : O'Reilly, 2002. – URL http://www.worldcat.org/oclc/76373879. – ISBN 3897212978

[Oracle 1999] ORACLE: *Dynamic Proxy Classes*. 1999. – URL http://download.oracle.com/javase/1.3/docs/guide/reflection/proxy.html. – Zugriffsdatum: 19.07.2011

[Parr 2010] PARR, Terence: *ANTLR v3*. 2010. – URL http://www.antlr.org/. – Zugriffsdatum: 19.07.2011

[Postel 1981] POSTEL, J.: *RFC792: Internet Control Message Protocol*. 1981. – URL http://tools.ietf.org/html/rfc792. – Zugriffsdatum: 19.07.2011

[RHQ Project 2011] RHQ PROJECT: *RHQ Projekt*. 2011. – URL http://www.rhq-project.org/display/RHQ/Home. – Zugriffsdatum: 19.07.2011

[Schmidt 2000] SCHMIDT, Douglas C.: *Pattern-oriented software architecture*. Chichester : Wiley, 2000. – URL http://www.worldcat.org/oclc/265337150. – ISBN 0471606952

[Sencha 2011] SENCHA ; SENCHA (Hrsg.): *Ext GWT: Internet Application Framework for Google Web Toolkit*. 2011. – URL http://www.sencha.com/products/extgwt/. – Zugriffsdatum: 19.07.2011

[Sun Microsystems 2002a] SUN MICROSYSTEMS: *Java Management Extensions Instrumentation and Agent Specification, v1.1*. 2002. – URL http://download.oracle.com/javase/6/docs/technotes/guides/jmx/index.html

[Sun Microsystems 2002b] SUN MICROSYSTEMS: *Java Message Service Specification: Version 1.1*. 2002. – URL http://www.oracle.com/technetwork/java/docs-136352.html

[Sun Microsystems 2003a] SUN MICROSYSTEMS: *Enterprise JavaBeans Specification, Version 2.1*. 2003. – URL http://java.sun.com/products/ejb/docs.html

[Sun Microsystems 2003b] SUN MICROSYSTEMS: *Java 2 Platform, Enterprise Edition (J2EE) Specification: Version 1.4*. 2003. – URL http://java.sun.com/j2ee/1.4/docs/#specs

[Sun Microsystems 2006] SUN MICROSYSTEMS: *Enterprise JavaBeans, Version 3.0: EJB Core Contracts and Requirements*. 2006. – URL http://java.sun.com/products/ejb/docs.html

[Tanenbaum und van Steen 2008] TANENBAUM, Andrew S. ; STEEN, Maarten van: *Verteilte Systeme: Prinzipien und Paradigmen*. 2., aktualisierte Aufl. München [u.a.] : Pearson Studium, 2008. – URL http://www.worldcat.org/oclc/237201545. – ISBN 3827372933

[Werum 2011] WERUM: *HyperTest Plattform Version 2 Informationsmanagement für Prüfungen und Messungen: Allgemeines Benutzerhandbuch: Version 1.36*. 2011

[Xerox Corporation 2003–2005] XEROX CORPORATION: *The AspectJ Development Environment Guide*. 2003-2005. – URL http://www.eclipse.org/aspectj/doc/released/devguide/index.html. – Zugriffsdatum: 19.07.2011

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 26. Juli 2011	
Ort, Datum	Unterschrift