Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. E. Plödereder
Betreuer:	Prof. Dr. E. Plödereder Dr. O. Wohlmuth (IBM Deutschland Entwicklung GmbH)
Beginn am:	1. Februar 2001
Beendet am:	28. August 2001
CR-Klassifikation:	H.4.3, D.2.12

Diplomarbeit-Nr. 1924

Vergleichende Implementierung einer verteilten Anwendung unter Nutzung von CORBA/IIOP, RMI und JSP

Kristian Tandjung

Inhaltsverzeichnis

1 Einleitung 6

1.1 Motivation 6

1.2 Aufgabenstellung 6

1.3 Umfeld bei der Durchführung der Diplomarbeit 7

1.4 Aufbau dieses Dokuments 7

2 Genutzte Technologien 8

2.1 Die Programmiersprache Java 8

2.2 Java Applets 9

2.3 JavaServer Pages™ (JSP) 10

2.4 CORBA / IIOP 11

2.5 Java RMI 12

3 Vorbereitung der Implementierung 13

3.1 Vorgehensplanung 13

3.2 Installation von Dritt-Software 15

3.3 Analyse und Spezifikation des „Problem Tracking“ Systems 16

3.3.1 Analyse des „Problem Tracking“-Workflows 16
3.3.2 Struktur der vorgegebenen Datenbank 18
3.3.3 Spezifikation 19
3.3.4 Testplanung 26

4 Entwicklung von PTS/RMI 28

4.1 Client-Server Kommunikation 28

4.2 Entwurf 29

4.2.1 Modellierung der Domain Klassen 30
4.2.2 Modellierung der Server-Schicht 32
4.2.3 Modellierung des Clients 33

4.3 Implementierung und Integration 41

4.3.1 Datenbankanbindung 42
4.3.2 Umwandlung von Datenbankanfragen in Domain Objekte 42
4.3.3 Grafische Darstellung von Domain Objekten 43

4.4 Erweiterung 44

4.4.1 Änderungen an der Datenbank 45
4.4.2 Änderungen an den Domain Klassen 45
4.4.3 Änderungen an der Server-Schicht 45
4.4.4 Änderungen am Client 46

5 Entwicklung von PTS/CORBA 47

5.1 Client-Server Kommunikation 47

5.2 Entwurf **48**
5.2.1 Modellierung der Domain Klassen 49
5.2.2 Modellierung der Server-Schicht 51
5.2.3 Modellierung des Clients 52

5.3 Implementierung und Integration **52**
5.3.1 Implementierung der Domain Klassen 52
5.3.2 Umwandlung von Datenbankanfragen in Domain Objekte 53
5.3.3 Grafische Darstellung von Domain Objekten 54
5.3.4 Kopie der Daten von Listenausschnitten 55

5.4 Erweiterung **56**
5.4.1 Änderungen an der Datenbank 56
5.4.2 Änderungen an der IDL Schnittstelle 56
5.4.3 Änderungen an den Domain Klassen 57
5.4.4 Änderungen an der Server-Schicht 57
5.4.5 Änderungen am Client 57

6 Entwicklung von PTS/JSP ***58***

6.1 Client-Server Kommunikation **58**

6.2 Entwurf **59**
6.2.1 Beziehungen zwischen den JSP-Seiten von PTS/JSP 61

6.3 Implementierung und Integration **62**

6.4 Erweiterung **63**
6.4.1 Änderungen an den Domain Klassen 63
6.4.2 Änderungen am Applikationskoordinator 63
6.4.3 Änderungen an JSP-Seiten 64
6.4.4 Änderungen am JSP-Koordinator 64

7 Durchführung funktionaler Tests ***65***

7.1 Funktionaler Test nach der Integration **65**

7.2 Funktionaler Test nach der Erweiterung **67**

8 Vergleich und Diskussion ***68***

8.1 Einschätzung der spezifizierten Funktionalität **68**

8.2 Erstellungsaufwand **69**
8.2.1 Einflüsse durch den Entwicklungsverlauf 69
8.2.2 Einflüsse durch Entwicklererfahrung 70
8.2.3 Einflüsse durch Technologien 71
8.2.4 Schätzungen für Entwicklungszeiten isolierter Realisierungen 72

8.3 Umfang des Quellcodes **73**
8.3.1 Produktivität 75

8.4 Erweiterbarkeit und Wartbarkeit **75**

9 Performanztests ***79***

9.1 Ziele **79**

9.2 Testumgebung **79**
9.3 Simulation von Benutzerverhalten **80**
9.4 Messmethoden **83**
9.5 Planung **85**
9.6 Messergebnisse **86**
9.6.1 Behinderung von Benutzern 87
9.6.2 Verteilung der Rechenzeit 87
9.7 Optimierungsmöglichkeiten **88**
9.8 Interpretation **88**
10 Schlussfolgerungen und Ausblick ***89***
Anhang ***91***
Anhang A Hilfsmittel **91**
Studium 91
Literatur 91
Betreuung 93
Dritt-Software 93
Anhang B Erklärung **94**

1 Einleitung

1.1 Motivation

Für die Realisierung von Client-Server Anwendungen in Internet oder Intranet kommen viele verschiedene Ansätze in Betracht. Jeder Realisierungsansatz impliziert eine eigene Systemstruktur und nutzt für die Kommunikation zwischen Client und Server eine spezielle Infrastruktur. Zumeist basieren diese Kommunikationsinfrastrukturen auf (zumindest theoretisch) plattformunabhängigen Standards.

Die sich daraus ergebenden Unterschiede zwischen drei Realisierungsansätzen - sowohl bei der Entwicklung als auch im Betrieb einer Client-Server Anwendung - sollen im Rahmen dieser Diplomarbeit untersucht werden.

1.2 Aufgabenstellung

Anhand einer prototypischen Implementierung der Basisfunktionalität eines „Problem Tracking“ Systems sollen drei Realisierungsansätze in Hinblick auf Entwicklungsaufwand, Erweiterbarkeit, Wartbarkeit und Performanz bewertet und verglichen werden. Die drei Realisierungsansätze sind:

1. Java Client-Applikationen, die CORBA/IIOP zur Kommunikation mit einer Java Server-Applikation nutzen. Dieser Ansatz wäre für eine neue Java Applikation denkbar, die auf einem bestehenden CORBA Server aufbaut.

2. Java Client-Applets, die mittels RMI eine Java Server-Applikation ansprechen. Dies ist ein allein auf Java zentrierter Realisierungsansatz.

3. JavaServer Pages: Für die Nutzung der Applikation wird auf Seiten des Clients lediglich ein Web-Browser benötigt. Dieser Ansatz eignet sich besonders für den Einsatz einer Applikation in einem Intranet oder im Internet.

Eine im Verlauf der Diplomarbeit zu erstellende Spezifikation des „Problem Tracking“ Systems dient als gemeinsame Grundlage für alle Implementierungen der Realisierungsansätze.

Die Realisierungsansätze werden im weiteren Verlauf des Dokuments als Varianten (im Sinne von [SE 1998]: Ergebnisse von Verzweigungen einer Software-Entwicklung) bezeichnet, da sie auf einer gemeinsamen Spezifikation basieren. Alle drei Varianten sollen dieselbe Funktionalität realisieren..

Einige Aspekte und Fragestellungen beim Vergleich der Varianten sind:

1. Projektdurchführung und Projektergebnisse: Welchen Entwicklungsaufwand benötigen die verschiedenen Varianten? Welchen Umfang und welche Struktur hat die entwickelte Software?

2. Nachträgliche Implementierung einer zusätzlichen Funktion: Welchen Aufwand und welche Auswirkungen auf den Code bringt eine nachträgliche Erweiterung der Funktionalität mit sich?

3. Performanz beim Benutzer: Wie viel Zeit vergeht zwischen einer Benutzeraktion und der grafischen Anzeige durch die Client-Benutzungsoberfläche bei der Übertragung von kleinen und großen Datenmengen von Server zu Client?

4. Server-Performanz in Abhängigkeit von der Benutzeranzahl: Wie viele Benutzer kann der Server gleichzeitig ohne merkliche Beeinträchtigung der Benutzer bedienen?

1.3 Umfeld bei der Durchführung der Diplomarbeit

Diese Diplomarbeit wurde am IBM Forschungszentrum Böblingen durchgeführt und durch Dr. Otto Wohlmuth mitbetreut. Die bei der Durchführung genutzten Rechnersysteme, die Netzwerkinfrastruktur, die Datenbanksoftware IBM DB2 sowie eine bereits gefüllte Datenbank wurden von der IBM zur Verfügung gestellt.

Aus Gründen der Geheimhaltung werden keine vertraulichen Informationen der IBM in dieser Diplomarbeit veröffentlicht.

1.4 Aufbau dieses Dokuments

Der Inhalt dieses Dokuments gliedert sich in drei Abschnitte. Im ersten Abschnitt (die Kapitel 1 und 2) werden Aufgabenstellung, organisatorisches und technisches Umfeld der Diplomarbeit beschrieben. Der erste Abschnitt soll dem Leser einen groben Überblick über das Aufgabengebiet bieten.

Der zweite Abschnitt (die Kapitel 3 bis 7) behandelt die softwaretechnische Umsetzung der drei Varianten. Dabei werden sowohl die Tätigkeiten bei der Software-Entwicklung als auch die entstandene Software selbst (hauptsächlich deren Struktur und die verwendeten Konzepte) beschrieben. Dieses Wissen wird für die anschließende Bewertung der drei Varianten von Bedeutung sein.

Im dritten Abschnitt (Kapitel 8 bis 10) werden subjektive Erfahrungen aus der Entwicklung sowie Daten über Entwicklung und Betrieb der drei Varianten aufgeführt. Diese dienen dann als Grundlage für Aussagen über Unterschiede zwischen den Varianten sowie über Vor- und Nachteile von einzelnen Varianten.

In Kapitel 10 werden die gewonnenen Erkenntnisse noch einmal kurz zusammengefasst.

2 Genutzte Technologien

In diesem Unterkapitel werden die in den drei Realisierungsansätzen genutzten Technologien kurz beschrieben. Außerdem wird begründet, warum der Einsatz einer speziellen Technologie in dieser Diplomarbeit sinnvoll erschien.

2.1 Die Programmiersprache Java

Eine Implementierung der Varianten in mehreren Programmiersprachen wäre allein wegen der Effekte durch die Unterschiede zwischen den Programmiersprachen wesentlich schwieriger zu vergleichen und kam nicht in Frage. Die Wahl für die einzige in dieser Diplomarbeit zu nutzende Programmiersprache fiel auf Java.

Java ist eine relativ junge objektorientierte Programmiersprache. Einige Vorzüge der Programmiersprache Java sind unter anderem (aus [JavaKK 1999] und [SunJavaTutorial]):

- Objektorientierte Konzepte: Java unterstützt Klassen und Interfaces, Attribute und Methoden auf Instanz- und Klassenebene, Vererbung, Polymorphismus und dynamisches Binden ([KAOS 1998]).

- Plattformunabhängigkeit: Java-Compiler generieren plattformunabhängigen Bytecode für eine virtuelle Maschine (Java Virtual Machine). Interpreter für diesen Bytecode sind für alle gängigen Plattformen kostenlos erhältlich. Java enthält außerdem eine plattformunabhängige Graphikbibliothek (AWT bzw. Swing). Dies vereinfacht die Entwicklung von Client-Programmen in heterogenen Netzwerkumgebungen.

- Parallelität: Java unterstützt Threads (leichtgewichtige Prozesse) und deren Synchronisation. Dadurch ist die parallele Bearbeitung von mehreren Client-Anfragen durch ein einzelnes Server-Objekt möglich.

- Bibliotheken zur Unterstützung von TCP/IP, CORBA/IIOP oder Applets sind in der Java 2 SDK Entwicklungsumgebung ([SunJavaSdk 1.3]) enthalten. Darüber hinaus kann der Entwickler eine Datenbankanbindung über die „Java Database Connectivity“ (JDBC) Schnittstelle realisieren.

- Die Java Entwicklungsumgebung ist kostenlos erhältlich und bietet sich dadurch als Programmiersprache bei nicht-kommerziellen Open-Source Projekten an.

Ausschlaggebend für die Nutzung von Java in dieser Diplomarbeit war die Kombination aus der Plattformunabhängigkeit der erstellten Programme einerseits und der Vielzahl an Möglichkeiten, Client-Server Software damit zu erstellen, andererseits. Für Web-basierte Applikationen können zum Beispiel CGI-Skripte in den Skriptsprachen Perl oder PHP verwendet werden. Perl und PHP eignen sich jedoch nicht für die Erstellung von eigenen Client-Programmen, die über ein Netzwerk mit einem Server Objekt kommunizieren. Dies wiederum ist mit den Programmiersprachen C++, Smalltalk oder

Cobol unter anderem durch die Verwendung von CORBA möglich. Client-Programme in diesen Sprachen sind jedoch nur mit entsprechendem Aufwand in heterogenen Netzwerkumgebungen zum Einsatz zu bringen. Die Nutzung der Programmiersprache Java nimmt dem Software-Entwickler die Bewältigung der meisten Portabilitätsschwierigkeiten bei der Erstellung von Client-Programmen ab. Für die Client-Server Kommunikation in Java steht dem Entwickler zusätzlich zu CORBA oder einer eigenen Socket-Lösung der Java-spezifische Mechanismus „RMI“ zur Verfügung. Mit Hilfe der Servlet und „JavaServer Pages“ Technologien können mit Java auch Web-basierte Applikationen erstellt werden.

2.2 Java Applets

Java Applets sind von einem Applet-Server über das Internet übertragene und in einem Web-Browser eines Client-Rechners ausgeführte Java Programme. Dadurch wird die Verteilung und Aktualisierung von Programmen vereinfacht, denn die jeweils aktuellste Version eines Programms muss nur auf dem Applet-Server als Bytecode bereitgestellt werden. Bei der nächsten Ausführung des Java Applets auf einem Client-Rechner wird die aktuelle Version übertragen und gestartet.

Es gibt zwei Möglichkeiten den Bytecode eines Java Applets zu übertragen:

1. Alle Bytecode-Dateien werden in einem Java-Archiv (JAR) komprimiert und zusammengefasst und vor Beginn der Ausführung des Java Applets übertragen.
2. Der Bytecode einer Klasse wird immer erst dann übertragen, wenn auf diese Klasse zugegriffen wird. In diesem Fall muss auf dem Applet-Server der Bytecode jeder einzelnen Klasse zugänglich gemacht werden.

Für das Client-Applet in dieser Diplomarbeit wurde die erste Möglichkeit gewählt. Denn die Vorteile durch die Komprimierung der zu übertragenden Daten und die geringere Anzahl an Netzwerkverbindungen zum Applet-Server überwogen den Nachteil einer längeren Wartezeit für den Start des Client-Applets.

Zur Ausführung von Java Applets, die auf Eigenschaften jüngerer Java Versionen (ab Version 1.2) zugreifen, muss bei den am weitesten verbreiteten Web-Browsern ein Plug-In installiert werden. Das war auch für den in dieser Diplomarbeit verwendeten Browser [Netscape 4.72] notwendig. Die dazu benötigten Dateien werden in der Java-Laufzeitumgebung ([SunJavaRun 1.3]) mitgeliefert.

Damit sichergestellt ist, dass mit Java Applets keine Schäden auf den ausführenden Rechnern verursacht werden können, existieren für Applets besondere Sicherheitseinschränkungen. Das sogenannte Sandbox-Prinzip (aus [SunJavaTutorial] und [JavaKK 1999]) beschränkt den Wirkungsradius eines Programms auf einen Bereich, in dem keine Schäden angerichtet können:

- Applets können nicht auf Dateien oder Verzeichnisse des Client-Rechners zugreifen. Applets können dadurch keine Dateien löschen, überschreiben oder verschieben. Auch das Ausspähen von vertraulichen gespeicherten Daten ist dadurch nicht mehr möglich.
- Applets dürfen nur Netzwerkverbindungen zum Applet-Server Rechner aufbauen. Dies soll sicherstellen, dass Daten nicht unbemerkt an Dritte übertragen werden können.

- Applets können keine anderen Programme auf einem Client-Rechner starten. Das Sandbox-Prinzip kann auf diese Weise nicht umgangen werden.
- Die von einem Applet erzeugten Fenster werden gekennzeichnet, um Imitationen bekannter Programme zu erschweren.

2.3 JavaServer Pages™ (JSP)

Mit der JavaServer Pages Technologie können Webseiten mit dynamischem Inhalt erzeugt werden. JSP-Dateien sind Textdokumente, in denen statisches Layout (hier in Form von HTML-Tags) mit dynamischen Abläufen und Daten (Skripte und JavaBeans) kombiniert werden kann.

Die JSP-Technologie basiert auf der Servlet-Technologie von Java. Servlets sind Java Programme, die auf HTTP-Anfragen reagieren und HTTP-Ausgaben erzeugen können. HTTP-Anfragen und HTTP-Ausgaben werden innerhalb des Servlets durch Objekte repräsentiert und bearbeitet. Servlets können (in ihrer Eigenschaft als Java Programm) auf die zahlreichen Java APIs zurückgreifen und eignen sich deshalb für Web-basierte Applikationen, die auf ein bereits bestehendes Java System aufbauen, oder über eine Java API ein anderes System ansprechen (zum Beispiel über JDBC eine Datenbank).

Ein Schwachpunkt der Servlet-Technologie ist jedoch die Verflechtung von Java Code und HTML bei der Generierung der HTTP-Ausgabe. Mit JSP-Seiten soll nun vorrangig die Präsentationsschicht erstellt und die Ausgabe von HTML-Text vereinfacht werden. In JSP-Seiten kann die von einem Web-Browser darzustellende HTTP-Ausgabe wie in HTML-Seiten durch Text und HTML-Tags beschrieben werden. Die Präsentationsschicht kann also wie bei herkömmlichen HTML-Seiten von HTML-Autoren und Grafikern entwickelt werden.

Aus einer JSP-Datei wird (durch eine JSP-Engine innerhalb des JSP-Containers) automatisch ein Servlet generiert, das vom JSP-Container als JSP-Seite verwaltet und angesprochen wird. Der in dieser Diplomarbeit verwendete JSP-Container [Tomcat 3.1.1] kann in den Webserver [Apache 1.3.14] integriert werden. Der Webserver leitet dann HTTP-Anfragen an Adressen, die zu JSP-Seiten führen, an den JSP-Container weiter. Die HTTP-Anfragen werden vom JSP-Container in Java Objekte transformiert und an das generierte Servlet übergeben. Das Servlet bearbeitet die Anfrage, und liefert eine Antwort in Form von HTML-Text zurück.

In JSP-Seiten steht darüber hinaus Java als Skriptsprache zur Verfügung. Damit kann zum Beispiel durch die Iteration über Spalten und Zeilen mit relativ wenig Code der HTML-Text einer umfangreichen Tabelle generiert werden. Die wichtigste Eigenschaft von JSP-Seiten ist die Möglichkeit, auf Objekte beliebiger Java Klassen (JavaBeans) für die Generierung der HTML-Ausgabe zugreifen zu können. Die in JSP-Seiten verwendeten JavaBean-Objekte werden durch den JSP-Container verwaltet.

Die JSP-Technologie ermöglicht die einfache Integration von Java Klassen in eine Web-basierte Applikation. Eine Benutzungsoberfläche kann mit HTML-Tags relativ schnell erstellt werden. Außerdem können die bei der Entwicklung der beiden anderen Varianten entstandenen Java Klassen in der JSP-Variante wiederverwendet werden.

2.4 CORBA / IIOP

CORBA ist die Abkürzung für “Common Object Request Broker Architecture”. CORBA wurde von der Object Management Group (OMG) entwickelt und soll die Kommunikation zwischen Objekten auf verschiedenen Rechnern ermöglichen. Die Infrastruktur für die Kommunikation wird durch „Object Request Broker“ (ORB) zur Verfügung gestellt. CORBA ist unabhängig von einer bestimmten Plattform und einer speziellen Programmiersprache.

Mit Hilfe der „CORBA Interface Definition Language“ (IDL) können die Schnittstellen von Objekten definiert werden. In IDL können keine Algorithmen und damit kein Verhalten von Objekten beschrieben werden. Abbildungen von IDL zu verschiedenen Programmiersprachen (unter anderem C++, Smalltalk und Java) werden von der OMG festgelegt.

Die Schnittstelle eines Objekts besteht aus der IDL Definition der Menge von Operationen, die ein Client aufrufen kann. Operationen entsprechen den Methoden eines Objekts und werden in IDL unter anderem durch Name, Eingabeparameter und Typ der Ausgabe definiert. Als Typen der Eingabeparameter und der Ausgabe sind nur IDL Datentypen zugelassen.

Über die in IDL definierten Schnittstellen können „Object Request Broker“ Botschaften zwischen Objekten auf verschiedenen Rechnern weiterleiten. Für die Kommunikation der ORB untereinander, wurden in CORBA mehrere Protokolle spezifiziert. Ein besonderes Protokoll, das „Internet InterORB Protocol“ (IIOP) ermöglicht die Kommunikation zwischen Objekten, deren ORB über ein Inter-/Intranet Netzwerk kommunizieren können.

CORBA bietet sich durch eine eigene Schnittstellendefinitionssprache und eine einheitlichen Kommunikationsinfrastruktur für die Entwicklung von plattformübergreifender und verteilter Software an. CORBA eignet sich deswegen besonders für die Integration bestehender Systeme in neue Software.

Es gibt eine Vielzahl von ORB Implementierungen von verschiedenen Herstellern. In der Literatur sind bereits Vergleiche zwischen einzelnen ORB Implementierungen veröffentlicht worden (beispielsweise [Hirano et al. 1998] oder [Gokhale et al.1996]). Für diese Diplomarbeit, die sich auf die Programmiersprache Java konzentriert, wird der ORB von SUN genutzt, der in der Java Laufzeitumgebung [SunJavaRun 1.3] enthalten ist. Der ORB von SUN unterstützt lediglich transiente Objekte, kann Objekte also nicht zu einem anderen ORB transferieren.

CORBA/IIOP ist eine plattformunabhängige und programmiersprachunabhängige Kommunikationsinfrastruktur mit eigenen Datentypen für den Datenaustausch unter Objekten. Die sich aus dieser Infrastruktur ergebende Software wird in dieser Diplomarbeit mit Software verglichen, die entweder einen programmiersprachspezifischen Kommunikationsmechanismus (Java RMI) oder einen auf Text basierten Datenaustausch (HTTP bei JavaServer Pages) zur Client-Server Kommunikation verwendet.

2.5 Java RMI

Der “Remote Method Invocation” (RMI) Mechanismus von Java ermöglicht die Kommunikation von Java Objekten, die sich in verschiedenen „Java Virtual Machines“ (JVMs) befinden. Im Gegensatz zu SUN RPC oder CORBA ist Java RMI auf eine spezielle Programmiersprache – Java – ausgerichtet.

Mit Java RMI können Objekte auf einer anderen JVM genauso angesprochen werden, wie Objekte auf derselben JVM. Da der Java Code, der entfernte Objekte anspricht, sich (bis auf die Notwendigkeit zusätzliche `Exception` Klassen behandeln zu müssen) nicht von Java Code unterscheidet, der auf ein Java Interface zugreift, erfolgt die Kommunikation zwischen den beiden JVMs für den Entwickler nahezu transparent.

Objekte, die nicht als entfernte Objekte angesprochen werden können und Parameter eines entfernten Methodenaufrufs oder Ergebnis eines solchen sind, werden vollständig zwischen Client und Server übertragen („by-value“ Semantik). Dies bedeutet, dass rekursiv auch alle in den Instanzvariablen eines solchen Objekts referenzierten Objekte bei diesem Vorgang übertragen werden. Dabei wird, falls die Klasse eines dieser Objekte auf Seiten des Empfängers noch nicht bekannt ist, der Bytecode der Klasse zur JVM des Empfängers übertragen. Der Bytecode neuer Klassen wird vom Sicherheitssystem der empfangenden JVM („Security Manager“ und „Class Loader“) geprüft.

Im Gegensatz zu CORBA kann das Verhalten von Objekten mit Java RMI zwischen den Teilen eines verteilten Java System ausgetauscht werden. Diese Eigenschaft von Java RMI ist besonders nützlich, denn sie erlaubt es komplexe Strukturen (wie zum Beispiel eine Hash-Tabelle) einschließlich aller darin enthaltenen Objekte mit einer einzigen Java Anweisung zu versenden. Bei der Datenübertragung können somit alle Klassen der Java API genutzt werden.

Die Übertragung von Objektverhalten kann bei entsprechenden Strukturen auch für die einfache Erweiterung der Systemfunktionalität genutzt werden, indem ein Objekt einer spezialisierten Unterklasse anstelle des ursprünglichen Objekts übermittelt wird.

Java RMI bietet als verteiltes Gegenstück zur lokalen Speicherverwaltung („Garbage Collection“) ein sogenanntes „Distributed Garbage Collection“. Die Verwaltung von Objekten erfolgt hierbei mit Hilfe eines „reference-counting“ Algorithmus. Der Entwickler braucht sich also auch bei verteilten Applikationen nicht explizit um die Speicherverwaltung kümmern.

Die Vorteile von Java RMI gegenüber CORBA beruhen auf der Spezialisierung auf eine einzige Sprache und Plattform – Java und die „Java Virtual Machine“. Dadurch ist Java RMI von vorneherein auf die Kommunikation zwischen Java Objekten sowohl spezialisiert als auch beschränkt. In dieser Diplomarbeit soll ermittelt werden, ob und wie weit sich diese Eigenschaft bei Entwicklung oder Betrieb des Java Client-Applets bemerkbar macht.

3 Vorbereitung der Implementierung

Dieses Kapitel beschreibt die ersten und gemeinsamen Schritte der Software-Entwicklung von den drei „Problem Tracking“ Systemvarianten während der Diplomarbeit. Jede Variante erhielt eine eigene kurze Bezeichnung:

- **PTS/CORBA** bezeichnet die Variante, die nach dem ersten Realisierungsansatz (eine Java Applikation als Client; CORBA zur Kommunikation mit einem Server-Prozess) entwickelt wird.
- **PTS/RMI** steht für die Variante, deren Entwicklung den zweiten Realisierungsansatz (ein Java Applet als Client; Kommunikation mit einem Server-Prozess mittels RMI) umsetzt.
- Als **PTS/JSP** wird im folgenden die Variante des „Problem Tracking“ Systems bezeichnet, die als JavaServer Pages realisiert wird.

3.1 Vorgehensplanung

Als Basis für die Planung der Software-Entwicklung wurde das Wasserfallmodell dem Prozessmodell zugrunde gelegt. Jede Variante erhielt den folgenden Phasenplan:

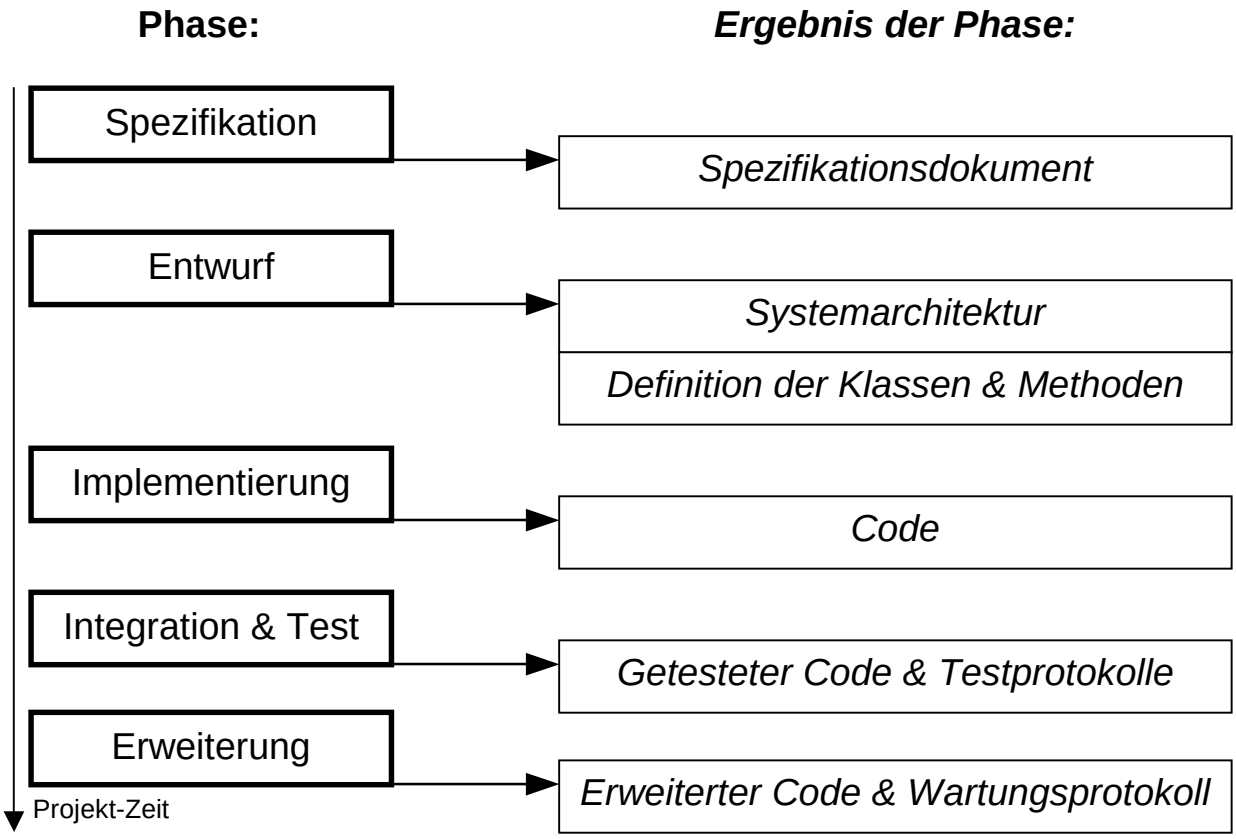


Abbildung 1 isolierter Phasenplan für die einzelnen Varianten

Da nur eine Person als Entwickler zur Verfügung stand, mussten die Phasen der drei voneinander unabhängigen Pläne für den Gesamtplan in eine sequentielle Abfolge gebracht werden. Die drei Spezifikationsphasen wurden im Gesamtplan zusammengefasst, da alle Varianten dieselbe Funktionalität zur Verfügung stellen sollten. Der Gesamtplan enthält außerdem Phasen für die Installation von Dritt-Software und für den abschließenden Performanztest.

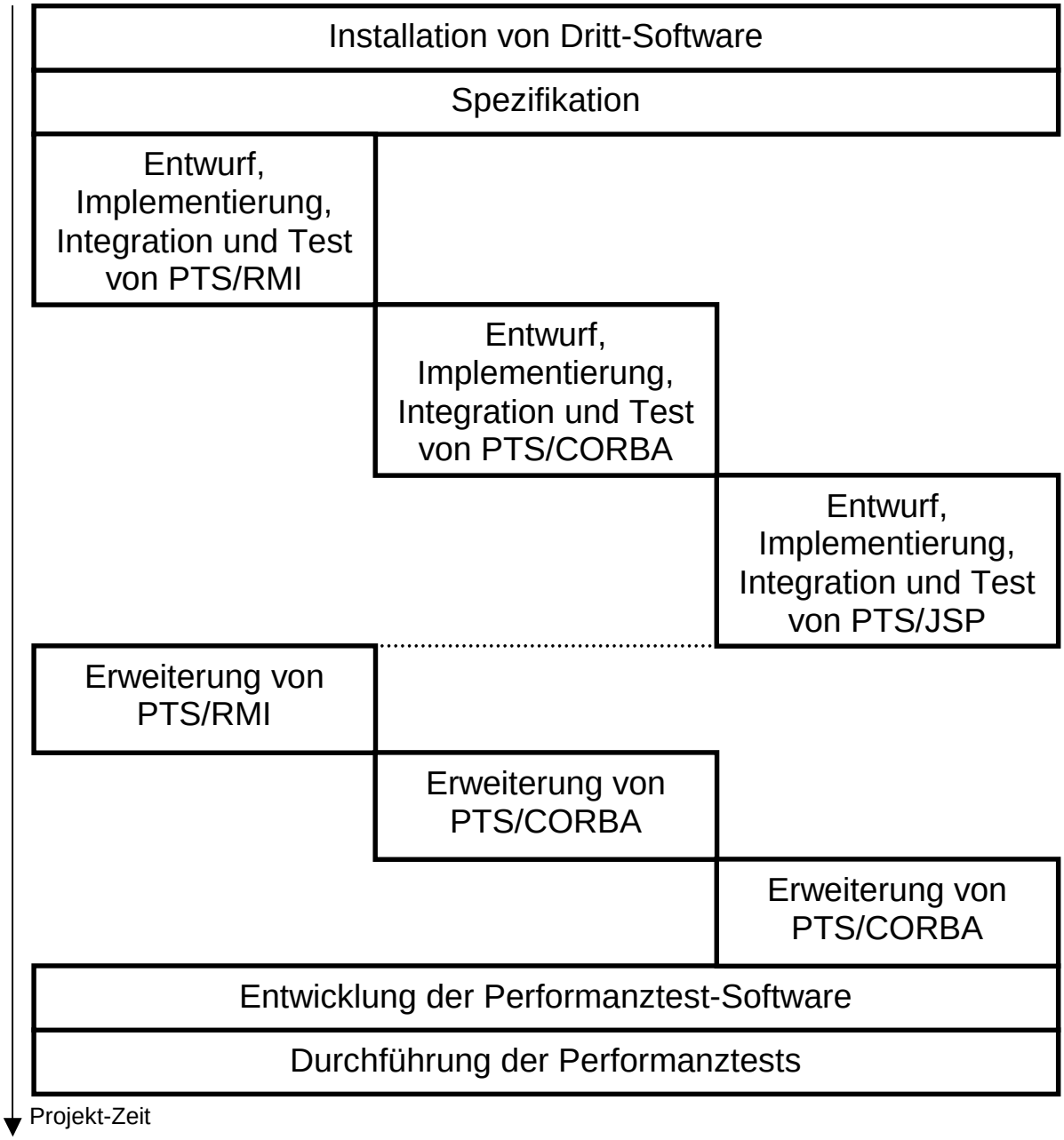


Abbildung 2 Gesamtplan

Die Reihenfolge PTS/RMI, PTS/CORBA, PTS/JSP wurde gewählt, um mehrere gleichzeitig auftretende Probleme bei der Entwicklung einer Variante zu vermeiden.

PTS/RMI nutzt nur Java Eigenschaften (zum Beispiel: RMI oder die Swing-Benutzungsoberfläche) und greift, wie alle anderen Varianten auch, auf eine Datenbank zu. Nach der Entwicklung von PTS/RMI können nun bekannte Java-Probleme und Probleme, die im Zusammenhang mit der Datenbank entstanden, bei der Entwicklung von PTS/CORBA und PTS/JSP umgangen werden. Bei der Entwicklung von PTS/CORBA liegt das Hauptrisiko deswegen hauptsächlich bei der Modellierung der Datenobjekte mit IDL (nach [OmgIDLJava]). Das Hauptrisiko bei der Entwicklung von PTS/JSP liegt dadurch allein bei der Benutzung der Dritt-Software. Sowohl die Anbindung des JSP-Containers [Tomcat] an den Webserver [Apache] als auch die Integration von PTS/JSP in den JSP-Container können bei der Entwicklung Verzögerungen verursachen.

In Wartungsprojekten (wie zum Beispiel [SP2 1999]) wird zumeist Code bearbeitet, der nicht von den Wartungs-Ingenieuren selbst stammt. Durch die zeitliche Trennung der Phasen Entwurf, Implementierung, Integration und Test von der Erweitungsphase einer Variante soll ein Mindestmaß an Wiedereinarbeitungsaufwand in Code und Architektur der Variante herbeigeführt werden, der diesen Umstand zumindest ansatzweise simuliert.

3.2 Installation von Dritt-Software

Es standen drei Rechnersysteme zur Verfügung, auf denen bereits [SuseLinux 7.0] installiert war:

- Eine IBM Netfinity Server Maschine mit zwei 550 MHz Pentium III Prozessoren, 1024 MB RAM und einem 100 Mbit Ethernet Netzwerkadapter. Auf dieser Maschine wurde die Software-Entwicklung durchgeführt.

- Ein IBM Personal Computer 300XL mit einem 266MHz Pentium II Prozessor, 192 MB RAM und einem 10 Mbit Token-Ring Netzwerkadapter.

- Ein IBM Thinkpad 765D mit einem 166MHz Pentium MMX Prozessor, 64 MB RAM und einem 10 Mbit Token-Ring Netzwerkadapter in Form einer PC-Card.

Auf der Server Maschine wurde nachträglich das Datenbanksystem [IBM_DB2], die SUN Java Entwicklungsumgebung [SunJavaSdk 1.3] und der HTML-Konverter [SunHtmlConv 1.3] installiert. Für die Entwicklung von PTS/JSP wurde darüber hinaus eine neuere Version des Webservers ([Apache 1.3.14]) installiert. Der Apache Werbserver in der Version 1.3.14 ist in der Lage, den JSP-Container [Tomcat 3.1.1] als ein Erweiterungsmodul zu integrieren. Der JSP-Container [Tomcat 3.1.1] wurde ebenfalls auf der Server Maschine installiert und in den Apache Webserver integriert.

Auf den beiden anderen Rechnersystemen wurde lediglich die Java Laufzeitumgebung [SunJavaRun 1.3] installiert und [Netscape 4.72] für die Nutzung des Java Plug-In konfiguriert.

3.3 Analyse und Spezifikation des „Problem Tracking“ Systems

Dieses Unterkapitel beschreibt Art und Umfang der Funktionalität, die von den drei Varianten prototypisch realisiert wird.

Die Analyse des Systems wurde gemeinsam mit Herrn Wohlmuth durchgeführt. Dabei wurde der „Problem Tracking“-Ablauf festgelegt und in [PTS Workflow] dokumentiert. Da auf eine existierende Datenbank ([PTS Datenbank]) zugegriffen werden sollte, gab Herr Wohlmuth deren SQL Datenbanktabellendefinition an.

3.3.1 Analyse des „Problem Tracking“-Workflows

Bei der Analyse des „Problem Tracking“-Workflows wurden drei fachliche Klassen identifiziert:

- Die Klasse „Problem“ modelliert die Menge der entdeckten und im „Problem Tracking“ System erfassten Probleme. Objekte der Klasse „Problem“ werden nachfolgend abgekürzt als *Problem* bezeichnet. *Probleme* besitzen einen Zustand und können durch Aktionen verändert werden.
- Die Klasse „Bearbeiter“ stellt die Menge der Personen dar, die Probleme mit Hilfe des „Problem Tracking“ Systems erfassen und durch Aktionen bearbeiten. Im folgenden werden „Bearbeiter“-Objekte kurz als *Bearbeiter* bezeichnet.
- Die Klasse „Problembereich“ steht für eine konstante Menge logischer Bereiche, denen ein *Problem* zugeordnet werden kann. Ein Problembereich enthält beispielsweise die Information, welcher *Bearbeiter* zuständig ist.

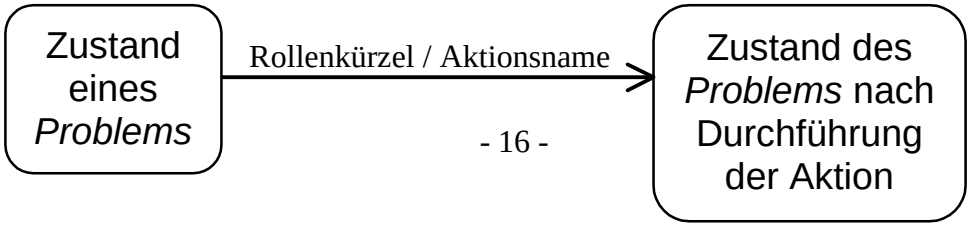
Eine Aktion wird von genau einem *Bearbeiter* durchgeführt. Ein *Bearbeiter* kann aus Sicht eines *Problems* zwei Rollen erfüllen:

- Problemersteller (kurz: E): Ein *Bearbeiter* erfüllt die Rolle des Problemerstellers, wenn der Bearbeiter die Aktion „create“ auf dem *Problem* durchgeführt und damit das *Problem* erstellt hat.
- Problemlöser (kurz: L): Ein *Bearbeiter* erfüllt die Rolle des Problemlösers, wenn dem *Bearbeiter* bei einer „create“- oder „reroute“-Aktion die Lösung des *Problems* zugeordnet wurde.

Welche Aktionen ein konkreter *Bearbeiter* auf einem *Problem* durchführen kann, ist abhängig davon, in welchem Zustand sich das *Problem* befindet und welche Rollen dieser *Bearbeiter* aus Sicht des *Problems* erfüllt.

Das nachfolgende Diagramm aus [PTS Workflow] veranschaulicht die möglichen Zustände eines *Problems* und durch welche Rollen dann jeweils Aktionen durchgeführt werden können. Dabei wird folgende Notation benutzt:

Die Knoten des Graphen sind die möglichen Zustände eines *Problems*. Die beschrifteten gerichteten Kanten sind Aktionen, die einen Zustandsübergang des betroffenen *Problems* zur Folge haben. Die Beschriftung der Kanten enthält die Kürzel der Rollennamen (E oder L), die zur Durchführung einer Aktion in einem Zustand berechtigt sind, und die Namen der Aktionen.



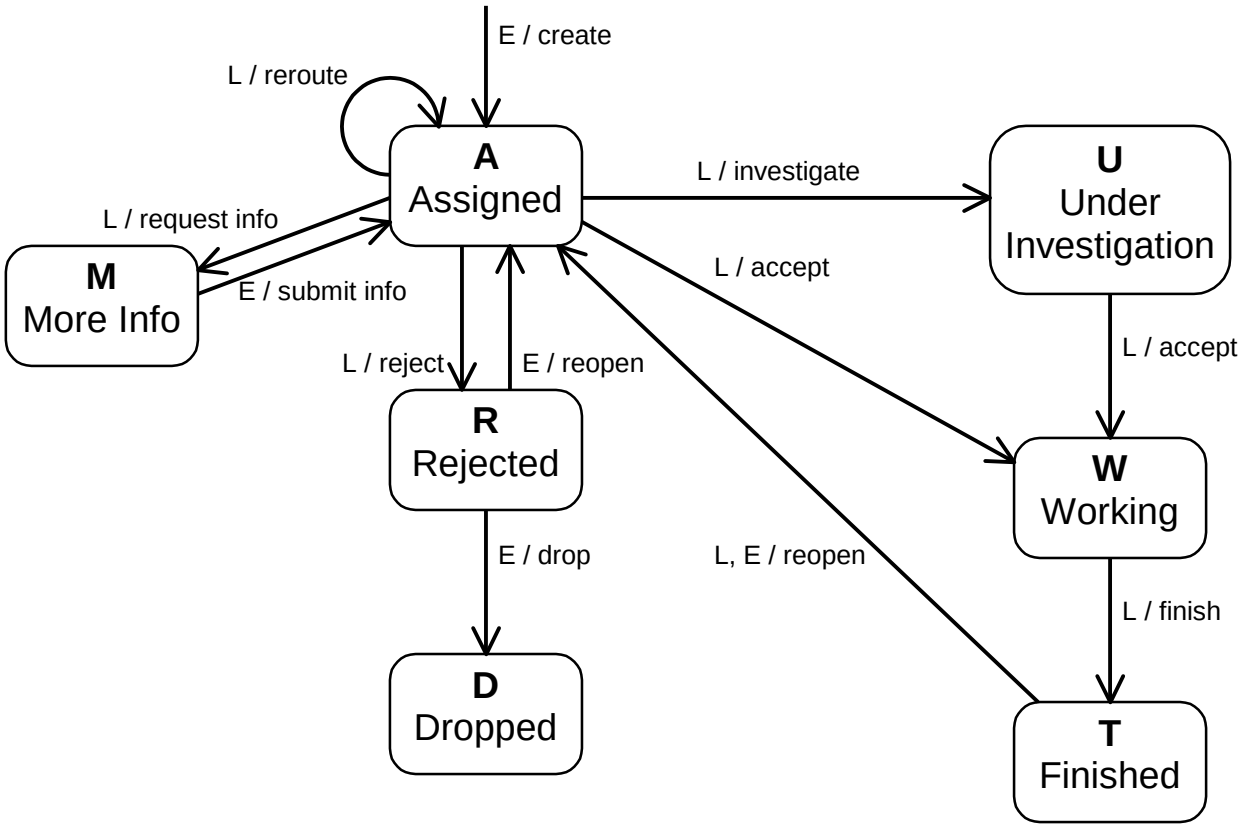


Abbildung 3 Zustandsübergangsdiagramm für *Problem*-Objekte

Die nachfolgende Tabelle beschreibt kurz alle Aktionen und die Informationen, die zur Durchführung einer Aktion notwendig sind.

Aktion	Beschreibung	notwendige Informationen
Create	Erstellung eines neuen *Problems*	alle obligatorischen Daten eines *Problems* (einschließlich Problemersteller und Problemlöser)
Reroute	Nachträgliche Zuordnung der Lösung eines *Problems* zu einem anderen Bearbeiter.	der Name des neuen Problemlösers
Investigate	Kennzeichnung eines *Problems*, das im Moment untersucht wird	
Accept	Kennzeichnung eines *Problems*, das momentan gelöst wird	
Reject	Ablehnung eines *Problems*	Grund für die Ablehnung
Reopen	erneute Aufforderung zur Lösung eines *Problems*	
Drop	endgültige Ablehnung eines *Problems*	
Request info	weitere Informationen zu einem *Problem* anfordern	
Submit info	weitere Informationen zu einem *Problem* übermitteln	ein kurzer Text, der weitere Informationen zum *Problem* enthält

3.3.2 Struktur der vorgegebenen Datenbank

Die Aussagen über die drei Varianten im Rahmen dieser Diplomarbeit sollen repräsentativ für eine bestimmte Klasse von Client-Server Informationssystemen sein. Um einen möglichst realitätsnahen Ablauf der Performanztests zu gewährleisten, wurden die Inhalte einiger Datenbanktabellen aus einem bestehenden Problemverfolgungssystem der IBM zur Verfügung gestellt. Dadurch ist sichergestellt, dass das Datenbanksystem auf einer realistischen Menge von Daten operieren muss (um zum Beispiel das Ergebnis von Selektionen zu ermitteln). Zusätzlich hat die vorgegebene Struktur der Datenbanktabellen (die Definition der Tabellenspalten) zur Folge, dass bei der Client-Server Kommunikation eine wirklichkeitsnahe Menge und Struktur von Daten übertragen wird. Außerdem hat die vorgegebene Struktur auch Auswirkungen auf den Umfang des Codes, denn Daten aus allen Spalten müssen durch die jeweilige Variante gelesen, geschrieben und bearbeitet werden.

Insgesamt wurden vier Tabellen für [PTS Datenbank] zur Verfügung gestellt. Die folgende Auflistung beschreibt aus Geheimhaltungsgründen nur die Spalten, deren Bedeutung für diese Diplomarbeit notwendig ist. Die nicht beschriebenen Spalten werden in den Varianten nicht bearbeitet, sind aber (da sie zum Umfang eines *Problem*-Objekts beitragen) für die Performanztests von Bedeutung.

1. Eine Tabelle, die alle *Problem*-Objekte enthält, „**`Problem`**“:

Spaltenname	SQL-Definition	Beschreibung
`ProblemId`	`Char(8) NOT NULL`	Primärschlüssel dieser Tabelle
`Problemtype`	`Char(1) NOT NULL`	
`Severity`	`Char(1) NOT NULL`	
`Significance`	`Char(8) NULL`	
`Description`	`Varchar(254) NOT NULL`	
`Status`	`Char(1) NOT NULL`	enthält den Zustand eines *Problems*
`Owninglocation`	`Integer NOT NULL`	
`Creationd`	`Date NOT NULL`	Erstellungsdatum
`Estimatedfinishd`	`Date NULL`	
`Actualfinishd`	`Date NULL`	
`Closingdate`	`Date NULL`	
`Pmhnumber`	`Integer NULL`	
`Planformnumber`	`Integer NULL`	
`Rejectreason`	`Char(3) NULL`	Grund für die Reject-Aktion
`Mcfrequired`	`Char(1) NOT NULL`	
`AssigneeuserID`	`Varchar(254) NOT NULL`	Primärschlüssel des Problemlösers
`AssproblemgrpID`	`Char(8) NULL`	
`OriginatoruserID`	`Varchar(254) NOT NULL`	Primärschlüssel des Problemerstellers
`Defectareacode`	`Char(8) NOT NULL`	Primärschlüssel des Problembereichs

Encrefcodes	Varchar(8) NULL	
Foundindriver	Char(8) NULL	
Foundinec	Varchar(125) NULL	
Foundinphase	Varchar(30) NULL	

2. Die Tabelle „**User**" enthält alle *Bearbeiter*-Objekte:

Spaltenname	SQL-Definition	Beschreibung
UserID	Varchar(254) NOT NULL	Primärschlüssel dieser Tabelle
Name	Varchar(30) NOT NULL	
Mailingaddress	Varchar(30) NOT NULL	
Location	Integer NOT NULL	
Authorization	Char(20) NOT NULL	
ManageruserID	Varchar(30) NOT NULL	
Role	Char(1) NULL	

3. In der Tabelle „**Defectarea**" sind alle Objekte der Klasse „Problembereich" enthalten:

Spaltenname	SQL-Definition	Beschreibung
Areacode	Char(8) NOT NULL	Primärschlüssel dieser Tabelle
ManageruserID	Varchar(254) NOT NULL	
Areadesctext	Varchar(254) NOT NULL	
FocalpointuserID	Varchar(254) NOT NULL	

4. Die Primärschlüssel von *Problem*-Objekten sind Zahlen. Die Tabelle „**Id_Table**" enthält den nächsten zu vergebenden *Problem*-Primärschlüssel. Dabei handelt es sich um den Wert des letzten vergebenen *Problem*-Primärschlüssels um 1 inkrementiert.

Spaltenname	SQL-Definition	Beschreibung
Table_Name	Varchar(30) NOT NULL	Name der Tabelle, deren Primärschlüssel gespeichert ist.
Next_Id	Char(8) NOT NULL	der nächste Primärschlüssel

3.3.3 Spezifikation

Zusätzlich zu den dokumentierten Erkenntnissen in [PTS Workflow] wurde anschließend ein Spezifikationsdokument erstellt. Die darin beschriebene Benutzungsoberfläche und die in Form von Use-Cases beschriebenen funktionalen Anforderungen sollen in allen Varianten umgesetzt werden. Deswegen ist die Spezifikation jedoch nur bis zu einem gewissen Grad präzisierbar. Details, die nur für eine bestimmte Variante angegeben werden können, verblieben dadurch ungeklärt.

In den Use-Cases werden Elemente der Benutzungsoberfläche für die Beschreibung der Interaktion zwischen System und Aktor herangezogen. Da die Spezifikation eine Java-Swing-Benutzungsoberfläche und eine HTML-Benutzungsoberfläche beschreibt, steht das Wort „Fenster" im folgenden sowohl für ein Java-Swing-Fenster als auch für den dargestellten HTML-Inhalt in einem Web-Browserfenster.

Die Spezifikation schreibt nicht vor, wie Eingabefelder und Schaltflächen bei jeder Variante realisiert werden müssen. Dem Benutzer muss lediglich die Möglichkeit geboten werden, im Falle von Eingabefeldern Daten einzugeben oder auszuwählen beziehungsweise im Falle von Schaltflächen diese eindeutig zu selektieren.

Elemente der Benutzungsoberfläche:

1. Anmeldefenster: Das Anmeldefenster ist das erste Fenster, das durch das System nach dem Start angezeigt wird. Es enthält je ein Eingabefeld für eine Benutzerkennung und ein Passwort. Außerdem werden drei Schaltflächen „Anmelden“, „Neuen Bearbeiter registrieren“ und „Abbrechen“ angezeigt.

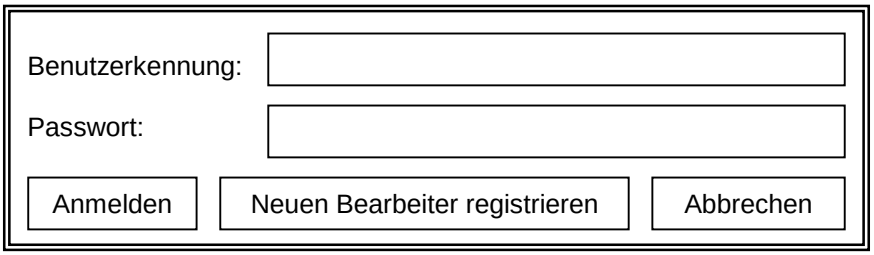


Abbildung 4 Skizze des Anmeldefensters

2. Bearbeiterregistrierungsfenster: Dieses Fenster dient der Eingabe von Daten für neue *Bearbeiter*. Das Bearbeiterregistrierungsfenster wird angezeigt, wenn der Benutzer im Anmeldefenster die Schaltfläche „Neuen Bearbeiter registrieren“ selektiert. Es enthält je ein Eingabefeld für jede Spalte der „**`User`**“-Datenbanktabelle und die zwei Schaltflächen „Bearbeiter erstellen“ und „Abbrechen“.

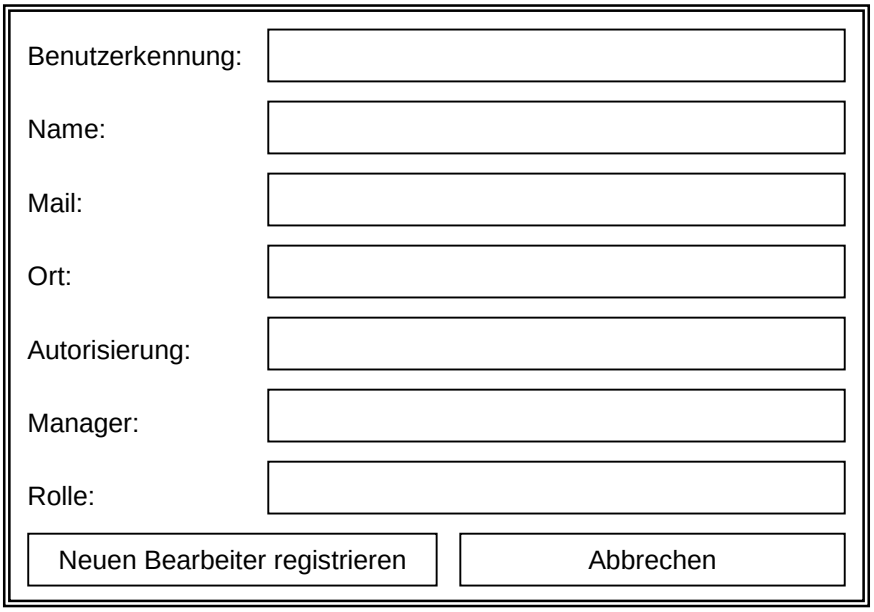


Abbildung 5 Skizze des Bearbeiterregistrierungsfensters

3. Hauptfenster: Das Hauptfenster bietet dem Benutzer die weitere Funktionalität des Systems zur Auswahl an. Das Hauptfenster wird angezeigt, nachdem der Benutzer sich über das Anmeldefenster mit einer Benutzerkennung und einem

gültigen Passwort angemeldet hat. Im Hauptfenster sind Schaltflächen für Eingabefelder enthalten (siehe Skizze). Die Eingabefelder werden lediglich für die Funktionalität der Schaltfläche „Benutzerdefinierte Suche nach Problemen“ zur Eingabe der Suchkriterien benötigt.

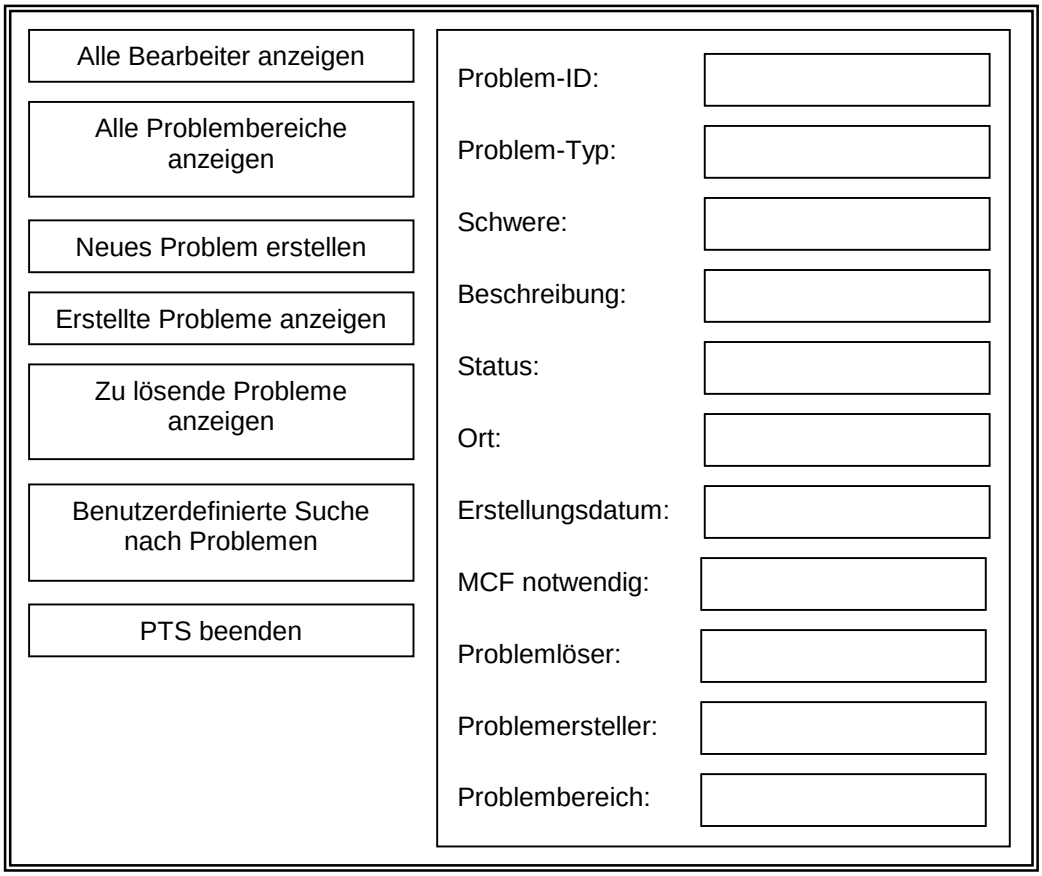


Abbildung 6 Skizze des Hauptfensters

4. Listenfenster: Für die Anzeige einer Menge von Daten in Listenform wird ein eigener Fenstertyp definiert. Listenfenster werden für die Anzeige von mehreren *Problemen*, *Benutzern* oder Problembereich-Objekten genutzt. Da in dieser Anwendung immer Objekte derselben Klasse in Listen dargestellt werden, erfolgt die Darstellung eines Listenausschnitts in Tabellenform (z.B. laut [Herczeg 1994] soll die Darstellung gleichartiger Informationsgruppen tabellenartig erfolgen).

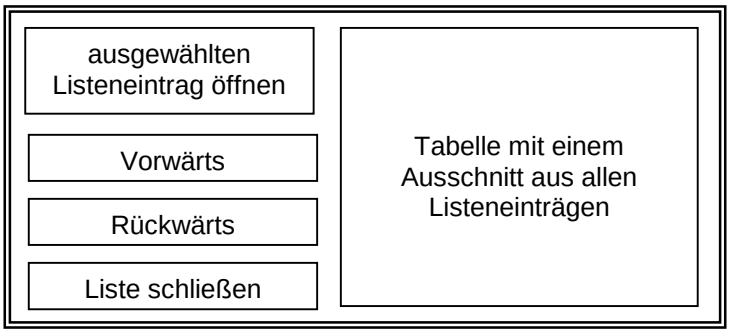


Abbildung 7 Skizze eines Listenfensters

Die Darstellung einer Liste in Abschnitten ist sinnvoll, damit Listen nicht im ganzen vom Server über das Netzwerk übertragen werden müssen. Andernfalls kann bei sehr großen Listen sowohl die Datenbank als auch das Netzwerk schon durch eine einzige Anfrage signifikant beansprucht oder gar blockiert werden.

5. Problemfenster: Das Problemfenster ermöglicht die Bearbeitung von *Problem*-Objekten nach dem in Kapitel 3.3.1 ermittelten „Problem Tracking“-Workflow. Die Daten des Problem-Objekts werden in Eingabefeldern angezeigt. Soll ein neues Problem-Objekt erstellt werden, sind die Eingabefelder leer. Alle Workflow-Aktionen und die Anzeige der „Submit Info“-Zusatzinformationen zu einem *Problem*-Objekt können durch Schaltflächen ausgelöst werden.

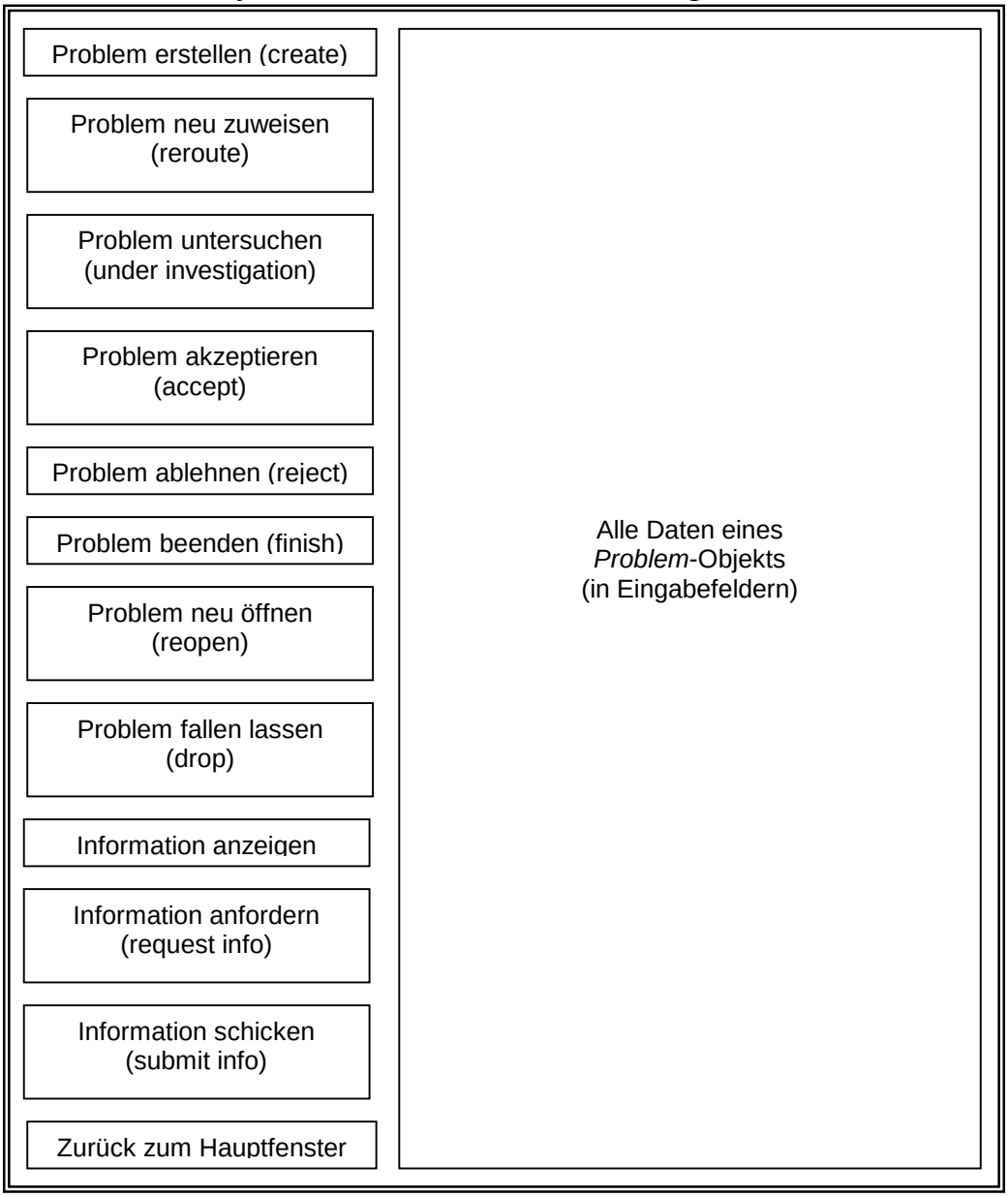


Abbildung 8 Skizze eines Problemfensters

Das Problemfenster wird angezeigt, wenn der Benutzer im Hauptfenster die Schaltfläche „Neues Problem erstellen“ selektiert, oder wenn in einem

Listenfenster, das eine Liste von *Problem*-Objekten darstellt, ein Listeneintrag ausgewählt und mit der Schaltfläche „ausgewählten Listeneintrag öffnen“ geöffnet wird. Im Problemfenster kann auch ein zusätzliches Eingabefeld für die Eingabe von notwendiger Information bei den Aktionen „Reroute“, „Reject“ und „Submit Info“ dem Benutzer zur Verfügung gestellt werden. Dann ist die Einblendung eines weiteren Fensters zur Eingabe dieser Information nicht notwendig.

Spezifizierte Use-Cases:

Die folgende Use-Case Auflistung beschreibt (aus Gründen der Lesbarkeit nicht in UML Notation, sondern sinngemäß informal) die in der Spezifikation dokumentierten funktionalen Anforderungen an das „Problem Tracking“ System. Use-Cases dienen in UML dazu, Interaktionen zwischen einem Aktor und einem System zu beschreiben.

1. Anmeldung eines Benutzers: Durch Eingabe von Benutzerkennung und korrektem Passwort eines *Bearbeiters* und anschließender Selektion der Schaltfläche „Anmelden“ im Anmeldefenster kann der Benutzer auf die weitere Funktionalität des Systems zugreifen. Alle Aktionen, die der Benutzer im weiteren Verlauf durchführt, werden dem *Bearbeiter* zugeordnet, mit dessen Kennung der Benutzer sich angemeldet hat. War die Anmeldung erfolgreich, wird das Hauptfenster, andernfalls eine Fehlermeldung und darauf folgend das Anmeldefenster angezeigt.

2. Neuen Bearbeiter registrieren: Der Benutzer selektiert im Anmeldefenster die Schaltfläche „Neuen Bearbeiter registrieren“. Daraufhin wird das Bearbeiterregistrierungsfenster angezeigt. Im Bearbeiterregistrierungsfenster gibt der Benutzer die Daten (Benutzerkennung, Name etc.) eines neuen *Bearbeiters* ein und selektiert die Schaltfläche „Neuen Bearbeiter registrieren“. Konnten die Daten des neuen *Bearbeiters* fehlerfrei der Datenbank des Systems hinzugefügt werden, wird das Anmeldefenster angezeigt. Andernfalls wird zuerst eine Fehlermeldung und anschließend das Bearbeiterregistrierungsfenster angezeigt (der Benutzer kann somit die Eingabe der Daten wiederholen).

3. Bearbeiterübersicht anfordern: Der Benutzer selektiert im Hauptfenster die Schaltfläche „Alle Bearbeiter anzeigen“. Daraufhin wird ein Listenfenster mit dem ersten Abschnitt einer nach Benutzerkennung aufsteigend sortierten Liste aller *Bearbeiter* angezeigt.

4. Problembereichsübersicht anfordern: Der Benutzer selektiert im Hauptfenster die Schaltfläche „Alle Problembereiche anzeigen“. Daraufhin wird ein Listenfenster mit dem ersten Listenausschnitt einer nach Bereichscode aufsteigend sortierten Liste aller Problembereich-Objekte angezeigt

5. Problemübersicht anfordern: Der Benutzer selektiert im Hauptfenster eine der folgenden Schaltflächen:
   a) „Erstellte Probleme anzeigen“
   b) „Zu lösende Probleme anzeigen“
   c) „Benutzerdefinierte Suche nach Problemen“

Daraufhin wird ein Listenfenster mit dem ersten Listenausschnitt einer nach Primärschlüssel aufsteigend sortierten Liste von *Problemen* angezeigt. Die Liste enthält eine Menge von *Problemen*. Diese Menge enthält alle *Probleme* aus der Datenbank, für die gilt:

a) Der Attributwert von `OriginatoruserID` ist gleich dem Primärschlüssel des *Bearbeiters*.
b) Der Attributwert von `AssigneeuserID` ist gleich dem Primärschlüssel des *Bearbeiters*.
c) Die Werte aller nicht leeren Eingabefelder im Hauptfenster sind gleich den entsprechenden Attributwerten des *Problems*.

Ist die Menge der *Probleme* leer, wird eine Fehlermeldung ausgegeben.

6. Problem erstellen: Der Benutzer selektiert im Hauptfenster die Schaltfläche „Neues Problem erstellen“. Daraufhin wird ein Problemfenster angezeigt, wobei die Eingabefelder zur Anzeige der Daten eines *Problems* leer sind. Der Benutzer trägt die Daten des neuen Problems in die Eingabefelder ein und betätigt die Schaltfläche „Problem erstellen (create)“. Konnte kein neues Problem in die Datenbank des Systems aufgenommen werden, oder wurde vom Benutzer eine andere Schaltfläche im Problemfenster betätigt, wird eine Fehlermeldung angezeigt.

7. Problem bearbeiten (nach den in [PTS Workflow] definierten Regeln): Der Benutzer markiert in einem Listenfenster, das eine Liste von *Problemen* anzeigt, das zu bearbeitende *Problem* und betätigt die Schaltfläche „ausgewählten Listeneintrag öffnen“. Daraufhin wird ein Problemfenster angezeigt, in dem die Eingabefelder bereits die Daten des ausgewählten *Problems* enthalten. Der Benutzer gibt alle notwendigen Informationen in die Eingabefelder ein und selektiert eine mit dem Namen einer Workflow-Aktion beschriftete Schaltfläche. War die Durchführung der Workflow-Aktion erfolgreich, wird eine Erfolgsmeldung, andernfalls eine Fehlermeldung angezeigt.

8. Information zu einem Problem anzeigen: Der Benutzer betätigt in einem Problemfenster, das die Daten eines existierenden *Problems* anzeigt, die Schaltfläche „Information anzeigen“. Daraufhin wird ein Listenfenster mit der zum *Problem* bei der Aktion „Submit info“ übermittelten Information angezeigt.

Identifikation zusätzlicher Klassen

Beim Spezifizieren der Funktionalität wurden zusätzliche zu den in der Analyse entdeckten Klassen „Problem“, „Bearbeiter“ und „Problembereich“ weitere fachliche Klassen identifiziert:

- Da die bei der Aktion „Submit info“ übermittelte Information genauso wie andere Objekte in einem Listenfenster dargestellt wird, ist es sinnvoll, eine eigene Klasse „Problemzusatzinformation“ für diese Information zu erstellen. Damit kann diese gemeinsame Eigenschaft genutzt werden.
- Die Klasse „Problemsuchmaske“ enthält die Bedingung, die für Probleme gilt, die in einer Problemübersicht angezeigt werden. Eine Nachfrage bei Herrn Wohlmuth ergab, dass dies alle Spalten der Tabelle „**`Problem`**“ sein sollen, deren Wert nicht auf NULL gesetzt werden kann.

- Die Klasse „Listenausschnitt“ dient dazu, einen Teil einer Liste von Objekten zu repräsentieren. Eine Liste von Objekten wird grundsätzlich in Abschnitten von Server zu Client übertragen.
- Objekte der Klasse „Listeneintrag“ sind Teil von Listenausschnittsobjekten und werden als Zeile einer Tabelle grafisch dargestellt. Ein Listeneintrag enthält die Daten eines Objekts, die in einer Tabellenzeile angezeigt werden sollen.

Festlegung der nachträglichen Erweiterung

Um bei der nachträglichen Erweiterung Auswirkungen in Datenbank, Anwendungslogik und Benutzungsoberfläche zu erzielen, wurde festgelegt, dass der Zustand „More Info“ einschließlich der Aktionen „Request info“ und „Submit info“ im Use-Case „Problem bearbeiten“ sowie das Use-Case „Information zu einem Problem anzeigen“ erst nachträglich zu implementieren sind (Vergleichsaspekt 3 der Aufgabenstellung).

In der Datenbank [PTS Datenbank] muss eine zusätzliche Tabelle angelegt werden, die einem Problem eine Menge von Zusatzinformationen zuordnet. Für den Zugriff auf diese neue Tabelle wird auf Seiten des Servers zusätzlicher Code in den Varianten benötigt.

Die Anwendungslogik ändert sich nicht im großen Umfang. Die Ergänzung eines weiteren Zustands und einer neuen fachlichen Klasse zu einem bestehendem Workflow ist eine realistische Wartungsaufgabe. Auswirkungen werden in den Teilen des Codes erwartet, die den Workflow abbilden und die neuen Server-Dienste zur Verfügung stellen werden.

Die Benutzungsoberfläche muss zum einen um die drei Schaltflächen „Information anzeigen“, „Information anfordern“, „Information schicken“ im Problemfenster und zum anderen um das Listenfenster, das die Problemzusatzinformation anzeigt, ergänzt werden. Bei der Erweiterung soll also eine neue fachliche Klasse (Problemzusatzinformation) in eine bestehende grafische Benutzungsoberfläche integriert und von dieser dargestellt werden.

Selbstkritik an der Vorgehensweise zur Definition der nachträglichen Erweiterung

Die Vorgehensweise, zuerst die gesamte Funktionalität zu spezifizieren und danach einen Teil daraus als nachträgliche Erweiterung festzulegen, ist rückblickend beurteilt ungünstig gewesen. Die Kenntnis der zukünftigen Erweiterung kann dazu führen, dass bei der Entwicklung der Varianten Strukturen entstehen, die von vorne herein auf die nachträgliche Integration der hinzukommenden Funktionalität ausgelegt sind. Dies kann im Vergleich zu einem Projekt, in dem diese Information nicht von Beginn an vorliegt, den Aufwand für die Entwicklung vergrößern und für die Erweiterung verringern.

Die Umsetzung der spezifizierten Funktionalität in Systemarchitektur und Entwurf wurde in einem möglichst nachvollziehbaren Verfahren bewerkstelligt. Dadurch soll ausgeschlossen werden, dass die Kenntnis der zukünftigen Erweiterung in die Entwicklung der Varianten mit einbezogen wird. Dennoch kann nicht ausgeschlossen werden, dass bei der Entwicklung die spätere Erweiterung unterbewusst bereits in Teilen vorbereitet wurde. Bei der Interpretation der Aufwandszahlen muss man sich dieser Tatsache bewusst sein.

3.3.4 Testplanung

Die funktionalen Tests der drei Varianten sollen im Black-Box-Testverfahren durchgeführt werden. Aus den spezifizierten funktionalen Anforderungen wurde eine Menge von Testfällen abgeleitet, die alle Funktionen der Varianten aktivieren sollte (Funktionsüberdeckung). Zusätzlich wurde zu jedem Testfall (T) in dieser Menge mindestens ein weiterer Testfall (FT) definiert, der eine Fehlersituation provoziert (Fehlertestfall). Funktionale Tests sind bei allen Varianten unmittelbar nach abgeschlossener Integration und nach Beendigung der nachträglichen Erweiterung durchgeführt worden.

Die nachfolgende Auflistung von Testfällen des Testplans enthält keine genauen Test-Eingaben oder Soll-Resultate, da in nahezu allen Testfällen entweder keine Meldung oder eine Erfolgsmeldung das Soll-Resultat ist. Dagegen ist das Soll-Resultat bei allen Fehlertestfällen die Ausgabe einer Fehlermeldung und keine Veränderung der Datenbankinhalte.

Test der Bearbeiterverwaltung

1. (T) Anmeldung mit der Benutzerkennung eines in der Datenbank gespeicherten *Bearbeiters*.
2. (FT) Anmeldung mit einer leeren Benutzerkennung.
3. (FT) Anmeldung mit einer nicht in der Datenbank gespeicherten Benutzerkennung eines *Bearbeiters*.
4. (T) Registrierung eines neuen, noch nicht in der Datenbank enthaltenen *Bearbeiters*.
5. (FT) Registrierung eines *Bearbeiters* mit einer bereits in der Datenbank gespeicherten Benutzerkennung.
6. (FT) Registrierung eines *Bearbeiters* mit unvollständigen Daten (ohne Namen).

Test des Listenfensters und der Suche nach Problemen

1. (T) Anzeige einer Liste aller *Bearbeiter*. Im Listenfenster wird solange die Schaltfläche „Vorwärts“ betätigt, bis keine weiteren *Bearbeiter* mehr angezeigt werden können.
2. (T) Anzeige einer Liste aller Problembereiche. Im Listenfenster wird solange die Schaltfläche „Vorwärts“ betätigt, bis keine weiteren Problembereiche mehr angezeigt werden können.
3. (T) Anzeige einer Liste aller *Probleme* nach Bearbeiter. Nach der Anmeldung mit einem bereits in der Datenbank enthaltenen Benutzerkennung wird im Hauptfenster eine Liste aller erstellten und aller zu lösenden *Probleme* angefordert. Von den dabei darzustellenden Listen von *Problemen* werden alle möglichen Listenausschnitte angefordert.
4. (T) Anzeige einer Liste von *Problemen* nach einer Suchmaske. Es werden 13 Suchanfragen gestellt. 11 mal wird genau ein Eingabefeld der Problemsuchmaske mit Daten versehen. Einmal werden alle Eingabefelder der Problemsuchmaske zugleich mit Daten versehen und ein weiteres mal wird eine Anfrage mit nur leeren Eingabefeldern gestellt (im letzten Fall werden alle *Probleme* in der Datenbank in der Ergebnisliste angezeigt).

Test des Workflows

1. (T) Erstellen eines neuen Problems.
2. (FT) Erstellen eines neuen Problems jedoch ohne Eingabe in einem Eingabefeld für eine obligatorische Spalte der „`Problem`“-Datenbanktabelle (Wert darf nicht NULL sein).
3. (FT) Erstellen eines neuen Problems mit ungültigen Eingabetypen (beispielsweise alphabetische Zeichen und Zeichenketten beim SQL-Datentyp INTEGER).
4. (T) Durchlaufen aller Zustände und Zustandsübergänge des Workflows. Nacheinander werden die folgenden Workflow-Aktionen auf einem neuen *Problem* druchgeführt: „Create“, „Reroute“, „Accept“, „Finish“, „Reopen“ (danach befindet sich das *Problem* wieder im Zustand „Assigned“), „Investigate“, „Accept“, „Finish“, „Reopen“ (erneut Zustand „Assigned“), „Reject“, „Reopen“, „Reject“, „Drop“ (Zustand „Dropped“).
5. (FT) In einem Zustand nicht mögliche Workflow-Aktionen durchführen: Im Zustand „Assigned“ die Aktionen „Drop“, „Reopen“, und „Finish“. Im Zustand „Working“ die Aktionen „Investigate“ und „Accept“.
6. (FT) Workflow-Aktion durchführen, ohne die richtige Rolle zu erfüllen: Als Problemersteller im Zustand „Assigned“ die Aktionen „Accept“ und „Investigate“. Als Problembearbeiter im Zustand „Rejected“ die Aktionen „Drop“ und „Reopen“.

Nach der Erweiterung der Varianten wurde für jede Variante ein Regressionstest durchgeführt. Alle Testfälle und Fehlertestfälle, die nach der Integration an den Varianten durchgeführt wurden, sind erneut durchlaufen worden. Da es keine Veränderung der damit getesteten Funktionalität gab, sind keine Unterschiede bei den Ist-Testresultaten zu erwarten. Die neu hinzugekommene Funktionalität erfordert die Aufnahme von zusätzlichen Testfällen:

Test des Listenfensters und der Suche nach Problemen

- (T) Anzeige der Liste von Problemzusatzinformationen eines Problems.

Test des Workflows

- (T) Durchlaufen des „More Info“ Zustands: Auf einem *Problem* im Zustand „Assigned“ werden die Workflow-Aktionen „Request Info“ und danach „Submit Info“ durchgeführt.
- (FT) Ungültige Aktionen durchführen: Im Zustand „More Info“ wird die Aktion „Accept“ und im Zustand „Working“ die Aktion „Request Info“ durchgeführt.

4 Entwicklung von PTS/RMI

In diesem Kapitel wird die softwaretechnische Umsetzung der spezifizierten Funktionalität in die Variante PTS/RMI beschrieben. Dabei wurde auf Erfahrungen aus vorangegangenen Softwareentwicklungsprojekten ([IP2 2000], [SP1 1998], [SP3 1999]) und Erkenntnisse aus Lehrveranstaltungen ([SE 1998], [KAOS 1999], [JavaKK 1999]) und der Literatur ([Züllighoven 1998]) zurückgegriffen.

4.1 Client-Server Kommunikation

Das folgende Diagramm zeigt die beteiligten Komponenten bei der Client-Server Kommunikation von PTS/RMI.

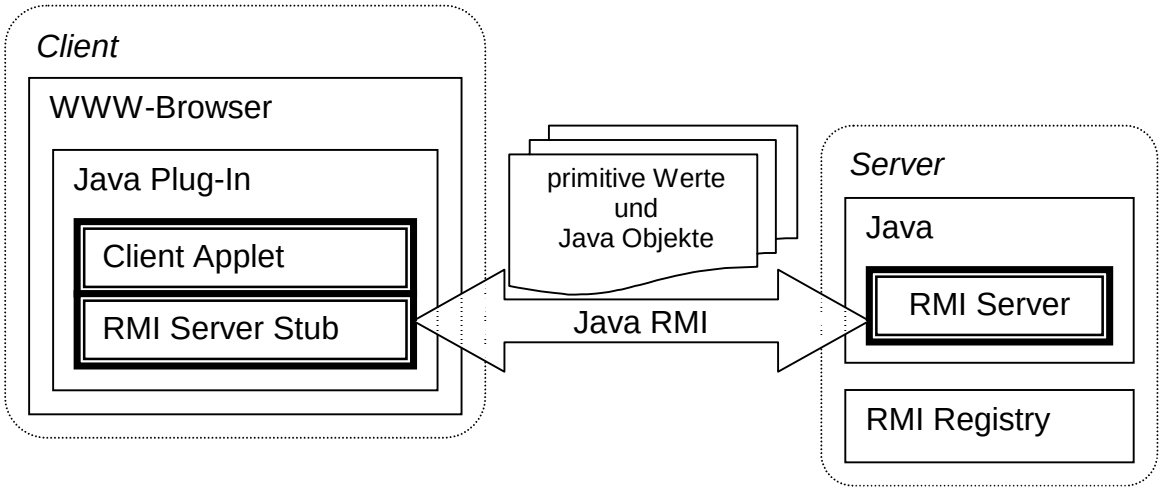


Abbildung 9 Client-Server Kommunikation bei PTS/RMI

Beim Server muss ein Java Programm gestartet werden, welches das RMI Server Objekt erstellt, initialisiert und bei der „RMI Registry“ anmeldet. Die „RMI Registry“ ist ein separates Programm aus der Java Entwicklungsumgebung, das die von der Server-Maschine aus nach außen sichtbaren Java Objekte verwaltet.

Zur Ausführung des Client Applets muss auf Client-Rechnern ein Web-Browser gestartet werden, für den ein Java Plug-In installiert ist. Das Client Applet ermittelt mit Hilfe der „RMI Registry“ auf der Server-Maschine die Objektreferenz auf das RMI Server Objekt. Innerhalb der JVM des Plug-Ins wird für die Kommunikation zwischen Client Applet und RMI Server Objekt implizit ein Stub-Objekt erzeugt. Zwischen Client Applet und RMI Server werden mittels Java RMI primitive Werte (wie z.B. `boolean` Werte) und Java Objekte übertragen. Von der Möglichkeit Objektreferenzen oder Java Klassen zu übertragen wird kein Gebrauch gemacht.

Stub-Objekte dienen als lokale Repräsentation oder „Proxy“ für entfernte Objekte und besitzen dieselbe Schnittstelle wie die entfernten Objekte. Botschaften (Methodenaufrufe) von lokalen Objekten an das entfernte Objekt werden an das Stub-Objekt weitergeleitet. Bei Java RMI führen Methodenaufrufe an Stub-Objekte die folgenden Tätigkeiten aus (aus [SunRmiSpec], Seite 15):

1. Das Stub-Objekt stellt eine Verbindung zur JVM, die das entfernte Objekt enthält, her.
2. Die Parameter des Methodenaufrufs werden in einer codierten Form zur entfernten JVM und damit zum entfernten Objekt übertragen („marshaling“).
3. Das Stub-Objekt wartet auf das Ergebnis des Methodenaufrufs an das entfernte Objekt.
4. Rückgabewert oder Ausnahme („exception“) des entfernten Methodenaufrufs werden gelesen und decodiert („unmarshaling“).
5. Der Rückgabewert (oder die Ausnahme) des entfernten Methodenaufrufs wird an das ursprünglich aufrufende lokale Objekt weitergeleitet.

Seit der Java Version 1.2 sind bei Java RMI (anders als bei „Remote Procedure Call“) analog zu den Stub-Objekten keine Skeleton Objekte auf Seiten des Servers mehr notwendig.

4.2 Entwurf

Das folgende Diagramm und die daran anschließenden Erläuterungen vermitteln einen groben Überblick über die Architektur des PTS/RMI Systems. Die einzelnen Teile der Systemarchitektur werden in weiteren Unterkapiteln im Detail beschrieben. Auf Seiten des Clients ist die Interaktion mit dem Benutzer und ein Großteil der Funktionalität untergebracht. Auf Seiten des Servers findet nur die Anbindung an die Datenbank statt.

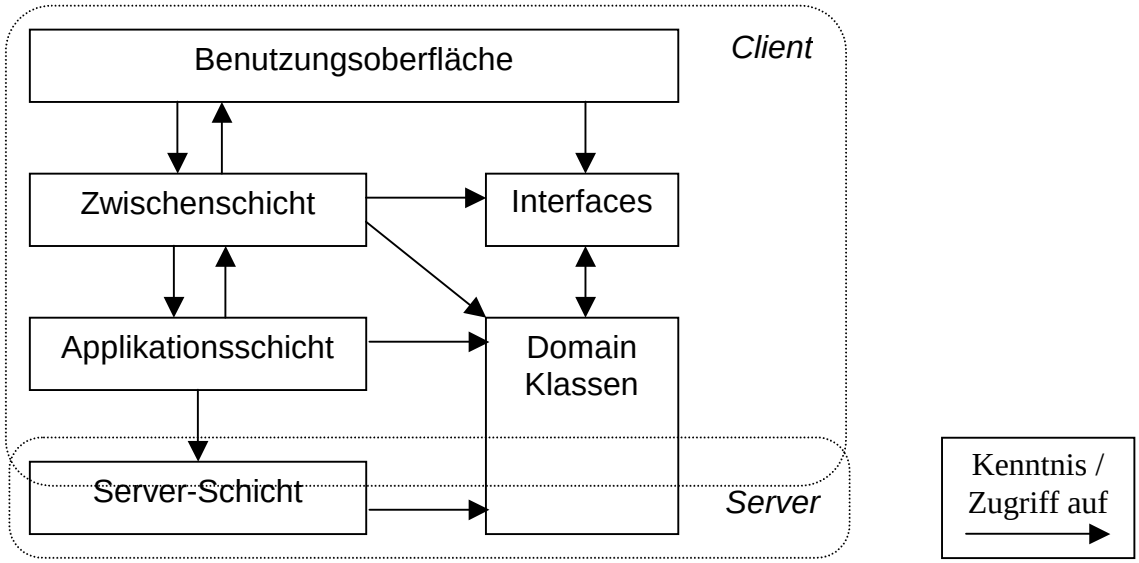


Abbildung 10 Systemarchitektur von PTS/RMI

- **Benutzungsoberfläche:** Die Klassen der Benutzungsoberfläche sind für die grafische Darstellung von PTS/RMI und die Interaktion mit dem Benutzer zuständig. Objekte der Benutzungsoberfläche warten entweder auf Benutzereingaben oder auf eine von der Zwischenschicht angestoßene Ausgabe von Daten. Ein Objekt der Benutzungsoberfläche kennt genau ein Objekt der Zwischenschicht, an das es die Benutzerinteraktion weiterleitet. Für die grafische Benutzungsoberfläche werden Java Swing Komponenten genutzt. Einige Java Swing Komponenten greifen dabei über Interfaces auf Domain Objekte zu.

- **Zwischenschicht:** Die Zwischenschicht ist das Bindeglied zwischen Applikationsschicht und Benutzungsoberfläche. Klassen der Zwischenschicht sind sowohl für die Weiterleitung und Interpretation von Benutzereingaben an die Applikationsschicht als auch für die Darstellung von Domain Objekten an der Benutzungsoberfläche zuständig. Ein Objekt der Zwischenschicht erzeugt in der Regel für die grafische Darstellung ein Objekt der Benutzungsoberfläche und hat Kenntnis von genau einem Objekt der Applikationsschicht.

- **Interfaces:** Einige Java Swing Komponenten (wie zum Beispiel die Tabellenklasse `JTable`) beziehen die von ihnen darzustellenden Daten aus Model-Objekten (im Sinne des Model-View-Controller Prinzips). Model-Objekte müssen bestimmte Swing Interfaces implementieren. Einige Domain Objekte von PTS/RMI implementieren diese Swing Interfaces, um direkt als Model-Objekt einer Swing Komponente dienen zu können. In der Interface-Schicht sind zusätzlich Klassen für die Konvertierung von Domain Objekten in Swing Objekte (und umgekehrt) enthalten.

- **Applikationsschicht:** Die Klassen der Applikationsschicht bilden die spezifizierte Funktionalität von PTS/RMI ab, ohne direkte Kenntnis von der grafischen Darstellung oder der Datenhaltung zu besitzen. Objekte der Applikationsschicht fordern Domain Objekte von Objekten der Server-Schicht an, bearbeiten diese Domain Objekte nach der spezifizierten Funktionalität und erzeugen oder informieren Objekte der Zwischenschicht (wenn sich beispielsweise Daten geändert haben). Objekte der Applikationsschicht können weitere Objekte der Applikationsschicht erzeugen.

- **Server-Schicht:** Die Server-Schicht stellt die Server-Dienstleistungen zur Verfügung und kapselt die Datenbankanbindung vom Rest des Systems. Zwischen Server-Schicht und Applikationsschicht findet die Kommunikation mit Java RMI statt. In der Server-Schicht befinden sich lediglich zwei Klassen: Ein Interface, über das entfernte Objekte auf das Server Objekt zugreifen können und die Implementierung des Interfaces - die Klasse des Server Objekts.

- **Domain Klassen:** Die Schicht „Domain Klassen“ enthält die in der Analyse und Spezifikation entdeckten Klassen. Mit Hilfe von Objekten dieser Klassen werden Daten zwischen Server-Schicht, Applikationsschicht und Zwischenschicht ausgetauscht. Objekte der Domain Klassen dienen als das gemeinsame Vokabular dieser drei Schichten. Objekte dieser Klassen werden per Java RMI zwischen Client und Server übertragen, weshalb Domain Klassen das Java Interface `Serializable` implementieren müssen.

4.2.1 Modellierung der Domain Klassen

Während der Analyse und bei der Betrachtung der vorgegebenen Datenbank wurden die drei Klassen „Problem“, „Bearbeiter“ und „Problembereich“ identifiziert. Das folgende UML-Klassendiagramm zeigt das Ergebnis der Modellierung dieser Klassen (alle

Bezeichner sind in englischer Sprache). Aus Gründen der Übersichtlichkeit werden die Vererbungsbeziehungen zum Interface `Serializable` nicht dargestellt.

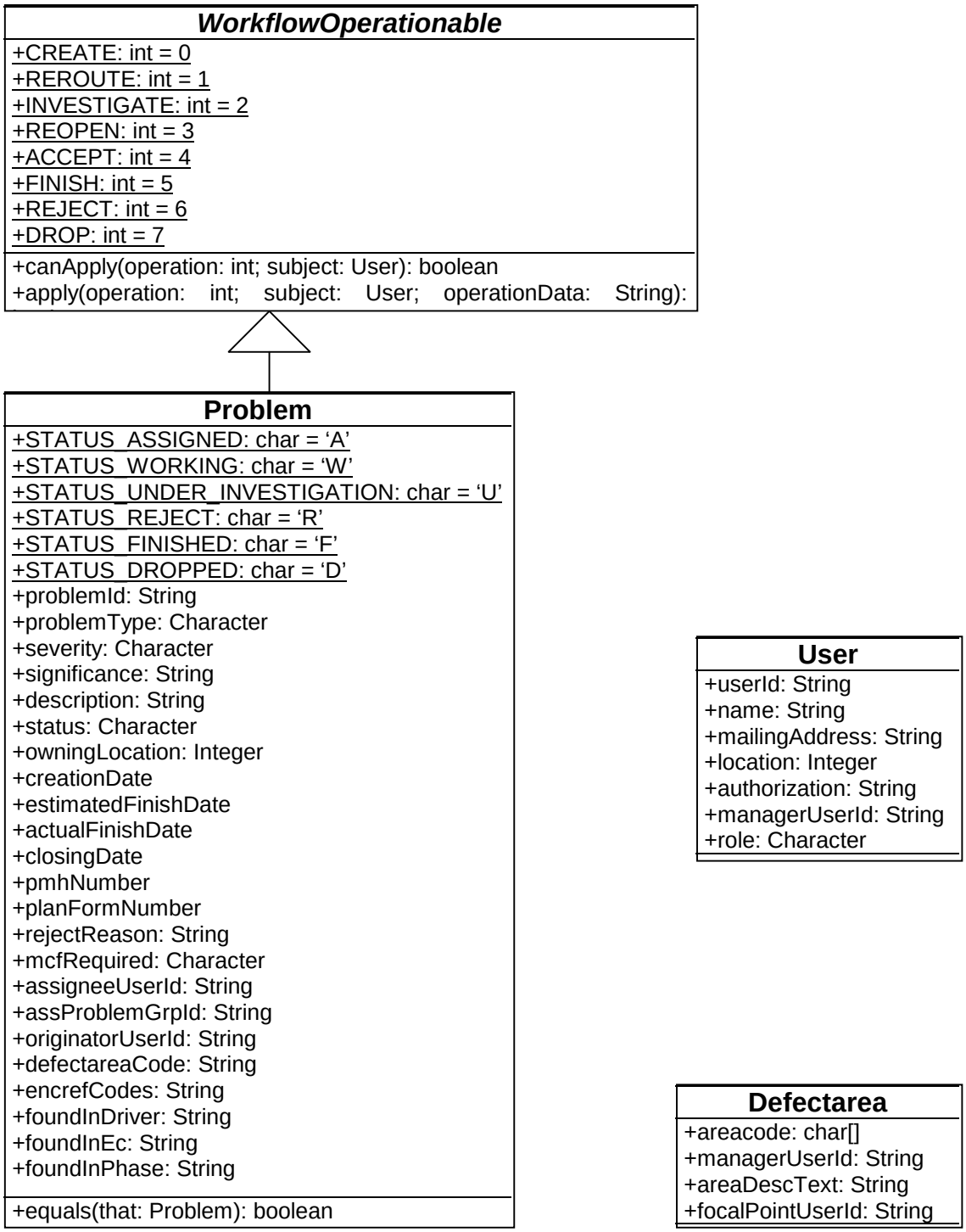
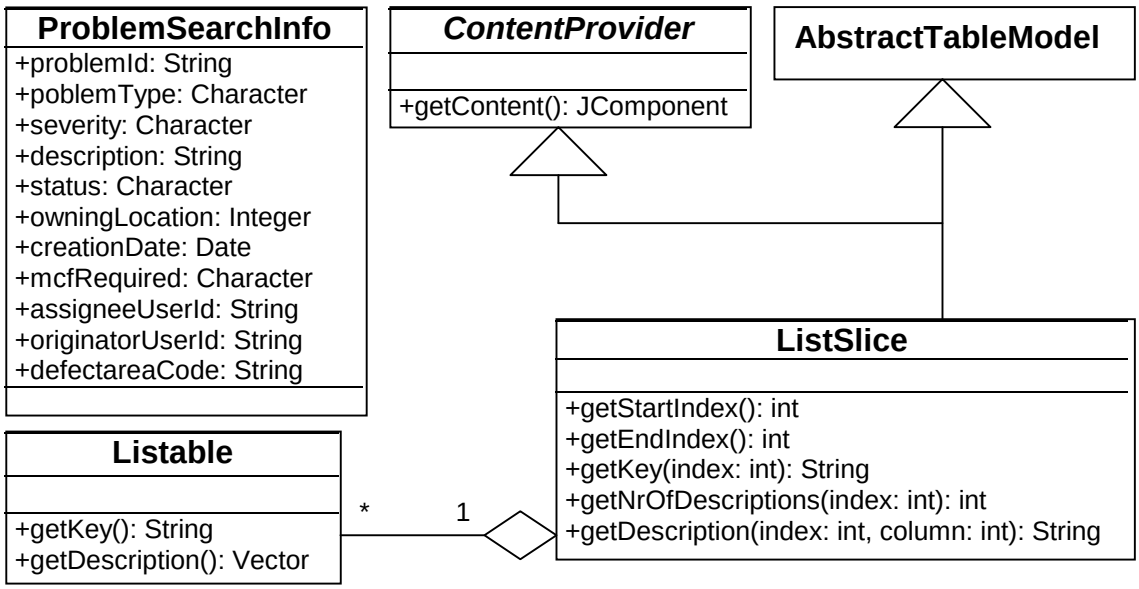


Abbildung 11 UML-Klassendiagramm der Klassen `WorkflowOperationable`, `Problem`, `User` und `Defectarea`

Aus der Struktur von [PTS Datenbank] wurden die öffentlichen Attribute der Klassen hergeleitet. Zur Abbildung der Null-Zeiger Semantik wurden für die Attribute die „Wrapper"-Klassen und nicht die primitiven Datentypen gewählt (also z.B. `Integer`

anstelle von `int`). Aus dem Verhalten eines *Problems* in [PTS Workflow] wurde das Interface `WorkflowOperationable` erstellt. Das Interface enthält für jede Workflow-Aktion eine konstante Klassenvariable („static field“) und zwei Methoden. Die eine Methode prüft, ob eine Workflow-Aktion durchgeführt werden kann, die andere führt eine Workflow-Aktion durch.



Nach der Spezifikation der Funktionalität wurde die Notwendigkeit für einige zusätzliche Klassen deutlich. Außer der Klasse „Problemzusatzinformation“, die erst bei der Erweiterung in PTS/RMI aufgenommen wird, zeigt das nachfolgende UML-Klassendiagramm deren Modellierung (auch hier wird die Vererbung vom Interface `Serializable` nicht dargestellt).

Abbildung 12 UML-Klassendiagramm der Domain Klassen `ProblemSearchInfo`, `ListSlice` und `Listable`

Die Klasse `ProblemSearchInfo` ist ein Ausschnitt der Klasse `Problem`, denn sie enthält ebenfalls Spalten der Problem-Datenbanktabelle, jedoch nur die Spalten, deren Werte nicht auf NULL gesetzt werden können.

Die Klasse `ListSlice` erbt von den zwei Interface-Klassen `ContentProvider` und `AbstractTableModel`. Die Klasse `AbstractTableModel` ist Teil des Java Swing Pakets. Objekte der `AbstractTableModel` Klasse können einer Swing Tabelle (Klasse `JTable`) die darzustellenden Daten zur Verfügung stellen. Das Interface `ContentProvider` ist eine PTS/RMI spezifische Schnittstelle, mit deren Hilfe die Benutzungsoberfläche auf die grafische Darstellung eines Objekts zugreifen kann.

4.2.2 Modellierung der Server-Schicht

In der Server-Schicht wird die Server-Funktionalität realisiert. Alle Dienste, auf die Clients zurückgreifen können, werden durch das Interface `ServerAccess` zur Verfügung gestellt. Das Interface `ServerAccess` erbt von der Klasse `Remote` im

`java.rmi` Paket und kennzeichnet sich dadurch als ein Interface, auf das mit Java RMI von anderen „Java Virtual Machines“ aus zugegriffen werden kann.
Die Implementierung des `ServerAccess` Interfaces erfolgt in der Klasse `PtsRmiServer`. Die Klasse `PtsRmiServer` erbt von der Klasse `UnicastRemoteObject` des Pakets `java.server.rmi` das Verhalten der Methoden `equals`, `hashCode` und `toString` und benutzt den `UnicastRemoteObject`-Konstruktor, um Objekte der Klasse für Client-Anfragen zugänglich zu machen.

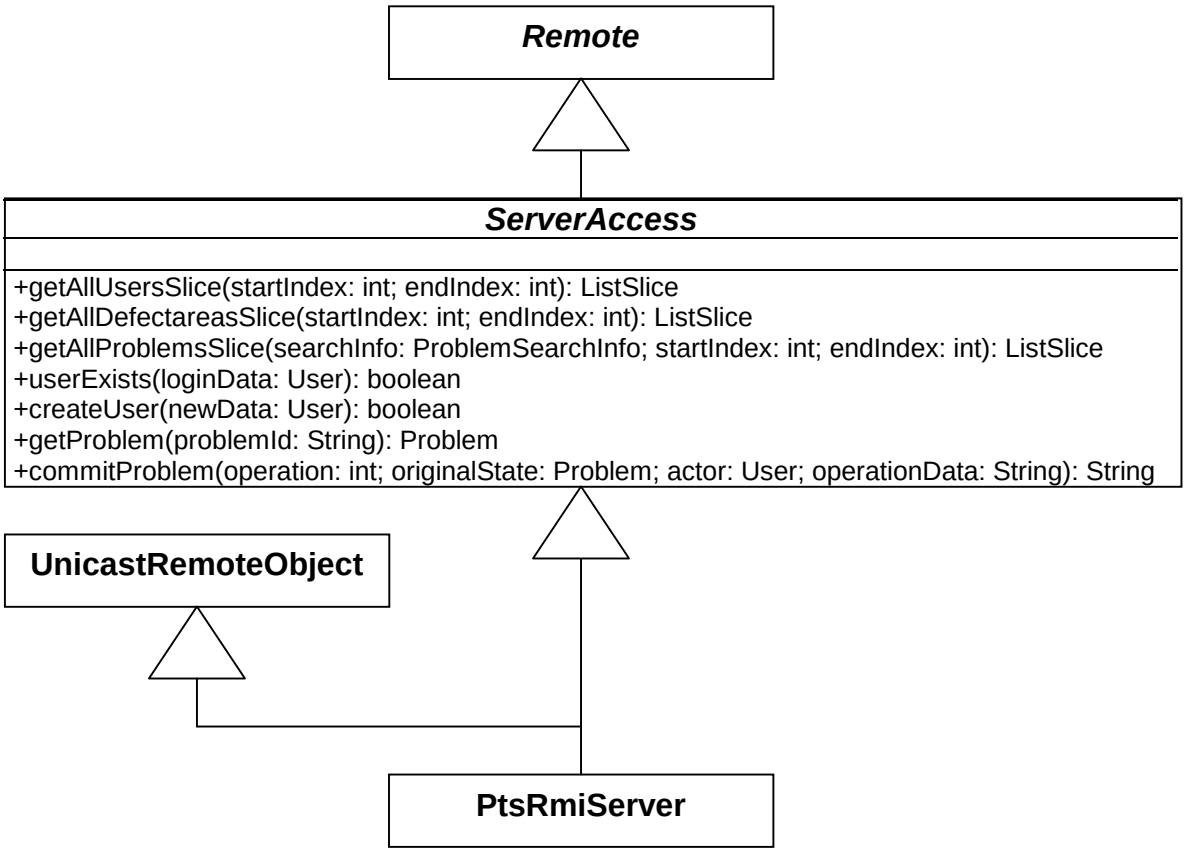


Abbildung 13 UML-Klassendiagramm der Klassen `ServerAccess` und `PtsRmiServer`

Objekte der `PtsRmiServer` Klasse kommunizieren mittels primitiver Werte und mit Hilfe von Objekten der Domain Klassen `ListSlice`, `ProblemSearchInfo`, `Problem` und `User` mit Client Objekten. Der Parameter `operation` der `commitProblem`-Methode ist für die Konstanten der Klasse `WorkflowOperationable`, die jeweils einer Workflow-Aktion entsprechen, vorgesehen.

4.2.3 Modellierung des Clients

Dieses Kapitel beschreibt die Zusammenarbeit der Applikationsschicht mit der Zwischenschicht und der Benutzungsoberfläche. Das Konzept der „Trennung von Funktion und Interaktion“ aus [Züllighoven 1998] hat die Definition der Schichten und Klassen maßgeblich geprägt. An einem Beispiel wird die Zusammenarbeit der drei Schichten bei Benutzerinteraktionen demonstriert.

Strukturierung der Funktionalität

Die Unterteilung der spezifizierten Funktionalität in kleinere Teile und deren Zusammenwirken erfolgt in den Klassen der Applikationsschicht. Das folgende Diagramm zeigt die baumähnlichen Beziehungen zwischen den Klassen der Applikationsschicht (Vererbungsbeziehungen werden in UML-Notation dargestellt).

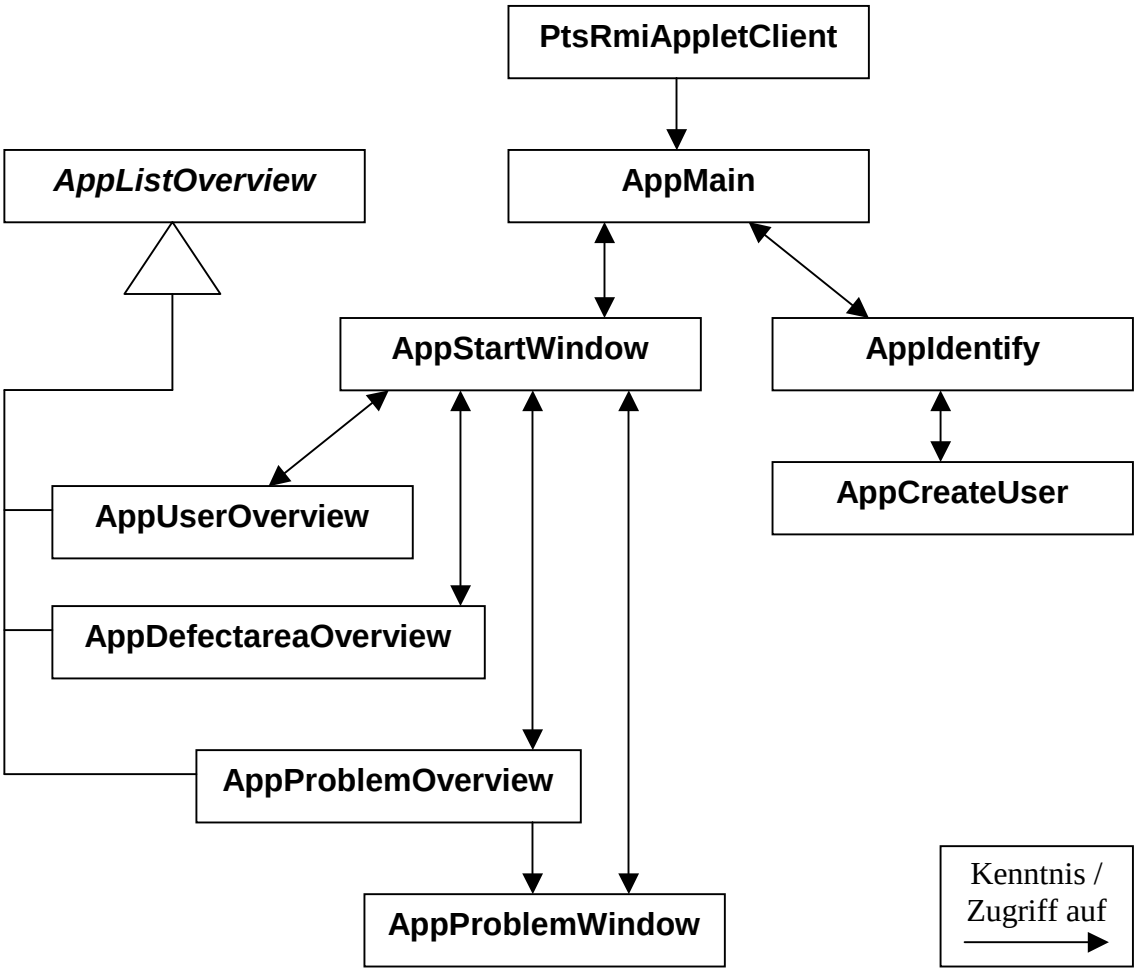


Abbildung 14 Beziehungen zwischen den Klassen der Applikationsschicht

Funktion und Beziehungen der Klassen in der Applikationsschicht sind aus den Elementen der Benutzungsoberfläche und deren Beziehungen untereinander abgeleitet. Das Aktivieren und Deaktivieren der Benutzungsoberfläche eines Objekts aus der Applikationsschicht wird durch die Methode `setEnabled(boolean enabled)` ermöglicht, die alle Klassen mit dem Präfix „`App`“ zur Verfügung stellen. Allen Objekten der Applikationsschicht wird eine Referenz auf das Server-Objekt weitergegeben. Sie können dadurch unabhängig voneinander Dienste des Servers in Anspruch nehmen.

- **`PtsRmiAppletClient`** ist die Klasse des in der HTML-Seite eingebundenen Applets. Das Applet zeigt eine Schaltfläche an, mit der PTS/RMI gestartet werden kann. Bei Betätigung dieser Schaltfläche ermittelt das Applet eine Referenz auf das Server-Objekt und gibt diese im Konstruktor an ein neu erzeugtes `AppMain` Objekt weiter und ruft dessen `run`-Methode auf.

- **`AppMain`** ist eine Klasse, für die kein Gegenstück in der grafischen Benutzungsoberfläche existiert (der Aufruf ihrer `setEnabled`-Methode bewirkt nichts). Sie ist für Start und Ende von PTS/RMI zuständig. Die von der Applet-Klasse `PtsRMiAppletClient` aufgerufene `run`-Methode erzeugt ein `AppIdentify`-Objekt und aktiviert dessen Benutzungsoberfläche. Nach der Anmeldung des Benutzers wird ein `AppStartWindow`-Objekt erzeugt und aktiviert.

- **`AppIdentify`** stellt das Anmeldefenster und dazugehörige Fehlermeldungen dar. Meldet sich der Benutzer mit Benutzerkennung und Passwort an, wird das erzeugende `AppMain`-Objekt benachrichtigt. Soll eine neuer Bearbeiter registriert werden, wird ein neues Objekt der Klasse `AppCreateUser` erzeugt und dessen Benutzungsoberfläche aktiviert, während die eigene Benutzungsoberfläche deaktiviert wird.

- Die Klasse **`AppCreateUser`** zeigt das Bearbeiterregistrierungsfenster an und realisiert die darin angebotene Funktionalität. Selektiert der Benutzer eines der beiden Schaltflächen im Bearbeiterregistrierungsfenster, wird das Fenster geschlossen und das erzeugende `AppIdentify`-Objekt benachrichtigt.

- Die Klasse **`AppStartWindow`** ist für Darstellung und Funktionalität des Hauptfensters zuständig. Soll eine Liste von *Bearbeitern*, Problembereichobjekten oder *Problemen* angezeigt werden, so werden Objekte der entsprechenden Unterklassen von `AppListOverview` erzeugt und aktiviert. Soll ein neues *Problem* erstellt werden, wird ein Objekt der Klasse `AppProblemWindow` erzeugt und aktiviert. Soll PTS/RMI beendet werden, wird das erzeugende `AppMain`-Objekt davon benachrichtigt.

- Das Interface **`AppListOverview`** fasst die Gemeinsamkeiten der drei Klassen `AppProblemOverview`, `AppDefectareaOverview` und `AppUserOverview` zusammen. Über diese gemeinsame Schnittstelle kann eine einzige Klasse der Zwischenschicht `ControlListOverview` die Benutzerinteraktion koordinieren.

- Die Klassen **`AppProblemOverview`**, **`AppDefectareaOverview`** und **`AppUserOverview`** greifen je auf einen anderen Server-Dienst zu und unterscheiden sich durch den Server-Zugriff und die Daten, die sie dafür zum Server-Objekt übermitteln müssen, voneinander. Die Benutzungsoberfläche der drei Klassen ist ein Listenfenster. Wird dieses geschlossen, benachrichtigen Objekte dieser drei Klassen das erzeugende `AppStartWindow`-Objekt. Objekte der Klasse `AppProblemOverview` erzeugen und aktivieren bei der Selektion der Schaltfläche „ausgewählten Listeneintrag öffnen“ ein Objekt der Klasse `AppProblemWindow`. Das *Problem*, das dem ausgewählten Listeneintrag entspricht, wird im Konstruktor des `AppProblemWindow`-Objekts als das zu bearbeitende *Problem* übergeben.

- In der Klasse **`AppProblemWindow`** wird die Funktionalität des Problemfensters realisiert. Bei der Instanziierung wird im Konstruktor das zu bearbeitende *Problem* und eine Referenz auf ein `AppStartWindow`-Objekt übergeben. Wird kein *Problem* (NULL-Zeiger) angegeben, so interpretiert das `AppProblemWindow`-Objekt dies als den Wunsch, in diesem Problemfenster ein neues *Problem* erstellen zu wollen. Nach erfolgreicher Bearbeitung und Übertragung des *Problems* zum Server-Objekt wird das Problemfenster geschlossen und das `AppStartWindow`-Objekt davon benachrichtigt.

Durch die Beziehungen der Klassen der Applikationsschicht untereinander soll eine einfache, aus der Spezifikation abgeleitete Struktur geschaffen werden. Jede Klasse der Applikationsschicht erfüllt eine klar begrenzte Teilmenge der spezifizierten Funktionalität. Botschaften von und zu Objekten anderer Klassen der Applikationsschicht beschränken sich auf die Übergänge von einem Fenster der Benutzungsoberfläche (und der damit verbundenen Menge an Funktionalität) zu einem anderen Fenster.

Trennung von Funktion und Interaktion

Das Konzept zur „Trennung von Funktion und Interaktion“ ist Teil eines im Buch [Züllighoven 1998] vorgestellten Ansatzes zur Konstruktion objektorientierter Systeme. Zwar wurde dieser Ansatz nicht für die Entwicklung von PTS/RMI zugrunde gelegt, aber vor allem dessen Konzept zur „Trennung von Funktion und Interaktion“ hat den Entwurf von PTS/RMI entscheidend geprägt. Dieses Konzept beschreibt, wie Werkzeuge (die Teile eines Systems, die fachliche Objekte bearbeiten) aufgebaut sein sollten (Zitat aus [Züllighoven 1998] Seite 236):

“Unterteile ein Werkzeug grundsätzlich in eine Funktionskomponente (FK) und in eine Interaktionskomponente (IAK). Die FK realisiert die Funktion eines Werkzeugs, die IAK die Interaktion. Kombiniere FK und IAK so, dass eine IAK ihre FK kennt und benutzt, während die FK möglichst keinerlei Kenntnis von ihrer IAK haben soll. Dadurch wird die primäre Kontrollflussrichtung vom Benutzer in das System hinein abgebildet und die Entkopplung der IAK von der FK möglich.”

Das Konzept wurde auch schon für den Entwurf des Java-Systems von [IP2 2000] angewendet. Die dabei entstandenen Strukturen wurden in PTS/RMI übernommen. Die Klassen der Applikationsschicht stellen im Sinne des obigen Zitats die fachlichen Komponenten eines Werkzeugs dar. Sie bearbeiten Objekte der Domain Klassen.

Die Klassen der Zwischenschicht sind die Interaktionskomponenten zu den funktionalen Komponenten. Die Klassen der Zwischenschicht erfüllen dazu die Java Interfaces `ActionListener` und `WindowListener`, um Ereignisse der Java Swing Benutzungsoberfläche als Empfänger („event listener“) erhalten zu können. In der Zwischenschicht werden Java Swing Ereignisse interpretiert und in der Regel einen Methodenaufruf der zugehörigen fachlichen Komponente ausgelöst.

Die Klassen der Benutzungsoberfläche entkoppeln die Interaktionskomponenten schließlich von den Eigentümlichkeiten einzelner Java Swing Klassen. Das

automatische Layout der einzelnen Java Swing Objekte mit Hilfe von Layout-Managern und die dafür nötige Konfiguration und Gruppierung der Java Swing Objekte wird in dieser Schicht realisiert.

Zu einem Objekt der Applikationsschicht mit einer grafischen Benutzungsoberfläche gehört also in jedem Falle ein Objekt der Zwischenschicht und zu diesem wiederum ein Objekt der Benutzungsoberfläche. Das folgende Diagramm und die darauf folgenden Erläuterungen illustrieren die Aufgabenteilung, zwischen Objekten der Applikationsschicht, Objekten der Zwischenschicht und Objekten der Benutzungsoberfläche am Beispiel des Hauptfensters.

Instanziierung der am Hauptfenster beteiligten Objekte

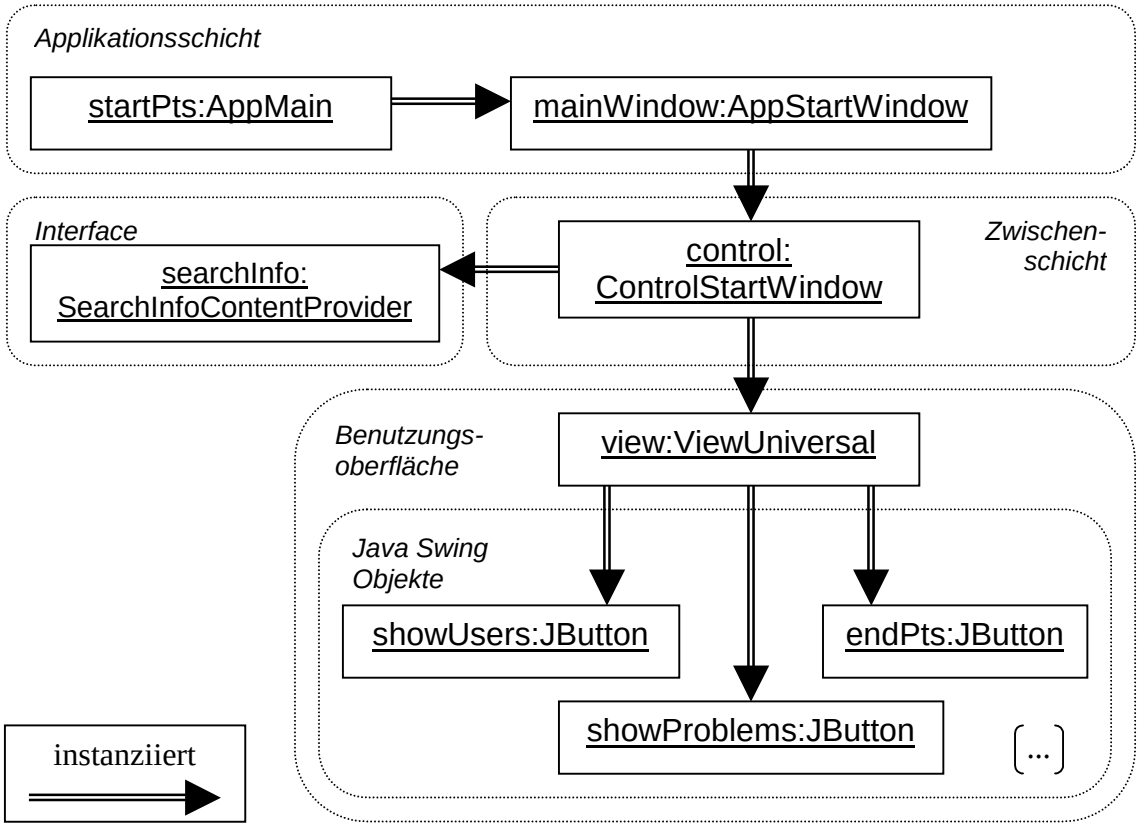


Abbildung 15 Instanziierung der Objekte des Hauptfensters

Die Instanziierung aller der Applikationsschicht untergeordneten Objekte erfolgt in den Konstruktoren der jeweils übergeordneten Objekte um sicherzustellen, dass untergeordnete Objekte für alle späteren Methodenaufrufe zur Verfügung stehen. Die untergeordneten Objekte werden bei ihrer Instanziierung für die weitere Zusammenarbeit mit einem übergeordneten Objekt konfiguriert:

- Das **`mainWindow`**`-Objekt` gibt eine Referenz auf sich selbst an das `control`-Objekt weiter. Das `control`-Objekt kann damit zukünftig Botschaften der Benutzungsoberfläche an das `mainWindow`-Objekt weiterleiten.

- Das **`control`**-Objekt gibt eine Referenz auf sich selbst an das `view`-Objekt weiter. Das Interface-Objekt `searchInfo` wird erzeugt, und dessen grafische Darstellung an das `view`-Objekt weitergegeben.

- Das **`view`**-Objekt erstellt die sichtbaren Java Swing Objekte, darunter die den spezifizierten Schaltflächen entsprechenden Objekte der Klasse `JButton`: `showUsers`, `showProblems` und `endPts`. Als Ereignisempfänger („event listener“) wird das `control`-Objekt angegeben, das für die Klasse `JButton` das Interface `ActionListener` erfüllt. Die grafische Darstellung des `searchInfo`-Objekts wird ohne Kenntnis des genauen Inhalts im rechten Teil des Fensters dargestellt.

- Das **`searchInfo`**-Objekt ist ein Objekt der Interface-Schicht. Als grafische Darstellung besitzt es die Menge an Eingabefeldern, die für die Eingabe der Daten eines Problemsuchmasken-Objekts notwendig sind. Aus den Daten in den Eingabefeldern kann das `searchInfo`-Objekt ein neues Objekt der Klasse `ProblemSearchInfo` erzeugen.

Das Zusammenwirken von Schichten der PTS/RMI Architektur und die unterschiedlichen Zuständigkeiten der einzelnen Objekte werden nun am Beispiel der „benutzerdefinierten Suche nach Problemen“ demonstriert:

1. Der Benutzer betätigt die Schaltfläche mit der Aufschrift „Benutzerdefinierte Suche nach Problemen“.
2. Das `showProblems`-Objekt zeigt die Betätigung durch eine sichtbare Farbveränderung der dargestellten Schaltfläche an.
3. Das `showProblems`-Objekt schickt ein Ereignisobjekt an alle Ereignisempfänger und damit an das `control`-Objekt.
4. Das `control`-Objekt empfängt das Ereignisobjekt und identifiziert als Quelle des Ereignisses das `showProblems`-Objekt. Daraufhin ruft es die fachliche Methode zur „benutzerdefinierten Suche nach einem Problem“ des `mainWindow`-Objekts auf (diese Methode besitzt den Namen „`customSearch`“, benötigt keine Parameter und liefert kein Ergebnis zurück).

Die Benutzereingabe hat auf diesem Weg die Applikationsschicht erreicht. In den folgenden Schritten soll deutlich werden, wie Domain Klassen zur Kommunikation zwischen Applikationsschicht, Zwischenschicht und Server-Schicht dienen.

5. Innerhalb der `customSearch`-Methode des `mainWindow`-Objekts wird vom `control`-Objekt ein Objekt der Klasse `ProblemSearchInfo` angefordert.
6. Das `control`-Objekt leitet die Anforderung an das `searchInfo`-Objekt weiter.
7. Das `searchInfo`-Objekt erstellt aus den Eingaben in den Eingabefeldern ein `ProblemSearchInfo`-Objekt mit den vom Benutzer eingetragenen Daten und gibt dieses zurück.

8. Das `control`-Objekt gibt seinerseits das `ProblemSearchInfo`-Objekt zurück.
9. In der `customSearch`-Methode des `mainWindow`-Objekts wird daraufhin ein neues Applikationsobjekt der Klasse `AppProblemOverview` (mit dem Namen `problemView`) erzeugt. An dessen Konstruktor wird eine Referenz auf das Server-Objekt und das `ProblemSearchInfo`-Objekt weitergegeben. Damit endet die `customSearch`-Methode.

Die Aufgabe, eine Liste von Problemen anzuzeigen, wird vom `mainWindow`-Objekt also an ein anderes, auf diese Aufgabe spezialisiertes Objekt der Applikationsschicht delegiert. Dieses Objekt ist für die Anforderung der Liste vom Server und die grafische Darstellung der Liste zuständig.

10. Beim Instanziieren des `problemView`-Objekts entsteht (wie bei der Instanziierung des `mainWindow`-Objekts) eine dem Objekt untergeordnete Struktur aus Objekten der Zwischenschicht, der Interface-Schicht und der Benutzungsoberfläche.
11. Das `problemView`-Objekt fordert vom Server-Objekt den ersten Listenausschnitt für die Problemsuchmaske an.
12. Das Ergebnis der Serveranfrage wird über die Zwischenschicht und ein Interface schließlich an der Benutzungsoberfläche grafisch dargestellt.

Durch die „Trennung von Funktion und Interaktion“ entsteht also eine einfache aus der Spezifikation abgeleitete Struktur zwischen den Objekten der verschiedenen Schichten. Der Entwurf von PTS/RMI besitzt dadurch einige vorteilhaften Eigenschaften:

- Unter den Schichten existiert eine klare Aufgabenteilung.

- Applikationsschicht und Benutzungsoberfläche können völlig entkoppelt auf eigenen Abstraktionsebenen (Domain Klassen bzw. Java Swing Klassen) operieren.

- Objekte der Zwischenschicht und der Interface-Klassen müssen keine funktionalen Anforderungen erfüllen und sind allein auf die Vermittlung von Daten und Ereignissen zwischen Applikationsschicht, Domain Klassen und Benutzungsoberfläche spezialisiert.

Das nachfolgende Diagramm zeigt zusammenfassend alle Klassen der Applikationsschicht, der Zwischenschicht und der Benutzungsoberfläche sowie deren Beziehungen untereinander. Auf eine detaillierte Erläuterung der Funktion jeder einzelnen Klasse wird verzichtet (die Funktion der Klassen in der Applikationsschicht wird bereits am Anfang dieses Kapitels beschrieben). Stattdessen werden nur einige ausgewählte Teile des Diagramms erläutert.

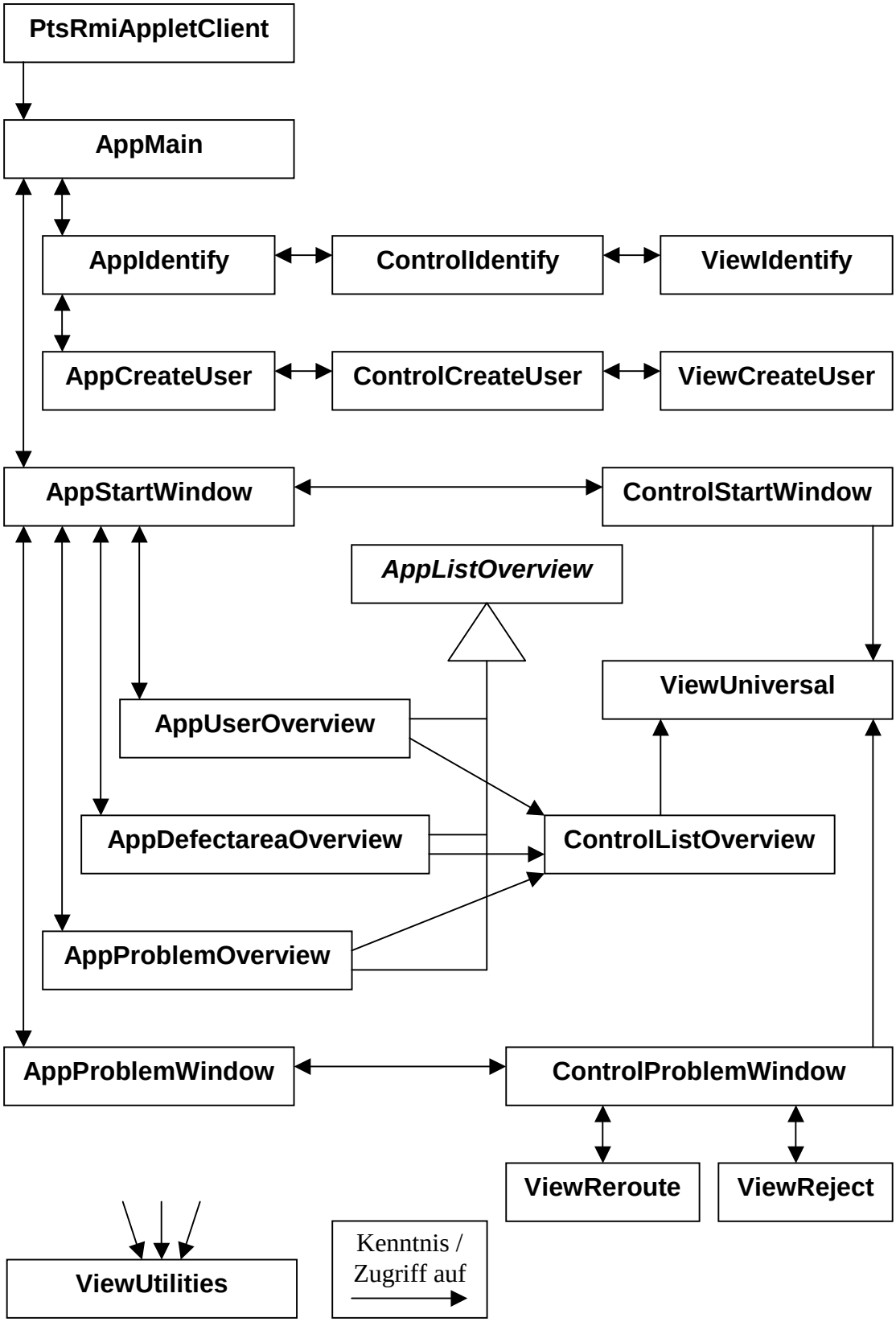


Abbildung 16 alle Klassen der Applikationsschicht, Zwischenschicht und der Benutzungsoberfläche

- Die Klasse **`ViewUniversal`** ist eine Unterklasse von `JFrame` – einem Java Swing Fenster. Mit der eigenen Klasse `MenuDescriptor` werden Schaltflächen auf der linken Seite des Fensters erzeugt, denen allen als Ereignisempfänger ein einziges Objekt der Zwischenschicht zugeordnet wird. Ereignisse die das Fenster selbst betreffen werden an ein `WindowListener`-Objekt weitergeleitet. Da alle Klassen der Zwischenschicht dieses Interface erfüllen, kann diese Klasse mit allen Objekten der Zwischenschicht zusammenarbeiten.
  Die rechte Seite des Fensters kann mit einer beliebigen Java Swing Komponente (`JComponent`) gefüllt werden. Objekte der Klasse `ViewUniversal` werden für die grafische Darstellung des Hauptfensters, der Listenfenster und der Problemfenster genutzt.

- Objekte der Zwischenschicht-Klasse **`ControlListOverview`** sprechen ihre Applikationsobjekte über das gemeinsame Interface `AppListOverview` an (siehe Klassendiagramm rechts). Die in der abstrakten Klasse vorgeschriebenen Methoden reichen für die korrekte Weiterleitung von Benutzerereignissen in Listenfenstern aus.

AppListOverview
+setEnabled(enabled: boolean) +selectItem(key: String) +next() +previous() +backToStartWindow()

- Bei der Klasse **`ControlProblemWindow`** wird deutlich, dass zu einem Applikationsobjekt auch mehrere verschiedene Klassen von Fenstern gehören können. Die Klassen **`ViewReroute`** und **`ViewReject`** sind für die Eingabe von Information zu den Workflow-Aktionen „Reroute“ und „Reject“ zuständig, die in PTS/RMI innerhalb eines eigenen kleinen Fensters erfolgt. Neue Interaktionsformen können so sehr einfach einem Fenster hinzugefügt werden.

- Die Klasse **`ViewUtilities`** stellt den Klassen der Benutzungsoberfläche oder der Zwischenschicht einige nützliche Methoden zur Verfügung. Die Methoden zum Zentrieren eines Fensters auf dem Bildschirm und zum Anzeigen einer kleinen Fehlermeldung, die ein anderes Fenster bis zu Bestätigung durch den Benutzer blockiert, sind die beiden am häufigsten genutzten Methoden dieser Klasse.

4.3 Implementierung und Integration

In diesem Kapitel werden einige Details der Implementierung von PTS/RMI beschrieben, die im weiteren Verlauf der Diplomarbeit von Bedeutung sein werden:

- Die Datenbankanbindung wurde bei allen Varianten identisch gelöst. Die für PTS/RMI entwickelte Implementierung wird auch in PTS/CORBA und PTS/JSP eingesetzt und deswegen hier erläutert.
- Die Umwandlung von Ergebnissen aus Datenbankanfragen in Objekte der Domain Klassen erfolgt in PTS/RMI und PTS/JSP auf eine andere Art und Weise als in PTS/CORBA. Ursache ist die unterschiedliche Modellierung der

Domain Klassen. Für den Vergleich mit PTS/CORBA werden die Auswirkungen der Domain Klassenmodellierung auf den Code gezeigt.

- Die grafische Darstellung von Domain Klassen erfolgt in PTS/RMI und PTS/CORBA mit Hilfe von Interface Klassen. Da sich die Domain Klassen von PTS/RMI und PTS/CORBA fundamental unterscheiden, soll gezeigt werden, welche Auswirkungen die Modellierung der Domain Klassen auf die Generierung der grafischen Darstellung in PTS/RMI hat.

4.3.1 Datenbankanbindung

Die Datenbankanbindung von PTS/RMI an [PTS Datenbank] erfolgt mit mehreren Verbindungen. Transaktionen, die sich über mehrere Datenbankzugriffe erstrecken, werden in JDBC mit einer Verbindung gekoppelt. Damit von der Transaktion unabhängige Funktionalität ohne (auf Ebene des Java Programms) sichtbare Einschränkungen auf die Datenbank zugreifen kann, werden diese Datenbankanfragen über eine andere Verbindung gestellt. Insgesamt werden sieben Verbindungen im Server-Objekt erzeugt. Über jede Verbindung wird eine Teilmenge der Datenbankzugriffe abgewickelt:

- *Bearbeiter* in der Datenbank suchen (zur Anmeldung)
- *Bearbeiter* auflisten
- Problembereiche auflisten
- *Probleme* auflisten
- ein neues *Problem* erstellen oder einen neuen *Bearbeiter* erstellen
- Daten eines *Problems* von der Datenbank anfordern
- Primärschlüssel eines neuen *Problems* ermitteln

Da es Probleme bei der Nutzung von vorkompilierten SQL-Anweisungen („prepared statements“) mit dem JDBC-Treiber von DB2 gab, werden alle SQL-Anweisungen als Zeichenkette an die Datenbank übergeben. Fehlerhafte SQL-Anweisungen werden erst zur Laufzeit erkannt. Um schon beim Kompilationsvorgang ein Mindestmaß an Korrektheit der Zeichenketten zu gewährleisten, aber auch um einen einfachen Austausch von Tabellen- und Spaltennamen zu ermöglichen, werden alle Bezeichner der Datenbank in `String`-Konstanten gehalten.

4.3.2 Umwandlung von Datenbankanfragen in Domain Objekte

Jedes Attribut eines Domain Objekts wird in PTS/RMI durch eine eigene Anweisung in der `PtsRmiServer`-Klasse mit Daten aus der Datenbank gefüllt. Die Details der Datenbankanbindung bleiben dadurch den Domain Klassen verborgen. Im folgenden Code-Ausschnitt aus der Klasse `PtsRmiServer` wird ein *Problem* mit Daten aus der Datenbank versehen:

```
Problem result = new Problem();
  // (...) Datenbankanfrage stellen.
  // Das Ergebnis liegt im “ResultSet“-Objekt “rs“ vor.
result.problemId    = rs.getString(DB_PROBLEM_ID);
result.creationDate = rs.getDate(DB_PROBLEM_CREATION_DATE);
```

Die beiden Konstanten `DB_PROBLEM_ID` und `DB_PROBLEM_CREATION_DATE` enthalten die Spaltennamen für Primärschlüssel und Erstellungsdatum eines *Problems*.

4.3.3 Grafische Darstellung von Domain Objekten

Ähnlich wie bei der Umwandlung von Datenbankanfragen, muss für den Datentransfer zwischen grafischer Darstellung eines Domain Objekts und dem Domain Objekt selbst, jedes Attribut des Domain Objekts einzeln angesprochen werden. Einige Code-Ausschnitte aus der Interface-Klasse `ProblemContentProvider` (zuständig für die Visualisierung von *Problemen*) veranschaulichen dies:

- Zunächst wird die Hilfsmethode `createField` (zur Erzeugung von Eingabefeldern mit Hilfe der NULL-Zeiger Semantik) definiert:

```
private JTextField createField(Object content) {
    if (content != null) {
     return new JTextField(content.toString());
    } else {
     return new JTextField();
    }
}
```

- Im Konstruktor von `ProblemContentProvider` werden die Swing Objekte zur grafischen Darstellung eines *Problems* instanziiert:

```
private final String[] labelText
     = {"Problem Id", "Problem Type*",
      (…für jedes Attribut wird ein Beschriftungstext angegeben)
        };
```

```
public ProblemContentProvider(Problem target) {
    problem = target;
    labels = new JLabel[labelText.length];
    fields = new JTextField[labelText.length];

    // alle Beschriftungen erzeugen
    for (int i = 0; i < labelText.length; i++) {
     labels[i] = new JLabel(labelText[i] + ":");
    }

    // alle Eingabefelder erzeugen
    fields[0]  = createField(problem.problemId);
    fields[1]  = createField(problem.problemType);
   (… für jedes Attribut wird mit einer eigenen Anweisung ein Eingabefeld
      erzeugt)
```

- Das Einlesen der eingegebenen Daten erfolgt mit Hilfsmethoden zur Datenkonvertierung auf ähnliche Art und Weise wie bei der Umwandlung von Datenbankanfragen in Domain Objekte in der Klasse `PtsRmiServer`.

Zunächst einige Hilfsmethoden in der Klasse `ProblemContentProvider`:

```
private String getNonEmpty(String str) {
    if (str == null || str.equals("")) {
     return null;
    } else {
     return str;
    }
}

private Character getFirstChar(String str) {
    char[] chars = str.toCharArray();
    if (chars.length > 0) {
     return new Character(chars[0]);
    } else {
     return null;
    }
}
```

Der Einlesevorgang selbst erfordert wieder das Ansprechen jedes einzelnen Attributs eines *Problems*. Der folgende Code-Ausschnitt aus der Methode `updateTarget` der Klasse `ProblemContentProvider` veranschaulicht dies:

```
problem.problemId   = getNonEmpty(fields[0].getText());
problem.problemType = getFirstChar(fields[1].getText());
```

Für jedes Attribut werden die `String`-Daten eines Eingabefelds mit einer Hilfsmethode in Java Objekte umgewandelt. Welche Hilfsmethode benutzt werden muss, hängt vom konkreten Attribut ab. Da das Attribut `problemId` ein `String` Objekt ist, aber keinen leeren Wert enthalten darf, ist die Hilfsmethode `getNonEmpty` anzuwenden. Der Wert des Attributs `problemType` wird mit Hilfe der Hilfsmethode `getFirstChar` auf das erste Zeichen einer nicht leeren Zeichenkette oder andernfalls auf NULL gesetzt.

4.4 Erweiterung

In diesem Kapitel werden die Veränderungen am Code von PTS/RMI infolge der nachträglichen Erweiterung der Funktionalität beschrieben. Durch die Erweiterung waren (wie bei der Festlegung der Erweiterung geplant) alle Schichten von PTS/RMI betroffen.

Die nachträgliche Erweiterung der Funktionalität von PTS/RMI wurde als eine einfache Aufgabe empfunden. Aus den neu hinzugekommenen Teilen der Spezifikation ließen sich mit den bereits beim Entwurf von PTS/RMI angewendeten Konzepten die notwendigen Änderungen an bestehenden Klassen und neue Klassen herleiten.

4.4.1 Änderungen an der Datenbank

In der Datenbank wurde eine neue Tabelle „**MORE_INFO**“ zur Aufnahme der Problemzusatzinformation erstellt. Die Zuordnung einer Problemzusatzinformation zu einem *Problem* erfolgt durch die Aufnahme des *Problem*-Primärschlüssels in die Tabelle.
Die Tabelle „**MORE_INFO**“ besitzt die folgenden Spalten:

Spaltenname	SQL-Definition	Beschreibung
`InfoID`	`Varchar(254) NOT NULL`	Primärschlüssel dieser Tabelle
`ProblemId`	`Char(8) NOT NULL`	Fremdschlüssel aus der Tabelle „**Problem**“
`Content`	`Varchar(254) NOT NULL`	die Problemzusatzinformation

Die Primärschlüssel der neuen Tabelle werden (genauso wie die Primärschlüssel der „**Problem**“ Tabelle) in der Tabelle „**Id_Table**“ verwaltet. In der Tabelle „**Id_Table**“ wurde zu diesem Zweck die neue Zeile `(“MORE_INFO“, 0)` hinzugefügt.

4.4.2 Änderungen an den Domain Klassen

Im Interface `WorkflowOperationable` kamen die zwei Konstanten `REQ_MORE_INFO` und `SUBMIT_MORE_INFO` für die beiden neuen Workflow-Aktionen „Request info“ und „Submit info“ hinzu.

Zur Klasse `Problem` wurde die Konstante `STATUS_MORE_INFO` für den Workflow-Zustand „More Info“ hinzugefügt. Außerdem wurden die Methoden `canApply` und `apply` um die korrekte Behandlung des neuen Workflow-Zustands und der beiden neuen Workflow-Aktionen erweitert.

Die Klasse `ProblemMoreInfo` (Problemzusatzinformation) wurde neu erstellt. Objekte dieser Klasse nehmen die Daten einer Zeile der Datenbanktabelle „**MORE_INFO**“ auf und sind serialisierbar.

4.4.3 Änderungen an der Server-Schicht

Dem Interface `ServerAccess` wurde die Methode `getMoreInfoSlice` zur Anforderung eines Listenausschnitts von Problemzusatzinformationen hinzugefügt.

Die Klasse `PtsRmiServer` wurde an mehreren Stellen erweitert:

- Die Datenbankanbindung musste um die Tabelle „**MORE_INFO**“ ergänzt werden (neue Konstanten für Datenbankbezeichner und neue SQL-Anweisungen). Es wurde eine neue Verbindung für die Bearbeitung von Datenbankanfragen, die Problemzusatzinformationen betreffen, hinzugefügt.
- Die Methode `getMoreInfoSlice` aus dem Interface `ServerAccess` wurde implementiert.
- Die Methode `commitProblem` wurde um die Behandlung der beiden neuen Workflow-Aktionen erweitert. Objekte der Klasse `ProblemMoreInfo` müssen dort erstellt und in der Datenbank gespeichert werden.

4.4.4 Änderungen am Client

Applikationsschicht:

Zur Klasse `AppProblemWindow` (Bearbeitung eines *Problems* in einem Problemfenster) wurden drei fachliche Methoden hinzugefügt:

- zur Anzeige der Problemzusatzinformation eines Problems
- zum Anfordern zusätzlicher Problemzusatzinformation (Workflow-Aktion „Request Info“)
- zur Übermittlung von neuen Problemzusatzinformationen (Workflow-Aktion „Submit Info“)

Bei der Übermittlung neuer Problemzusatzinformation wird ein neues Fenster zur Eingabe der Problemzusatzinformation erzeugt und angezeigt. Dieses Fenster wird in der Zwischenschicht durch ein Objekt der Klasse `ControlSubmitInfo` repräsentiert.

Die neu erstellte Klasse `AppMoreInfoOverview` verwaltet die Darstellung von Listen, die Problemzusatzinformationen enthalten. Dazu wird die bereits existierende Klasse `ControlListOverview` verwendet.

Zwischenschicht:

Die Klasse `ControlProblemWindow` erstellt im Problemfenster (einem Objekt der Klasse `ViewUniversal`) die drei neuen Schaltflächen „Information anzeigen“, „Information anfordern“ und „Information schicken“. Die Ereignisse der neuen Schaltflächen werden in der Methode `actionPerformed` an die entsprechenden Methoden des Applikationsobjekts weitergeleitet.

Objekte der neuen Klasse `ControlSubmitInfo` geben an das erzeugende `AppProblemWindow` Objekt vom Benutzer eingegebene Problemzusatzinformation zurück. Für die Darstellung der Benutzungsoberfläche werden Objekte der neuen Klasse `ViewSubmitInfo` verwendet.

Benutzungsoberfläche

Die neue Klasse `ViewSubmitInfo` stellt die Swing Benutzungsoberfläche zur Eingabe einer Problemzusatzinformation zur Verfügung. Die von Objekten dieser Klasse erstellten Swing Komponenten leiten Benutzerereignissen an das erzeugende `ControlSubmitInfo` Objekt weiter.

5 Entwicklung von PTS/CORBA

Bei der Entwicklung von PTS/CORBA konnten Erkenntnisse der vorhergehenden Entwicklung von PTS/RMI berücksichtigt werden. Sowohl die Datenbankanbindung als auch Architektur und Entwurf des interaktiven Client-Programms wurden zum größten Teil von PTS/RMI übernommen. In den Unterkapiteln „Entwurf“ und „Implementierung und Integration“ werden deswegen die Unterschiede zwischen PTS/CORBA und PTS/RMI aufgezeigt, anstatt die Funktion der einzelnen Teile von PTS/CORBA leicht verändert erneut zu erklären.

5.1 Client-Server Kommunikation

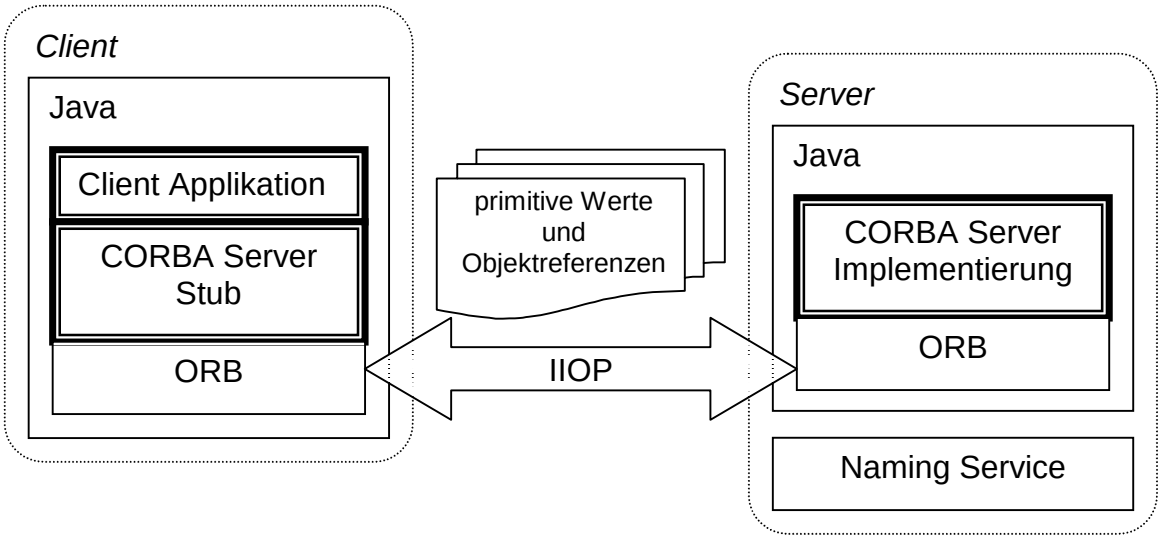


Abbildung 17 Client-Server Kommunikation von PTS/CORBA

Die Client-Server Kommunikationsinfrastruktur von PTS/CORBA unterscheidet sich bei der Client-JVM und bei den zwischen Client und Server übertragenen Daten von der Kommunikationsinfrastruktur bei PTS/RMI:

- Anders als bei PTS/RMI ist für PTS/CORBA kein Web-Browser mit installiertem Java Plug-In auf Seiten des Clients mehr nötig. Stattdessen genügt eine installierte Java Laufzeitumgebung (bei dieser Diplomarbeit [SunJavaRun 1.3]). Die JVM mit der Client Applikation wird direkt vom Benutzer gestartet. Die Client Applikation von PTS/CORBA unterliegt damit keinen Einschränkungen durch eine Sandbox.

- Da CORBA keine NULL-Zeiger Semantik besitzt, können auch keine Null-Zeiger direkt als Parameterwert oder Ergebnis eines Methodenaufrufs zwischen Client und Server ORB übertragen werden.

- Als Rückgabewert und als Parameter können nun nicht wie in PTS/RMI komplette Objekte zwischen Client und Server transferiert werden, sondern lediglich deren Objektreferenz („Inter ORB Reference“ – IOR).

Abgesehen davon sind die Infrastrukturen für die Client-Server Kommunikation von PTS/RMI und von PTS/CORBA einander relativ ähnlich:

- Genauso wie bei PTS/RMI wird bei PTS/CORBA ein separates Programm benötigt, das veröffentlichte Objekte verwaltet, und damit den Zugriff auf das Server-Objekt von anderen Rechnern aus möglich macht. Im Gegensatz zu Java RMI kann der „Naming Service"-Prozess in CORBA auch auf einem völlig anderen Rechner im Netzwerk laufen. Der „Naming Service" wird vom Programm `tnameserv` angeboten, das in der Java Entwicklungsumgebung enthalten ist.

- Auch bei PTS/CORBA werden Stub-Objekte generierter Hilfsklassen für die Kommunikation von einer Client-Applikation zum Server-Objekt benötigt.

- Der Kommunikationsvorgang selbst ist transparent für die Client- und Server-Programme und wird durch die ORB-Objekte und deren Protokoll IIOP realisiert.

5.2 Entwurf

Die Architektur von PTS/CORBA gleicht in großen Teilen der Architektur von PTS/RMI. Die in PTS/RMI entwickelten Strukturen müssen lediglich auf die striktere Trennung von Client- und Server-Klassen durch IDL Schnittstellen angepasst werden.

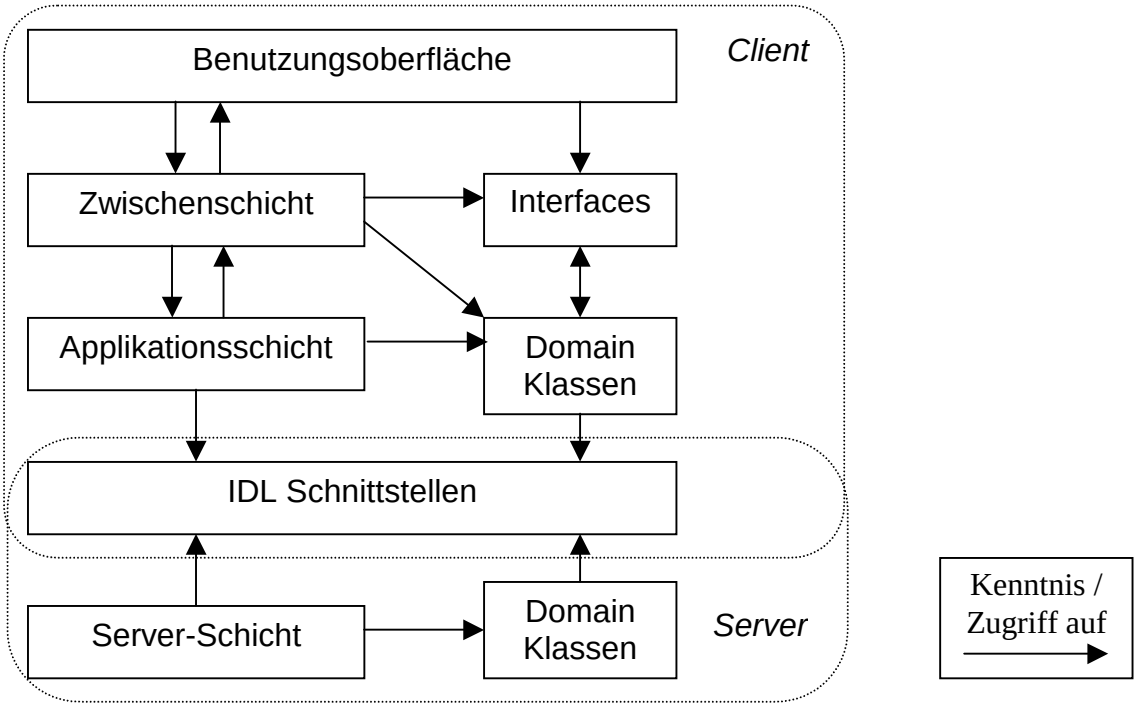


Abbildung 18 Systemarchitektur von PTS/CORBA

Die Trennung von Client und Server durch die IDL Schnittstellen führt lediglich bei den Domain Klassen zu einer Veränderung der PTS/RMI Architektur. Auf entfernte Domain Objekte kann in PTS/CORBA nur über eine IDL Schnittstelle zugegriffen werden. Da

der von SUN in der Java Entwicklungsumgebung [SunJavaSdk 1.3] mitgelieferte ORB nur transiente Objekte (also Objekte, die nicht von einem ORB zu einem beliebigen anderen ORB transferiert werden können) unterstützt, muss der Zustand eines entfernten Domain Objekts auf ein lokales Domain Objekt übertragen werden können. Andernfalls würden viele Zugriffe auf ein entferntes aber häufig benutztes Domain Objekt zu einer erheblichen Netzwerkbelastung führen. Denn der Aufruf einer in der IDL Schnittstelle des Objekts definierten Methode veranlasst den Aufbau mindestens einer Netzwerkverbindung zwischen den ORB-Objekten der beteiligten Rechner.

Der Zugriff auf das Server-Objekt in PTS/CORBA unterscheidet sich nicht wesentlich von der Realisierung in PTS/RMI. In beiden Varianten wird das Server-Objekt über ein Interface angesprochen. Während das Interface in PTS/RMI durch ein Java Interface definiert wird, erfolgt die Definition in PTS/CORBA mit Hilfe der CORBA IDL.

5.2.1 Modellierung der Domain Klassen

Anders als in der Variante PTS/RMI, erfolgte die Modellierung der Domain Klassen in PTS/CORBA nicht durch die direkte Umsetzung von in der Analyse und Spezifikationsphase entdeckten Klassen. Stattdessen wurde eine Struktur genutzt, die eine einfache Übertragung und Kopie der Daten von Domain Objekten ermöglicht. Darüber hinaus soll die NULL-Zeiger Semantik mit dieser Struktur übertragen werden können, die CORBA selbst nicht enthält.

Anstelle von eigenen Klassen, erhalten die identifizierten Klassen „Problem“, „Problembereich“, „Problemsuchmaske“ und „Bearbeiter“ eine übergeordnete Struktur. Mit dieser Struktur kann auch die Information zu den Workflow-Aktionen „Reject“ und „Reroute“, übertragen werden.

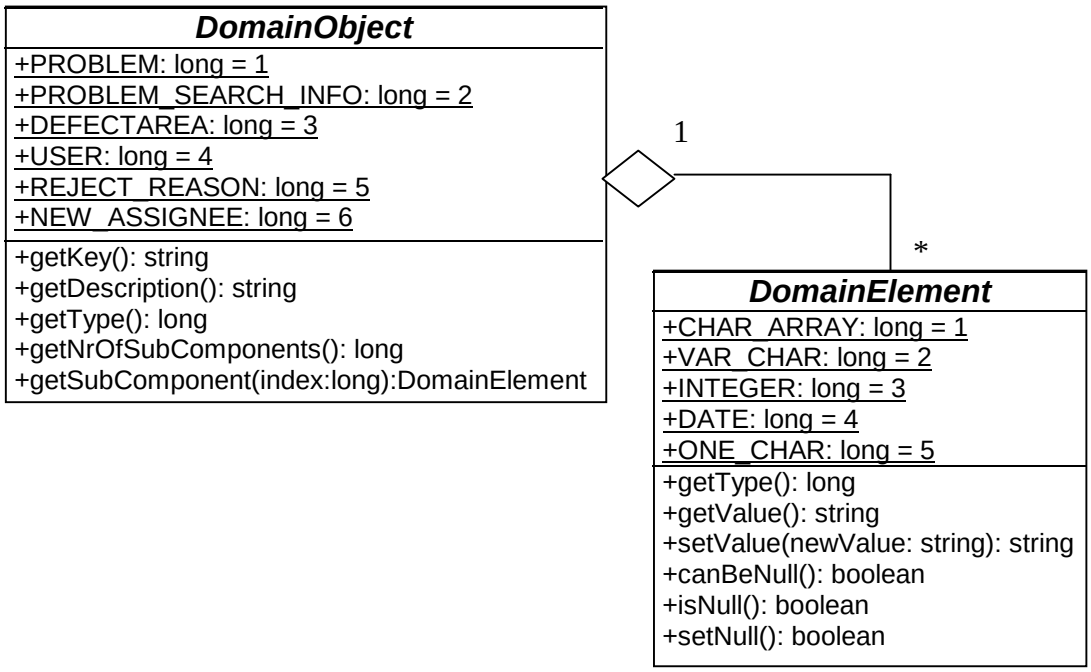


Abbildung 19 UML-Klassendiagramm der abstrakten Klassen `DomainObject` und `DomainElement`

Ein Objekt der Klasse `DomainObject` enthält eine Menge von `DomainElement` Objekten und ermöglicht den Zugriff auf einzelne `DomainElement` Objekte. Jedes `DomainObject` Objekt kennt die fachliche Klasse („Problem“, „Problemsuchmaske“ etc.) des Objekts, das es repräsentiert.

Objekte der Klasse `DomainElement` enthalten den Wert einzelner Attribute einer fachlichen Klasse. Sie sind auch in der Lage, NULL-Zeiger Semantik zwischen Client und Server zu übertragen. Attributwerte können nur als Zeichenketten gesetzt und gelesen werden. Diese Eigenschaft wurde gewählt, um eine einheitliche Konvertierungen beim Einlesen oder Ausgeben von Daten (für Benutzungsoberfläche oder Datenbank) zu gewährleisten, und wird an vielen Stellen des Codes von PTS/CORBA in Anspruch genommen.

Ein Vorteil dieser Modellierung der Domain Klassen in PTS/CORBA gegenüber PTS/RMI ist die Navigierbarkeit durch die einzelnen Attribute eines fachlichen Objekts. Sollen alle Attribute eines fachlichen Objekts ausgegeben oder eingelesen werden, kann dies, sofern kein fachliches Wissen dazu notwendig ist, mit einer Schleife in wenigen Java Anweisungen umgesetzt werden. Diese Modellierung entstand einerseits aus den (im Vergleich zu Java RMI) stärkeren Einschränkungen bei der Definition der Client-Server Schnittstelle und andererseits aus den Erfahrungen bei der Implementierung von PTS/RMI. Der Lerneffekt aus der vorhergehenden Entwicklung von PTS/RMI hat sich an dieser Stelle bereits auf den Entwurf von PTS/CORBA niedergeschlagen.

Die fachlichen Klassen „Listenausschnitt“ und „Listeneintrag“ sind nicht auf diese Struktur abgebildet worden. Da die Klasse „Listenausschnitt“ die Klasse „Listeneintrag“ lediglich zur internen Verwaltung der Daten genutzt hat, muss die Klasse „Listeneintrag“ nicht in die IDL-Schnittstelle aufgenommen werden. Die abstrakte Klasse „Listenausschnitt“ ist wie folgt in der IDL-Schnittstelle definiert:

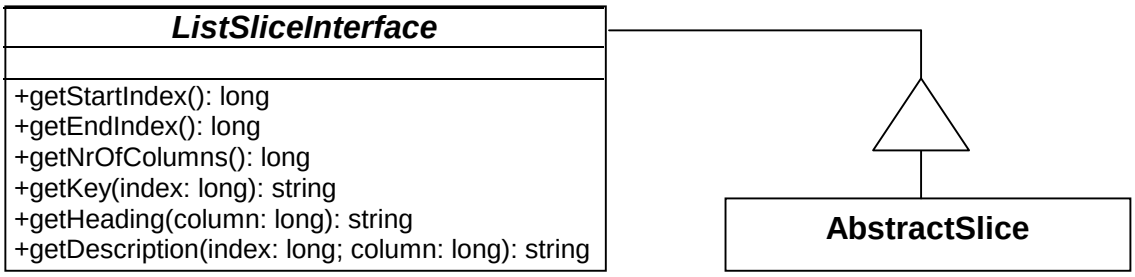


Abbildung 20 UML-Klassendiagramm der abstrakten Klasse `ListSliceInterface`

Bis auf die verwendeten Datentypen und den Namen (es handelt sich um IDL-Datentypen) entspricht die abstrakte Klasse `ListSliceInterface` der Java Klasse `ListSlice` von PTS/RMI. Die Java Implementierung erfolgt in `AbstractSlice`.

Die vorgestellten Domain Klassen von PTS/CORBA wurden nicht als reine Transportbehälter für die Daten anderer Klassen (beispielsweise der Domain Klassen von PTS/RMI) verwendet, was ein anderer denkbarer Einsatzzweck für diese Klassen wäre. Stattdessen sollte im Quellcode von PTS/CORBA beobachtet werden können, welche Auswirkungen eine durch CORBA erzwungene Beschränkung auf IDL Datentypen haben kann.

5.2.2 Modellierung der Server-Schicht

Die Server-Schicht von PTS/CORBA übernimmt auch Aufgaben, die in PTS/RMI von Domain Klassen wahrgenommen wurden. Da nur transiente Objekte vom SUN ORB unterstützt werden, kann das Verhalten von konkreten Objekten nicht zwischen Client und Server übertragen werden.

Anders als in PTS/RMI, werden die Schnittstellen zur Server-Schicht von PTS/CORBA in IDL definiert. Dabei wurden notwendigerweise die PTS/CORBA Domain Klassen anstelle der entsprechenden Domain Klassen in PTS/RMI als Ergebnisklasse oder Parameterklasse einer Methode verwendet. Das IDL-Interface **`CorbaServer`** entspricht dem Java Interface `ServerAccess` in PTS/RMI. In `CorbaServer` sind nun auch die Konstanten der Workflow-Aktionen untergebracht, die sich bei PTS/RMI noch in der Domain Klasse `Problem` befanden.

Die Implementierung der Server-Schicht erfolgt in der Klasse **`CorbaServant`**. Die Klasse `CorbaServant` ist eine Unterklasse von `_CorbaServerImplBase`. Die abstrakte Klasse `_CorbaServerImplBase` wird zusammen mit anderen CORBA Hilfsklassen, darunter auch `CorbaServerHelper`, von `idlj` (einem Programm der Java Entwicklungsumgebung) aus der IDL-Schnittstelle `CorbaServer` generiert.

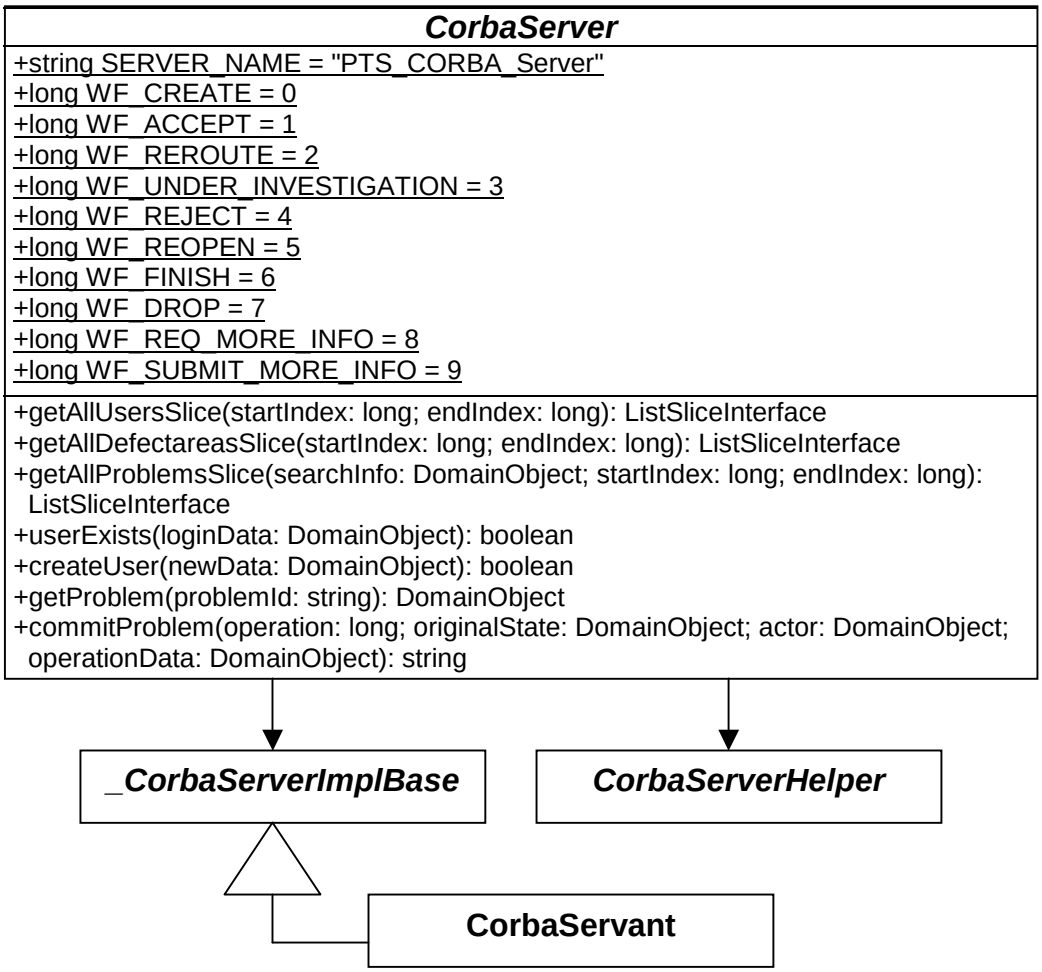


Abbildung 21 Server-Schicht von PTS/CORBA

5.2.3 Modellierung des Clients

Der in PTS/RMI umgesetzte Entwurf hat die Aufgaben eines interaktiven Systems mit grafischer Benutzungsoberfläche in sinnvoller und übersichtlicher Weise zwischen den Schichten und Klassen auf Seiten des Clients verteilt. Diese Aufgabenteilung ist in PTS/CORBA ebenso gut verwendbar, da es dieselbe Funktionalität wie PTS/RMI realisiert.

Der Entwurf für die Klassen des Clients konnte vollständig von PTS/RMI übernommen werden. PTS/CORBA besitzt dieselbe Strukturierung der Funktionalität in der Applikationsschicht und dieselbe Beziehungen zwischen den Klassen der Applikationsschicht, der Zwischenschicht, den Interface-Klassen und der Benutzungsoberfläche.

5.3 Implementierung und Integration

Da es wesentliche Unterschiede zwischen den Domain Klassen von PTS/CORBA und PTS/RMI gibt, sind alle auf diese Schicht zugreifenden Klassen in PTS/CORBA von Änderungen gegenüber den entsprechenden Klassen in PTS/RMI betroffen. Die Klassen der Benutzungsoberfläche greifen nicht auf Domain Klassen zu und konnten deswegen ohne jegliche Änderung des Quellcodes von PTS/RMI nach PTS/CORBA übernommen werden.

Die Unterschiede zwischen PTS/CORBA und PTS/RMI bei einigen Details der Implementierung sollen in diesem Kapitel dargelegt werden.

5.3.1 Implementierung der Domain Klassen

Die Schnittstellen zu den Domain Klassen stellen lediglich Methoden zur Bearbeitung von bereits erstellten fachlichen Objekten zur Verfügung. Die Instanziierung eines `DomainObject` Objekts und die Instanziierung aller darin enthaltenen `DomainElement` Objekte erfolgt im Konstruktor der implementierenden Klasse von `DomainObject` – genannt `AbstractObject`.

Mit dem Wissen um die interne Struktur von fachlichen Objekten dient der Konstruktor der Klasse `AbstractObject` als eine Fabrik („factory“) für `DomainObject` Objekte. Zur Veranschaulichung folgt ein Code-Ausschnitt für die Erzeugung der `DomainElement` Objekte:

```
protected DomainElement[] createComponents(int type) {
    DomainElement[] result = null;
    switch (type) {
    case DomainObject.PROBLEM:
        result    = new DomainElement[23];
        // erstelle Element für Problemprimärschlüssel
        result[0] = new AbstractElement
                    (DomainElement.CHAR_ARRAY, false);
```

(...) entsprechend werden alle anderen Teile eines fachlichen Objekts erzeugt.

Die Klasse `AbstractElement` implementiert die abstrakte Klasse `DomainElement` und enthält im Konstruktor keine großen algorithmischen Teile, sondern initialisiert dort lediglich den Zustand eines neuen Objekts der Klasse. Der Großteil des Quellcodes in dieser Klasse wird für die Prüfungen bei der Aufnahme von neuen Werten in ein `AbstractElement` Objekt verwendet: Ein Code-Ausschnitt für die Prüfung von Integer- und SQL-Datumswerten in der Methode `setValue`:

```
switch (type) {
case DomainElement.INTEGER:
    try {
        Integer.parseInt(newValue);
        value = newValue;
    } catch (Exception e) {
        errorMessage
        = "Error while parsing Integer: " + e.getMessage();
    }
    break;

    case DomainElement.DATE:
    try {
        Date sqlDate = Date.valueOf(newValue);
        if (sqlDate != null) {
            value = newValue;
        } else {
            errorMessage = "Invalid Date format!";
        }
    } catch (Exception e) {
        errorMessage = "Invalid Date format!";
    }
    break;
(...)
}
```

Bei den anderen Datentypen die ein Objekt der Klasse `AbstractElement` aufnehmen kann, sind keine so aufwändigen Prüfungen nötig. Durch die Prüfung beim Setzen jedes Werts wird sichergestellt, dass die als Zeichenkette gespeicherten Werte innerhalb von `AbstractElement` Objekten ohne Probleme in SQL-Anweisungen genutzt werden können

5.3.2 Umwandlung von Datenbankanfragen in Domain Objekte

Auf alle einzelnen Teile (der Klasse `DomainElement`) eines Objekts der Klasse `DomainObject` kann in PTS/CORBA zugegriffen werden mit der Methode `getSubComponent` zugegriffen werden.

Dieser Umstand und die Tatsache, dass einige Typen von `DomainObject` Objekten für jede Spalte der entsprechenden Datenbanktabelle ein `DomainElement` Objekt besitzen, werden beim Einlesen eines *Problems* von der Datenbank genutzt. Im

Vergleich zu PTS/RMI benötigt die Server-Methode `getProblem` von PTS/CORBA für die Übertragung von Daten aus der Datenbank in ein `DomainObject` Objekt wesentlich weniger Java Anweisungen:

```
result = new AbstractObject
            (DomainObject.PROBLEM, problemId);
  // (...) Datenbankanfrage stellen.
  // Das Ergebnis liegt im "ResultSet"-Objekt "rs" vor.
int max = result.getNrOfSubComponents();
for (int i = 0; i < max; i++) {
    errors += result.getSubComponent(i)
                .setValue(rs.getString(i + 1));
}
```

Die fachliche Bedeutung der Einzelnen Spalten der Tabelle ist beim Einlesevorgang nicht relevant. Deswegen genügt die richtige Reihenfolge der `DomainElement` Objekte innerhalb eines `DomainObject` Objekts, um über alle Spalten einer Tabellenzeile iterieren zu können.

Die Variable `errors` speichert eventuelle Fehlermeldungen beim Setzen eines Wertes (die Methode `setValue` liefert eine nicht-leere Zeichenkette zurück).

5.3.3 Grafische Darstellung von Domain Objekten

Ähnlich wie beim Einlesen von Daten aus der Datenbank, ist durch die navigierbare Struktur von `DomainObject` und `DomainElement` Objekten im Vergleich zu PTS/RMI weniger Code für die grafische Darstellung eines `DomainObject` Objekts notwendig. Zum Vergleich mit PTS/RMI (Kapitel 4.3.3) folgt nun ein Code-Ausschnitt aus dem Konstruktor der Interface-Klasse `ProblemContentProvider` (zuständig für die Visualisierung von *Problemen*):

```
private final String[] labelText
    = {"Problem Id", "Problem Type*",
```

(…für jedes Attribut wird ein Beschriftungstext angegeben)

```
      };
public ProblemContentProvider(DomainObject target) {
    problem = target;
    labels = new JLabel[labelText.length];
    fields = new JTextField[labelText.length];
    for (int i = 0; i < labelText.length; i++) {
        labels[i] = new JLabel(labelText[i] + ":");
        if (target.getSubComponent(i).isNull()) {
            fields[i] = new JTextField();
        } else {
            fields[i] = new JtextField
                    (target.getSubComponent(i).getValue());
        }
    }
```

(…)

Da der Test auf einen NULL-Wert nur einmal innerhalb der Schleife durchgeführt wird, lohnt sich eine Auslagerung des Tests in eine Hilfemethode wie bei PTS/RMI nicht mehr.

Das Einlesen der in die Benutzungsoberfläche eingegebenen Daten erfolgt ebenfalls mit Hilfe einer Schleife:

```
String errorMessage = "";
int max = problem.getNrOfSubComponents();
for (int i = 0; i < max; i++) {
    DomainElement el = problem.getSubComponent(i);
    String value = fields[i].getText();
    switch (el.getType()) {
    case DomainElement.CHAR_ARRAY:
    case DomainElement.VAR_CHAR:
    case DomainElement.DATE:
        if (!value.equals("")) {
            errorMessage += el.setValue(value);
        } else {
            el.setNull();
        }
        break;
(… andere Elementtypen behandeln)
    }
}
```

Jede Texteingabe wird je nach Typ des `DomainElement` Objekts in der Switch-Anweisung auf eine Zeichenkette, die einem NULL-Wert entspricht (meist eine leere Zeichenkette) geprüft.

5.3.4 Kopie der Daten von Listenausschnitten

Auf Objekte, die auf Seiten des Servers erzeugt wurden, können die Client Programme von PTS/CORBA nur über eine Referenz zugreifen. Da die Daten der Listenausschnitte in einer Swing Tabelle dargestellt werden, und Swing Tabellen bei jeder Aktualisierung der Benutzungsoberfläche auf ihr Model-Objekt zugreifen, müssen die Daten des Listenausschnitts zuerst vom Server zum Client kopiert werden, bevor sie in angezeigt werden. Andernfalls hätte beispielsweise eine Bewegung des Maus-Cursors über die Tabelle eine Vielzahl von Netzwerkzugriffen von Client zum Server zur Folge.

Der Kopiervorgang erfolgt in einem speziellen Konstruktor der Klasse `AbstractSlice`. Eine Referenz auf ein `ListSliceInterface` Objekt wird als Parameter angegeben. Alle Daten, die über die Schnittstelle der Referenz zugänglich sind, werden in den Attributen des neu instanziierten `AbstractSlice` Objekts abgelegt. Das neue `AbstractSlice` Objekt verhält sich nach außen genauso wie das referenzierte `ListSliceInterface` Objekt. Im Client-Programm von PTS/CORBA wird unmittelbar nach dem Empfang einer Referenz auf ein entferntes `ListSliceInterface` Objekt ein neues, lokales `AbstractSlice` Objekt mit diesem Konstruktor instanziiert.

Die Anforderung eines Listenausschnitts hat bei PTS/CORBA stets mehrere Zugriffe des Clients auf den Server zur Folge. Der Kopiervorgang der Daten erfolgt – anders als in PTS/RMI – explizit durch den Entwickler.

5.4 Erweiterung

Dieses Kapitel erläutert die Veränderungen am Code von PTS/CORBA bei der nachträglichen Erweiterung der Funktionalität. Durch die Erweiterung waren genauso wie bei PTS/RMI alle Schichten der Architektur betroffen. Der Entwurf von PTS/CORBA gleicht zum größten Teil dem Entwurf von PTS/RMI und bedingt dadurch Änderungen an denselben Stellen wie bei der Erweiterung von PTS/RMI.

Die Erweiterung von PTS/CORBA wurde wegen der vorausgegangenen Erweiterung von PTS/RMI als eine sehr einfache Aufgabe empfunden. Im Vergleich zu PTS/RMI wird bei PTS/CORBA mehr mit Daten (beispielsweise die neue Konstante `MORE_INFO`) gearbeitet, da der Zugriff auf Attribute fachlicher Klassen über die `DomainObject` Klasse erfolgt.

5.4.1 Änderungen an der Datenbank

Da bereits bei der Erweiterung von PTS/RMI die Tabelle „**`MORE_INFO`**“ zur Aufnahme der Problemzusatzinformation erstellt wurde, konnte auf diese Tabelle zurückgegriffen werden. Auch der Eintrag in der Tabelle „**`Id_Table`**“ zur Verwaltung von Primärschlüsseln der Tabelle „**`MORE_INFO`**“ war bei der Erweiterung von PTS/RMI erstellt worden.

5.4.2 Änderungen an der IDL Schnittstelle

Die zusätzliche Funktionalität bringt die fachliche Klasse „Problemzusatzinformation“ mit sich. Bei der Modellierung der Domain Klassen in PTS/CORBA bedeutet dies, dass ein neuer Typ von `DomainObject` Objekten zu den bestehenden Typen hinzu kommt. Die IDL Schnittstelle der abstrakten Klasse wird um eine neue Konstante `MORE_INFO` erweitert, mit der ein `DomainObject` Objekt die Repräsentation einer Problemzusatzinformation signalisieren kann.

Die neuen Workflow-Aktionen erfordern neue Aktionskonstanten in der IDL Schnittstelle der abstrakten Klasse `CorbaServer`. Die beiden neuen Konstanten `WF_REQ_MORE_INFO` und `WF_SUBMIT_MORE_INFO` entsprechen den Workflow-Aktionen „Request info“ und „Submit info“. Sie können als `operation`-Parameter der Methode `commitProblem` die Durchführung der beiden neuen Workflow-Aktionen bewirken.

Die IDL Schnittstelle von `CorbaServer` stellt außerdem die neue Methode `getMoreInfoSlice` zur Verfügung, mit der Listenausschnitte einer Liste von Problemzusatzinformationen, also im Falle von PTS/CORBA Objekte der Klasse `ListSliceInterface`, vom Server angefordert werden können.

5.4.3 Änderungen an den Domain Klassen

Die Implementierung der abstrakten Klasse `DomainObject`, die Klasse `AbstractObject`, behandelt in ihrem Konstruktor die neue Konstante `MORE_INFO` und erzeugt dafür die notwendigen `DomainElement` Objekte für eine Problemzusatzinformation.

5.4.4 Änderungen an der Server-Schicht

In der Klasse `CorbaServant` waren dieselben Erweiterungen wie bei PTS/RMI in der Klasse `PtsRmiServer` notwendig:

- Auch hier musste die Datenbankanbindung um die Anbindung der neuen Tabelle „**`MORE_INFO`**“ ergänzt werden (Konstanten für Datenbankbezeichner und SQL-Anweisungen, eine neue Verbindung für Zugriffe auf die Tabelle).
- Die neue Methode `getMoreInfoSlice` aus der IDL Schnittstelle von `CorbaServer` wurde implementiert.
- Die Methode `commitProblem` behandelt nun auch die beiden neuen Workflow-Aktionen. Die Verwaltung von Objekten der fachlichen Klasse „Problemzusatzinformation“ kam zur Funktionalität der Methode hinzu.

5.4.5 Änderungen am Client

Bei der Erweiterung der Funktionalität des Client-Programms von PTS/CORBA wurden dieselben Klassen erstellt und geändert wie in PTS/RMI. Bis auf die Behandlung der Domain Objekte, erfüllten Änderungen an bestehenden Klassen oder die Erstellung von neuen Klassen bei PTS/CORBA und PTS/RMI dieselben Ziele. Die Einzelheiten der Erweiterung werden nicht nochmals dargelegt (siehe dazu Kapitel 4.4.4).

Die folgenden Klassen des Client-Programms von PTS/CORBA waren von der nachträglichen Erweiterung betroffen:

- Zur Klasse `AppProblemWindow` wurden drei fachliche Methoden zur Behandlung von Problemzusatzinformation hinzugefügt.
- Die neue Klasse `AppMoreInfoOverview` verwaltet die in Darstellung von Listen mit Problemzusatzinformationen.
- Die Klasse `ControlProblemWindow` bewirkt die Darstellung der drei neuen Schaltflächen im Problemfenster.
- Objekte der neuen Klasse `ControlSubmitInfo` bewirken die Darstellung eines Fensters zur Eingabe von Problemzusatzinformationen und geben diese an das übergeordnete `AppProblemWindow` Objekt zurück.
- Die neue Klasse `ViewSubmitInfo` stellt der Klasse `ControlSubmitInfo` die Swing Benutzungsoberfläche zur Eingabe einer Problemzusatzinformation zur Verfügung.

6 Entwicklung von PTS/JSP

In diesem Kapitel, das die Entwicklung von PTS/JSP erläutert, wird vor allem auf die dynamischen Vorgänge während des Betriebs von PTS/JSP eingegangen. Anders als bei PTS/RMI oder PTS/CORBA besitzt das Programm auf Seiten des Clients kein fachliches Wissen über die Applikation, also über die Funktionalität des „Problem Tracking“ Systems.

6.1 Client-Server Kommunikation

Die Kommunikation zwischen Client und Server erfolgt in PTS/JSP ausschließlich über das „Hypertext Transfer Protocol“ (HTTP).

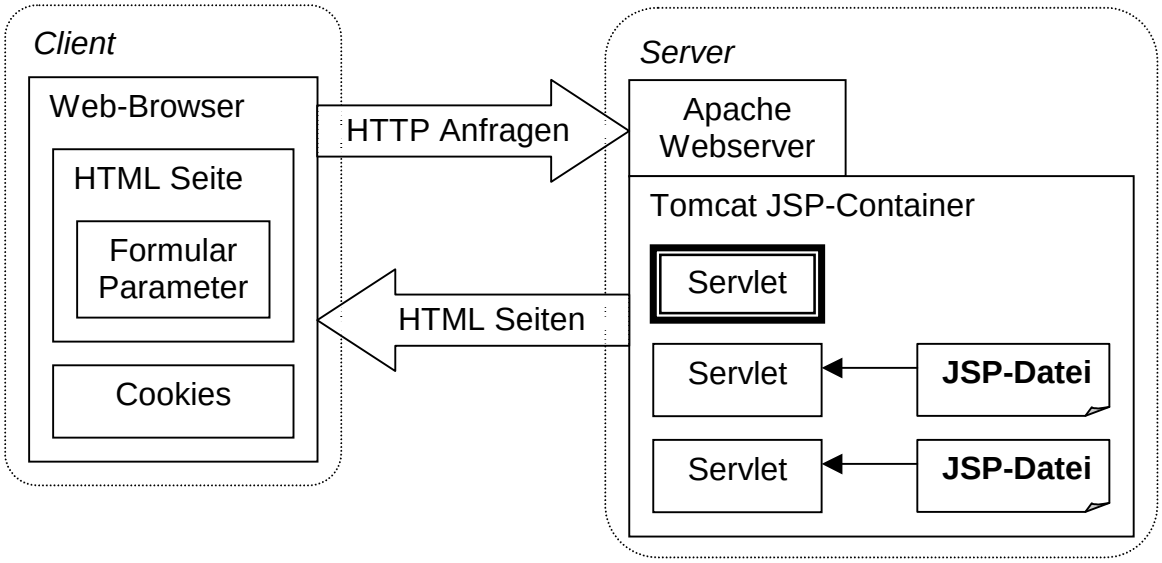


Abbildung 22 Client-Server Kommunikation bei PTS/JSP

HTTP-Anfragen eines Web-Browsers werden vom Webserver [Apache 1.3.14] selbst bearbeitet oder, wenn sie JSP-Seiten betreffen, an den JSP-Container [Tomcat 3.1.1] weitergegeben. In den HTTP-Anfragen werden Daten aus Benutzereingaben vom Client zum Server geschickt. Die Antwort des Servers wird in Form einer HTML-Seite zum Client geschickt. Die Beschreibung der PTS/JSP Benutzungsoberfläche in JSP-Seiten erfolgt also mit HTML-Tags. Der JSP-Container erzeugt aus HTTP-Anfragen Java Objekte, mit denen innerhalb eines Servlets oder einer JSP-Seite die HTTP-Anfrage bearbeitet werden kann.

Der Web-Browser kann über „Cookies“ Zustandsinformation auf Seiten des Clients behalten. Davon wird bei PTS/JSP implizit Gebrauch gemacht, denn zur Identifikation einer Sitzung („session“) muss in den HTTP-Anfragen ein Unterscheidungskriterium existieren. Jede Anfrage für eine JSP-Seite wird vom JSP-Container einer Sitzung zugeordnet. Zur eindeutigen Identifikation einer Sitzung wird durch den JSP-Container bei Beginn einer Sitzung eine noch nicht verwendete Nummer ermittelt und an die Sitzung vergeben. Der Web-Browser des Clients erhält diese Nummer in einem „Cookie“ mit dem Namen `JSESSIONID`. Dieses „Cookie“ wird bei jeder weiteren Anfrage des Web-Browsers an den Server innerhalb der HTTP-Anfragen verschickt.

Anstatt „Cookies“ zu verwenden, gibt es bei JSP auch die Möglichkeit, die Nummer einer Session in den Inhalt einer HTML-Seite zu integrieren. Verweise („Links“) auf andere JSP-Seiten werden innerhalb der Servlets so ergänzt, dass die Sitzungsnummer als ein HTTP-Parameter der nächsten HTTP-Anfrage zum Server geschickt wird („URL Rewriting“). Dies hat zum Beispiel den Vorteil, dass ein einzelner Benutzer gleichzeitig mehrere Sitzungen mit der Applikation abhalten kann. „Cookies“ werden von Web-Browsern für jeden Benutzer (im Sinne des Betriebssystems) oft nur einmal verwaltet.

Während „JSP-Datei“ eine physische Textdatei bezeichnet, steht der Ausdruck „JSP-Seite“ im nachfolgenden Text für die Semantik (Verwendung von JavaBeans, Erzeugung einer HTML-Seite) einer JSP-Datei im Sinne der JSP-Technologie.

6.2 Entwurf

Die Systemarchitektur von PTS/JSP besteht nur aus serverseitigen Teilen. Das Konzept, ein Servlet für die Koordination der JSP-Seiten zu verwenden, wurde aus einem Prototypen der IBM ([IBM_JSP_Prototyp]) übernommen, der ebenfalls JSP-Seiten zur Darstellung der Benutzungsoberfläche nutzt.

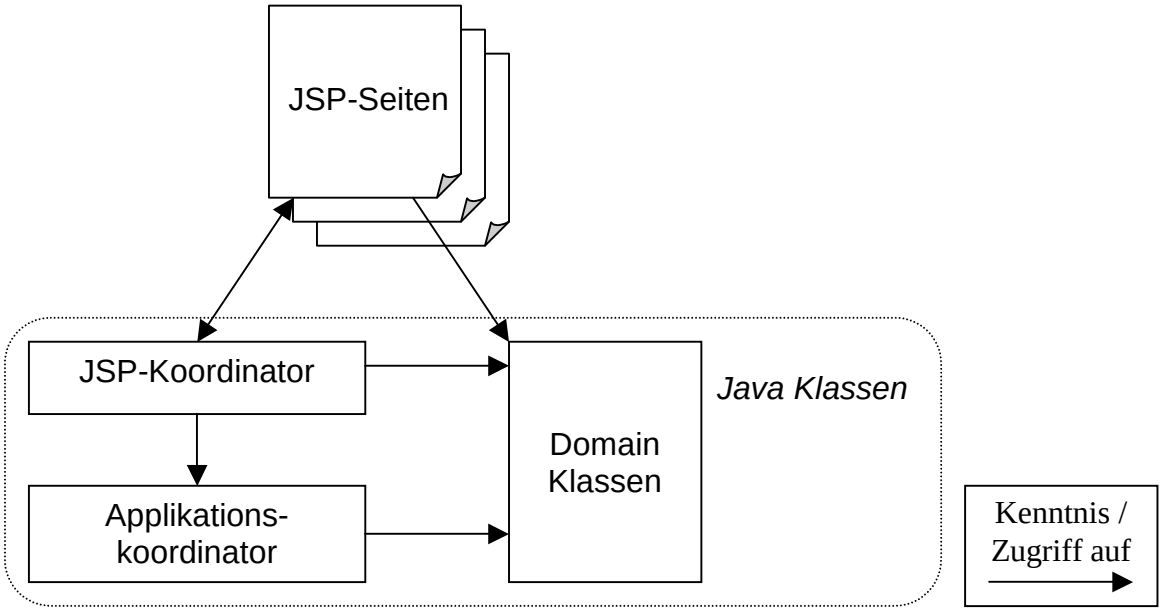


Abbildung 23 Systemarchitektur von PTS/JSP

- Die **JSP-Seiten** beschreiben mit HTML-Tags die im Web-Browser des Clients dargestellte Benutzungsoberfläche. Die darzustellenden Daten erhalten JSP-Seiten aus Domain Objekten, die als JavaBeans in die JSP-Seiten eingebunden werden. Daten, die von Client zu Server übertragen werden sollen, sind in HTML-Formularparametern untergebracht. Jede JSP-Seite von PTS/JSP kennzeichnet HTTP-Anfragen an den JSP-Koordinator durch den HTML-Formularparameter `fromJsp`, der als Wert den Namen ihrer JSP-Datei besitzt. Alle anderen Daten und Benutzereingaben werden ebenfalls in HTML-Formularparametern von Client zu Server geschickt.

- Der **JSP-Koordinator** ist ein Servlet der Klasse `PtsJspController`, das HTTP-Anfragen von JSP-Seiten entgegennimmt und interpretiert. Aus dem Wert des Parameters `fromJsp` und anderen seitenspezifischen Parametern der HTTP-Anfrage ermittelt der JSP-Koordinator, welche Funktionalität auf welchen Domain Objekten durchgeführt werden soll. Die Durchführung findet in vier Schritten statt:
  1. Falls erforderlich, instanziiert der JSP-Koordinator Domain Objekte aus den in HTML-Formularparametern angegebenen Daten einer HTTP-Anfrage.
  2. Die Funktionalität wird mit Hilfe des Applikationskoordinators (zum Zugriff auf die Datenbank) und gegebenenfalls auf den im ersten Schritt ermittelten Domain Objekten ausgeführt.
  3. Die Domain Objekte, die in der resultierenden JSP-Seite als JavaBeans eingebunden werden, werden vom JSP-Koordinator dem aktuellen HTTP-Anfrageobjekt hinzugefügt.
  4. Das HTTP-Anfrageobjekt wird an eine JSP-Seite weitergeleitet. Die Daten aus den in Schritt 3 hinzugefügten JavaBeans werden in der JSP-Seite dargestellt.

  Aufgaben, die der JSP-Koordinators in PTS/JSP wahrnimmt, entsprechen im Vergleich zu PTS/RMI oder PTS/CORBA:
  - der Applikationsschicht: Es wird entschieden, welcher Teil der Funktionalität auszuführen ist. Die Ausführung erfolgt mit Hilfe von Domain Objekten.
  - Teilen der Zwischenschicht: Benutzereingaben werden interpretiert.
  - einem Teil der Aufgaben von Interfaces: Benutzereingaben in Form von Zeichenketten werden in Domain Objekte konvertiert.

- Die **Domain Klassen** von PTS/RMI konnten unverändert in PTS/JSP übernommen werden. Eine neue Klasse `NotifyData` (siehe UML-Klassendiagramm rechts) wurde den Domain Klassen von PTS/JSP hinzugefügt, um in JSP-Seiten (`notification.jsp`) Meldungen zur Information des Benutzers darstellen zu können. Neben einer Überschrift und einer frei definierbaren Textmeldung enthält ein Objekt der Klasse `NotifyData` auch einen Verweis auf die nächste darzustellende Seite der PTS/JSP Applikation. Diese Eigenschaft wird vorwiegend dazu genutzt nach einer Erfolgs- oder Fehlermeldung dem Benutzer die einfache Rückkehr zum Hauptfenster zu ermöglichen.

NotifyData
+title: String +message: String +nextPageMessage: String +nextPageUrl: String

  Eine weitere neue Domain Klasse ist die Klasse `ListSpec`. Mit ihr kann beschrieben werden, welche Domain Objekte eine Liste enthalten soll (siehe UML-Klassendiagramm unten). Objekte der Klasse `ListSpec` werden benötigt, da auf Seiten des Clients keine Zustandsinformationen (außer über die Sitzung) gespeichert werden. Dies bedeutet, dass Listenfenster bei der Anforderung eines anderen Listenabschnitts ihren derzeitigen Zustand zum Server übertragen müssen, damit ein neuer Zustand ermittelt werden kann. Im HTML-Code des darauffolgenden

ListSpec
+type: String +moreInfoProblemId: String +filter: ProblemSearchInfo +getValuesAsParameters(): String

Listenfensters wird der Zustand für die Anforderung eines weiteren Listenabschnitts wieder integriert. Die Daten eines `ListSpec` Objekts werden innerhalb der JSP-Seite für die Darstellung von Listenfenstern durch den Aufruf der Methode `getValuesAsParameters` in HTML-Formularparameter umgewandelt.

- Der **Applikationskoordinator** ist ein Objekt der Klasse `ApplicationControl`, das den Server Objekten von PTS/RMI oder PTS/CORBA entspricht und ebenfalls die Datenbankzugriffe vom Rest des Systems kapselt. Der Code für die Klasse `ApplicationControl` wurde bis auf die Java RMI Besonderheiten von der PTS/RMI Klasse `PtsRmiServer` (in der Version nach den funktionalen Tests) übernommen.

6.2.1 Beziehungen zwischen den JSP-Seiten von PTS/JSP

Es gibt in PTS/JSP insgesamt 6 verschieden JSP-Seiten:

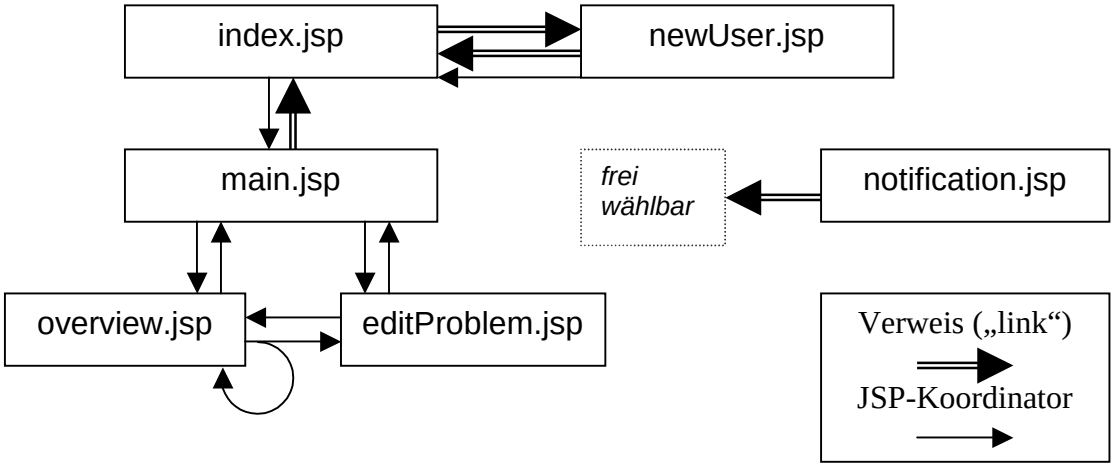


Abbildung 24 Beziehungen zwischen den JSP-Seiten von PTS/JSP

- Die PTS/JSP Applikation wird stets über die JSP-Seite **`index.jsp`** betreten. `index.jsp` zeigt das Anmeldefenster an und verweist direkt auf die JSP-Seite `newUser.jsp`. Meldet der Benutzer sich am System an, wird der JSP-Koordinator davon benachrichtigt. Dieser prüft die eingegebenen Daten des Bearbeiters und zeigt danach die JSP-Seite `main.jsp` oder eine Fehlermeldung an.

- **`newUser.jsp`** stellt ein Bearbeiterregistrierungsfenster dar. Für den Abbruch der Registrierung wird ein Verweis („link“) zurück zu `index.jsp` angeboten. Soll die Registrierung durchgeführt werden, wird der JSP-Koordinator davon benachrichtigt, der dann eine Erfolgs- oder Fehlermeldung anzeigt.

- Die JSP-Seite **`main.jsp`** stellt das Hauptfenster von PTS/JSP dar. Eingabefelder für die Daten der Problemsuchmaske werden durch Elemente eines HTML-Formulars vom Typ „`TEXT`“ zur Verfügung gestellt, die Schaltflächen sind Elemente vom Typ „`RADIO`“. Die Auswahl einer

Schaltfläche setzt den Wert des Parameters `task` auf bestimmte eindeutige Zeichenketten. Je nach Wert des `task` Parameters, entscheidet der JSP-Koordinator, welche Funktionalität des Hauptfensters auszuführen ist, und ob ein Problemfenster (zur Erstellung eines neuen *Problems*) oder ein Listenfenster als nächste JSP-Seite angezeigt wird.

- **`overview.jsp`** stellt mit Hilfe eines JavaBeans der Klasse `ListSlice` alle Arten von Listenfenstern dar und benutzt zur Repräsentation des Zustands (welche Domain Objekte angezeigt werden sollen) ein JavaBean der Klasse `ListSpec`. Auch hier wird dem JSP-Koordinator mit dem Parameter `task` signalisiert, welche Schaltfläche selektiert wurde.

- **`editProblem.jsp`** ist für die Darstellung eines *Problems* im Problemfenster zuständig. Die Daten des Problems werden durch ein JavaBean der Domain Klasse `Problem` in die JSP-Seite integriert. In Problemfenstern von PTS/JSP wird zusätzlich zu den Eingabefeldern für die Daten eines *Problems* ein Eingabefeld für die notwendigen Informationen der Workflow-Aktionen „Reroute“ oder „Reject“ bereitgestellt.

- **`notification.jsp`** ist eine simple JSP-Seite zur Anzeige von Meldungen. Aus einem JavaBean der Klasse `NotifyData` entnimmt die JSP-Seite die darzustellende Meldung, eine Überschrift dazu, eine Adresse zum Fortsetzen der Applikation und eine Beschreibung der Adresse für den Benutzer.

6.3 Implementierung und Integration

Da PTS/JSP unterscheidet sich in seiner Client-Server Kommunikation und in der Systemarchitektur grundlegend von den anderen Varianten. Andererseits konnte aber ein großer Teil des PTS/JSP Codes von PTS/RMI übernommen werden. Daher gibt es nur wenige Stellen, an denen die Implementierung von PTS/JSP mit dem Code der anderen Varianten verglichen werden kann.

Einen Eindruck über die Unterschiede zwischen PTS/JSP und den beiden anderen Varianten kann der Vergleich der grafischen Darstellung von Domain Objekten (am Beispiel eines *Problems*) veranschaulichen.

- Die Ausgabe wird in JSP-Dateien mit Hilfe von HTML-Tags implementiert. Zuerst wird ein *Problem* als JavaBean eingelesen. Anschließend werden die Daten des Problems mit einer Tabelle formatiert in einem HTML-Formulareingabefeld angezeigt:

  ```
  <%@ page import="business.Problem" %>
  <jsp:useBean id="PtsProblem" scope="request"
   class="business.Problem"/>
  ```

  (… damit kann auf ein Problem Objekt zugegriffen werden, das vom JSP-Koordinator zum HTTP-Anfrageobjekt hinzugefügt wurde)

```
<TD> Problem ID: </TD>
<TD><INPUT TYPE="Text" NAME="problemId" SIZE="40"
VALUE="<%= getString(PtsProblem.problemId) %>"></TD>
```

Der Wert des Attributs `problemId` eines `Problem` Objekts wird mit Hilfe der Hilfsmethode `getString` als Zeichenkette in den HTML-Text eingefügt. Die einfache Beschreibung des Layouts durch HTML ersetzt in PTS/JSP die Gruppierung und Konfiguration von Java Swing Objekten zur Zusammenarbeit mit einem „Layout Manager“ wie in der Benutzungsoberfläche von PTS/RMI und PTS/CORBA.

- Die von einem Benutzer eingegebenen Daten, erreichen den JSP-Koordinator in Form von HTML-Formularparametern. Deren Wert kann als `String` Objekt aus dem HTTP-Anfrageobjekt ausgelesen werden. Das Setzen der Werte als Attribute eines Problems geschieht auf dieselbe Art und Weise wie in PTS/RMI.

6.4 Erweiterung

6.4.1 Änderungen an den Domain Klassen

Von der Erweiterung von PTS/JSP waren dieselben Domain Klassen betroffen wie in PTS/RMI, also im einzelnen:

- Das Interface `WorkflowOperationable` wegen der neuen Workflow-Aktionen.
- Die Klasse `Problem` wegen des neuen Workflow-Zustands „More Info“ und damit die Methoden `canApply` und `apply`.
- Die Klasse `ProblemMoreInfo` (Problemzusatzinformation) wurde neu erstellt.

Außerdem musste die PTS/JSP-eigene Domain Klasse `ListSpec` um ein Attribut erweitert werden, damit auch die Anforderung von Problemzusatzinformation zu einem bestimmten Problem repräsentiert werden kann.

6.4.2 Änderungen am Applikationskoordinator

Auch hier waren Änderungen an denselben Stellen notwendig, wie bei der Erweiterung von PTS/RMI an dessen Server-Schicht:

- Die Methode `getMoreInfoSlice` zur Anforderung eines Listenausschnitts von Problemzusatzinformationen wurde hinzugefügt.
- Die Datenbankanbindung musste für den Zugriff auf die Tabelle „**`MORE_INFO`**“ erweitert werden (neue Konstanten für Datenbankbezeichner und neue SQL-Anweisungen, eine neue Verbindung zur Datenbank).
- Die Methode `commitProblem` verarbeitet nun auch die beiden neuen Workflow-Aktionen. Objekte der Klasse `ProblemMoreInfo` können nun dort erstellt und in der Datenbank gespeichert werden.

6.4.3 Änderungen an JSP-Seiten

In `editProblem.jsp` mussten drei neue Schaltflächen (mit eigenen `task` Konstanten) in die HTML-Tabelle eingefügt werden. Dies erforderte zwar eine größere Umstellung der HTML-Tags, da Tabellen in HTML zeilenweise definiert werden, aber bei einer JSP-Seite dieser Größe war dies noch keine Schwierigkeit.

6.4.4 Änderungen am JSP-Koordinator

Die Behandlung der neuen Schaltflächen von `editProblem.jsp` und die Anforderung von Listenausschnitten, die Problemzusatzinformation anzeigen, wurde zur Methode der Klasse `PtsJspController` hinzugefügt, die HTTP-Anfragen bearbeitet.

Für die Anforderung der Listenausschnitte von Problemzusatzinformationen war der größere Anpassungsaufwand notwendig. Die Kriterien, die vormals nur eine Suche nach *Bearbeitern*, Problembereichen oder *Problemen* beschrieben hatten, wurden mit einer `if`-Abfrage untersucht. Die neue Möglichkeit, auch Problemzusatzinformation anzufordern zu können, hatte eine Umstellung zu `switch-case`-Anweisungen zur Folge. Die Tatsache, dass mit dem Entwurf von PTS/JSP die Entscheidung, welche Applikationskoordinatormethode (möglich wären `getAllUsersSlice`, `getAllDefectareaSlice`, `getAllProblemsSlice` oder `getMoreInfoSlice`) aufgerufen werden muss, nur durch den JSP-Koordinator getroffen werden kann, zeigt dessen zentrale Bedeutung in PTS/JSP.

Eine Aufteilung der Ablaufkontrolle in mehrere Servlets wäre möglich, um diesen Umstand abzumildern. Dies erfordert im Gegenzug eine stärkere Parametrisierung der JSP-Seiten (ähnlich wie bei `notification.jsp`). Inwieweit der Vorteil durch die Einsparung von Code bei der Benutzungsoberfläche bei einem solchen Entwurf geschmälert wird, wäre eine interessante Fragestellung im Zusammenhang mit der Software-Entwicklung von unterschiedlichen JSP-Varianten.

7 Durchführung funktionaler Tests

Funktionale Tests wurden bei allen Varianten unmittelbar nach der Integration und nach der Erweiterung durchgeführt. Die einzelnen Testfälle können dem in Kapitel 3.3.4 beschriebenen Testplan entnommen werden.

7.1 Funktionaler Test nach der Integration

Durch die funktionalen Tests nach der Integration konnten bei allen Varianten nur wenige Fehler entdeckt werden. Die in PTS/RMI entdeckten Fehler wurden korrigiert und sind deshalb bei PTS/CORBA und PTS/JSP nicht nochmals aufgetreten.

In **PTS/RMI** wurden die folgenden Fehler entdeckt und korrigiert:

- Sichtbarkeit des Hauptfensters: Nach der Anmeldung (das Anmeldefenster verschwand ohne die Ausgabe einer Fehlermeldung) wurde kein Hauptfenster angezeigt.
  Die Stelle im Code, die den Fehler verursachte, war einfach zu finden. Übergänge zwischen Teilmengen der spezifizierten Funktionalität erfolgen nur innerhalb der Applikationsschicht. Die vollständige Suchreihenfolge für diesen Fehler wäre nach den Konzepten des Entwurfs (siehe dazu auch Abbildung 16):
  1. Prüfung der korrekten Weiterleitung der Benutzereingabe von der Benutzungsoberfläche (Swing Objekte der Klasse `ViewIdentify`) über die Zwischenschicht (`ControlIdentify`) an die Applikationsschicht (`AppIdentify`).
  2. Prüfung des korrekten Übergangs zum nächsten Applikationsobjekt, in diesem Fall von der Klasse `AppMain` zur Klasse `AppStartWindow`.
  3. Prüfung der korrekten Darstellung der grafischen Benutzungsoberfläche eines Applikationsobjekts. Dazu müssen hier die Konstruktoren der Klassen `AppStartWindow`, `ControlStartWindow` und `ViewUniversal` geprüft werden.

  Schon im ersten Schritt wurde entdeckt, dass in der Klasse `AppMain` das Objekt der Klasse `AppStartWindow` nicht aktiviert wurde.

- Eine falsche SQL-Syntax für SQL-Integer Werte führte zu Ausnahmen („exceptions“) beim Zugriff auf die Datenbank. In einer SQL-Anweisung werden Integer-Werte im Gegensatz zu String-Werten ohne umschließende Hochkommata angegeben.
  Der fehlerverursachende Code konnte auf die Klasse `PtsRmiServer` in der Server-Schicht eingegrenzt werden. An allen Stellen, an denen die Werte von Integer Objekten in eine SQL-Anweisung geschrieben wurden, musste nun mit einer neuen Hilfsmethode – speziell für die Umwandlung von Integer Objekten – gearbeitet werden. Da bereits vorher eine Hilfsmethode verwendet wurde (die Hochkommata um den Wert eines Objekts setzt), musste zur Behebung des Fehlers der Hilfsmethodenaufruf bei allen Integer Instanzvariablen geändert werden.

In PTS/CORBA konnte der Fehler unter Nutzung der Typinformation eines `DomainElement` Objekts (durch die Methode `getType`) mit einem „switch-case“ Konstrukt in einer einzigen, auf alle `DomainElement` Objekte anwendbaren Hilfsmethode gelöst werden.

- Tippfehler bei einem Spaltennamen einer Datenbanktabelle: Beim Zugriff auf die Tabelle „**`Problem`**“ trat eine Ausnahme auf.
  Durch die Änderung der betroffenen Zeichenkettenkonstante in der Klasse `PtsRmiServer` war der Fehler schnell behoben. Die Bewahrung von SQL-Bezeichnern in Java als Konstanten hat sich hier vorteilhaft auf den Aufwand zur Fehlerkorrektur ausgewirkt.

- Tippfehler einer Konstante für den Zustand eines *Problems*: In einem Zustand konnten einige nach dem spezifizierten Workflow unzulässige Aktionen durchgeführt werden.
  Auch hier konnte die Fehlerursache wegen der Verwendung von Konstanten schnell (in einer Statuskonstante der Klasse `Problem`) lokalisiert und korrigiert werden.

In **PTS/CORBA** wurden durch die funktionalen Tests zwei Arten von Fehlern entdeckt:

- Fehler beim Zugriff auf eine Tabellenspalte eines `ResultSet`-Objekts über deren Index führte zu Ausnahmen bei lesenden Datenbankzugriffen. Die Tabellenindizes beginnen mit 1 und nicht wie irrtümlich angenommen mit 0.
  In den drei Methoden `getAllUsersSlice`, `getAllDefectareaSlice` und `getProblem` der Klasse `CorbaServant` musste der Zugriff auf die Tabellenspalten dementsprechend angepasst werden. Die Methode `getAllProblemsSlice` nutzt die Methode `getProblem` und konnte daher unverändert gelassen werden.

- Übertragung von NULL-Zeigern: Obwohl die in CORBA fehlende NULL-Zeiger Semantik in den Domain Klassen realisiert ist, um Probleme damit auf diese Klassen zu beschränken, wurden auch an vielen anderen Stellen des PTS/CORBA Codes Ausnahmen durch die versuchte Übertragung eines NULL-Zeigern ausgelöst.
  Diese Fehlerquelle hat den größten Teil der Integrationsphase und der Testphase von PTS/CORBA allein für sich beansprucht. Meist konnte die Ursache bei Parametern gefunden werden, die ohne entsprechende Prüfungen an entfernte Methodenaufrufe weitergegeben wurden. Die Suche nach den Stellen im Code von PTS/CORBA, an denen Prüfungen auf NULL-Zeiger durchgeführt werden müssen, um Ausnahmen zu verhindern war sehr aufwändig. Schließlich musste bereits an der Stelle, an der ein NULL-Zeiger gesetzt wird, auf diesen Fall reagiert werden (meist mit einer Fehlermeldung).

Bei **PTS/JSP** förderten die funktionalen Tests keine Fehler mehr zutage.

7.2 Funktionaler Test nach der Erweiterung

Bei den Regressionstests und dem Test der erweiterten Funktionalität nach der Erweiterung der Varianten wurde lediglich bei **PTS/RMI** ein Fehler entdeckt:

- Die Betätigung der Schaltfläche „Information anzeigen“ erzeugte kein neues Listenfenster mit Problemzusatzinformationen.
  Die Suche nach der Fehlerursache im Code verlief nach dem gleichen Schema wie beim Fehler „Sichtbarkeit des Hauptfensters“ im Test vor der Erweiterung:
  1. Prüfung der korrekten Weiterleitung der Benutzereingabe von der Benutzungsoberfläche (Swing Objekte der Klasse `ViewUniversal`) über die Zwischenschicht (`ControlProblemWindow`) an die Applikationsschicht (`AppProblemWindow`).
  2. Prüfung des korrekten Übergangs zum nächsten Applikationsobjekt, in diesem Fall von der Klasse `AppProblemWindow` zur neuen Klasse `AppMoreInfoOverview`.
  3. Prüfung der korrekten Darstellung der grafischen Benutzungsoberfläche eines Applikationsobjekts. Dazu müssen dieses mal die Konstruktoren der Klassen `AppMoreInfoOverview`, `ControlListOverview` und `ViewUniversal` geprüft werden.

  Schon im ersten Schritt konnte festgestellt werden, dass in der Klasse `ControlProblemWindow` das Benutzerereignis über die Betätigung der Schaltfläche nicht an das Applikationsobjekt (der Klasse `AppProblemWindow`) weitergegeben wird. Mit zwei neuen Zeilen in der Methode `actionPerformed` war dieser Fehler behoben.

8 Vergleich und Diskussion

8.1 Einschätzung der spezifizierten Funktionalität

Das spezifizierte „Problem Tracking“ System ist ein Informationssystem. Der Benutzer kann Daten aus einer Datenbasis einsehen (Use-Cases 3, 4, 5, 8), neue Daten der Datenbasis hinzufügen (Use-Cases 2, 6) und bereits bestehende Daten manipulieren (Use-Case 7). Die Verwaltung von Datensätzen, hier sind es Daten über Probleme, ist der Hauptzweck des „Problem Tracking“ Systems.

Es sind keine umfangreichen Berechnungen oder Prüfungen beim Hinzufügen oder Bearbeiten der Daten vorgesehen. Weder auf Seiten des Clients noch auf Seiten des Servers müssen komplexe algorithmische Teile ausgeführt werden. Lediglich Prüfungen auf die Einhaltung des Workflows werden im „Problem Tracking“ System durchgeführt. In realistischen Applikationen sind mehr Sicherheitsbeschränkungen (verschiedene Zugriffsrechte für Benutzer), Konsistenzprüfungen (zum Beispiel zur Prüfung von Zusammenhängen zwischen mehreren Datensätzen) und auch mehr algorithmische Teile (etwa für die Erstellung von statistischen Aussagen) zu erwarten.

Für das „Problem Tracking“ System sind nur einfache Beziehungen zwischen den Datensätzen (vor allem die Rollen „Problemersteller“ und „Problemlöser“) relevant. Dies hat zur Folge, dass von einer Datenbanktransaktion nur relativ wenige Tabellen betroffen sind.

Die gespeicherten Datensätze ändern sich durch Bearbeitung kaum. Ein Benutzer kann mit jeder Benutzeraktion nur eine Zeile der Tabelle „**`Problem`**“ verändern. Dabei wird, außer bei der Erstellung eines *Problems*, nur der Wert für den Zustand des *Problems* geändert.

Der „Problem Tracking“ Workflow unterbindet fast gänzlich Konflikte durch veraltete Daten („Lost Update“). Bis auf die Workflow-Aktion „Reopen“ im Zustand „Finished“, können alle Workflow-Aktionen durch genau einen *Bearbeiter* durchgeführt werden. Im laufenden Betrieb ist daher kaum mit solchen Konfilikten zu rechnen.

Die Ausgaben des Systems beschränken sich auf Text und Text in Tabellen. Dadurch eignet sich das „Problem Tracking“ System für eine Realisierung als Web-basierte Applikation. Sollen auch andere Elemente (Diagramme, Bilder, Töne, etc.) ausgegeben werden, ist eine Realisierung mit reinen HTML-Seiten (ohne Nutzung von Plug-Ins, JavaScript oder ähnlichem) nur mit sehr hohem Aufwand oder überhaupt nicht möglich.

Zusammenfassend kann festgestellt werden, dass die in dieser Diplomarbeit getroffenen Aussagen gut auf Entwicklung und Betrieb von wenig komplexen Informationssystemen übertragen werden können. Aussagen über Software, die umfangreiche Algorithmen erfüllen, große Mengen an Daten in einer Datenbank bewegen und analysieren, oder mit besonderen Benutzerinteraktionsformen arbeiten müssen, sind aus den Ergebnissen dieser Diplomarbeit nur eingeschränkt zu treffen.

8.2 Erstellungsaufwand

Während der Entwicklung der drei Varianten wurde die aufgewendete Arbeitszeit protokolliert. Nach jedem Arbeitstag wurde die auf Stunden gerundete Arbeitszeit in ein dafür vorgesehenes Dokument eingetragen.

Für die gemeinsame Vorbereitung (Spezifikation & Testplanung, Analyse des Workflows, Einrichtung der Datenbank) wurden zusammengezählt **37 Stunden** ermittelt. Die Arbeitszeit für die weitere Entwicklung der drei Varianten wurde in vier Kategorien unterteilt. Die vier Kategorien lauten:

- **Entwurf:** Tätigkeiten zur Planung oder Änderung der Systemarchitektur, von Klassendefinitionen oder von Methodensignaturen.
- **Client GUI:** Tätigkeiten, die Klassen der Benutzungsoberfläche, Interface-Klassen oder JSP-Dateien betreffen.
- **Client Logik:** Tätigkeiten an Klassen der Applikationsschicht oder der Zwischenschicht.
- **Server-Logik:** Tätigkeiten an der Server-Schicht und an Domain Klassen und an beiden Koordinatorklassen von PTS/JSP.

Die für jede Variante aufgebrachte Arbeitszeit von Beginn des Entwurfs bis zum Abschluss der ersten funktionalen Testphase hat sich wie folgt auf die vier Kategorien verteilt:

Variante	Entwurf	Client GUI	Client Logik	Server Logik	**Summe**
PTS/RMI	17 h	74 h	74 h	101 h	**266 h**
PTS/CORBA	21 h	19 h	88 h	124 h	**252 h**
PTS/JSP	9 h	25 h		39 h	**73 h**

Auffällig ist bei diesen Zahlen ist der Unterschied bei der Entwicklung der Client GUI zwischen PTS/RMI und PTS/CORBA und die insgesamt viel geringere Aufwand für die Entwicklung von PTS/JSP gegenüber PTS/RMI und PTS/CORBA. Die Faktoren, die einen Einfluss auf den Aufwand hatten und das Zustandekommen der oben genannten Auffälligkeiten erklären könnten, werden in den folgenden Kapiteln dargelegt.

8.2.1 Einflüsse durch den Entwicklungsverlauf

Diese Aufwandszahlen können so nicht unmittelbar für eine Aufwandsprognose vergleichbarer Projekte übernommen werden. Die Verzerrung der Daten, die unter anderem durch die Übernahme bereits existierender Software aus anderen Varianten verursacht wurden, ist beträchtlich. Im einzelnen müssen die folgenden Punkte bei der Interpretation der Daten berücksichtigt werden:

- Die Spezifikation wurde für drei Realisierungsansätze erstellt. Dadurch konnten Anforderungen nur bis zu einem gewissen Präzisierungsgrad in der Spezifikation dokumentiert und festgelegt werden. Dies hat zur Folge, dass im Vergleich zu einer detaillierten Spezifikation für einen einzigen Realisierungsansatz, weniger Testfälle aus den Anforderungen hergeleitet erstellt werden können. In realen Projekten steigt dadurch das Risiko, fehlerhafte Anforderungen erst spät im Projektverlauf zu entdecken.

- Entwurf, Datenbankanbindung und alle Java Klassen des Client-Programms wurden von PTS/RMI nach PTS/CORBA übernommen und angepasst. Da die Entwurfskonzepte von PTS/RMI und PTS/CORBA identisch waren und im Code von PTS/RMI bereits ihre Anwendung fanden, mussten die Beziehungen zwischen Klassen nicht noch mal entwickelt und codiert werden. Dies hat die Entwicklungszeit von PTS/CORBA in erheblich verkürzt.

- Datenbankanbindung und Domain Klassen von PTS/RMI wurden in PTS/JSP übernommen. Neu entwickelt wurde in PTS/JSP Entwurf, Benutzungsoberfläche und die Steuerung der Funktionalität. Die Entwicklungszeit von PTS/JSP wurde dadurch ebenfalls verkürzt.

- Der erste funktionale Test von PTS/RMI hat maßgeblich dazu beigetragen, dass viele Fehler bei PTS/CORBA und besonders bei PTS/JSP gar nicht erst gemacht wurden. Der Aufwand für die Korrektur dieser Fehler konnte bei PTS/CORBA und PTS/JSP gegenüber PTS/RMI eingespart werden.

- Konzepte aus dem Entwurf von [IBM_JSP_Prototyp] flossen in den Entwurf von PTS/JSP ein. Die Orientierung an einem funktionierenden Beispiel hat die Entwicklung von PTS/JSP zusätzlich beschleunigt.

Während PTS/RMI keine Erfahrung aus vorheriger Entwicklung nutzen konnte, haben die Teilprojekte zur Entwicklung von PTS/CORBA und PTS/JSP von der vorausgegangenen Entwicklung von PTS/RMI profitiert. PTS/JSP bezog zusätzlich Erfahrung in Form eines Prototyps bei der Entwicklung mit ein.

8.2.2 Einflüsse durch Entwicklererfahrung

Neben den Einflüssen durch den Projektverlauf, hatten auch bereits vorhandenes Wissen und Erfahrung aus früheren Projekten Auswirkungen auf die Entwicklungszeit. Die wichtigsten Wissensgrundlagen aus Theorie und Praxis vor Beginn der Diplomarbeit waren:

- Praktische Erfahrung zur Projektplanung, mit Use-Case Spezifikation und der Erstellung von Testfällen (besonders durch [SE 1998], [SP1 1998], [SP2 1999] und [SP3 1999]).
- Theoretisches Wissen und praktische Erfahrung über Software-Entwicklung und Entwurf mit der Programmiersprache Java (aus [KAOS 1998], [JavaKK 1999], [SP3 1999] und [IP 2000]).
- Theoretisches Wissen über SQL und praktische Erfahrung mit dem Zugriff auf eine Datenbank über JDBC aus [GDB 1999] und [IP2 2000].
- Praktisches Grundwissen über HTML (aus [SP2 1999], [SP3 1999]).

Wenig gelehrtes oder kein praktisches Wissen war vor Beginn der Diplomarbeit über CORBA, Java Applets, Java RMI und JavaServer Pages vorhanden.

Aus technischer Sicht haben vor allem die Entwürfe von PTS/RMI und PTS/CORBA, die Java Swing Benutzungsoberflächen, die PTS/JSP Benutzungsoberfläche und die Datenbankanbindung über JDBC von dem bereits vorhandenen Wissen profitiert. In welchem Maß sich dieser Umstand auf den Erstellungsaufwand ausgewirkt hat, ist jedoch nur sehr schwer abzuschätzen.

8.2.3 Einflüsse durch Technologien

Jede in den drei Varianten umgesetzte Technologie besitzt Eigenschaften, die (auch in Kombination mit den Wissensgrundlagen der Entwickler) den Erstellungsaufwand verringert oder vergrößert haben können:

- Java RMI fügte sich problemlos in bestehende Arbeitsabläufe zur Entwicklung von Java Software ein. Mit Java RMI konnte in gewohnter Weise mit Klassen der Java API gearbeitet werden. Es gab also keine subjektiv empfundene Verlangsamung der PTS/RMI Entwicklung durch die Nutzung von Java RMI.

- Bei Java Applets ist es nicht sinnvoll, die Benutzungsoberfläche nur auf die HTML-Seite zu beschränken, in der das Applet integriert ist (die Größe des grafischen Ausgabebereichs innerhalb der HTML-Seite wird fest angegeben). Stattdessen stellt das Applet nur eine Schaltfläche zum Starten der Applikation an. Die weitere Benutzungsoberfläche wird in eigenen Java Swing Fenstern dargestellt. Dieser Umstand führt dazu, dass die Benutzungsoberfläche von PTS/RMI auch für Java Applikationen geeignet ist. Die Wiederverwendung der Benutzungsoberfläche in PTS/CORBA war dadurch ohne Anpassungsaufwand möglich.

- Die Nutzung von CORBA erforderte im Gegensatz zu Java RMI eine Umstellung des Java Programmierstils. Der weitgehende Verzicht auf NULL-Zeigern innerhalb des Codes von PTS/CORBA erforderte an vielen Stellen – nicht nur beim Aufruf einer entfernten Methode – Prüfungen und entsprechende Fehlerbehandlungen. Diese für den Entwickler ungewohnt häufig vorkommenden Konstrukte wurden oft nachträglich in den Code integriert und waren dadurch bei den Integrationstests von PTS/CORBA die Hauptfehlerquelle. Offensichtlich wurde diese Eigenschaft von CORBA vor Beginn der Implementierung unterschätzt. Für Testdurchführung und Fehlersuche wurden bei PTS/CORBA am Client-Programm 14 Stunden und am Server-Objekt 26 Stunden mehr Arbeitszeit aufgewendet als in vergleichbaren Phasen in PTS/RMI.

- Die Benutzungsoberfläche von PTS/JSP konnte ohne große Verzögerungen zur Einarbeitung in die JSP-Syntax fertiggestellt werden. Bei der Entwicklung der JSP-Seiten wurde stets von einer HTML-Seite ausgegangen, in die dann mit Hilfe von JSP-Anweisungen Daten aus Java Objekten eingefügt werden. Die bestehenden Grundkenntnisse in HTML reichten für die schnelle Erstellung der schlichten PTS/JSP Benutzungsoberfläche aus.

- Die Kommunikation von PTS/JSP über HTTP hat den Verlust von Objektidentität zur Folge. Anstatt mit Objekten arbeiten zu können, musste die Ablaufsteuerung im JSP-Koordinator anhand der Parameterzeichenketten erfolgen. Die Erstellung dieses Codes wurde mehr als prozedurale denn als objektorientierte Lösung der Ablaufsteuerung empfunden. Die Vermengung der Programmierparadigmen hatte aus subjektiver Sicht jedoch zu keinen Verzögerungen der Entwicklung von PTS/JSP geführt.

8.2.4 Schätzungen für Entwicklungszeiten isolierter Realisierungen

Mit Kenntnis der Einflüsse auf die ermittelten Aufwandszahlen, können Vermutungen über Entwicklungszeiten bei alleiniger Umsetzung des CORBA oder des JSP Realisierungsansatzes aufgestellt werden:

- Eine **hypothetische alleinige Umsetzung des CORBA Ansatzes** (im folgenden HYP/CORBA genannt) hätte beim Entwurf Applikationsschicht, Zwischenschicht und Benutzungsoberfläche selbst entwickeln müssen. Dies wären schätzungsweise 9 zusätzliche Arbeitsstunden im Vergleich zum Aufwand des Entwurfs in PTS/CORBA.

- Die Entwicklung der Client GUI von HYP/CORBA hätte vermutlich etwas kürzer gedauert als bei PTS/RMI. Der Grund dafür sind die einfacheren Interface-Klassen von HYP/CORBA. Etwa 10% bis 20% des Codes könnten hier eingespart werden. Linear übertragen auf den Entwicklungsaufwand entspräche dies im Mittel ungefähr einer Ersparnis von 12 Stunden gegenüber der Client GUI Entwicklung von PTS/RMI.

- Die Zeit für die Entwicklung der Client Logik kann von PTS/CORBA übernommen werden, denn hier gab es durch die unterschiedlichen Domain Klassen relativ wenig Wiederverwendung von PTS/RMI Code

- Die Fertigstellung der Server Logik könnte in stärkerem Maße von einem spezialisierten Entwurf für CORBA profitiert haben. Die Datenbankanbindung hat etwa 50% der Server Logik Entwicklungszeit von PTS/RMI in Anspruch genommen. 50% der Server Logik wiederum befassen sich mit der technischen Datenbankanbindung, also Funktionen, die unabhängig von Domain Klassen zu erfüllen sind (dazu gehört beispielsweise die Generierung von SQL-Anweisungen). Ein denkbarer Aufwand für die Fertigstellung der Server Logik von HYP/CORBA könnte daher ungefähr 124 h (PTS/CORBA Server Logik) + 25 h (25% der PTS/RMI Server Logik) = 149 h betragen.

Zusammen könnte der Entwicklungsaufwand für HYP/CORBA also 37h (Vorbereitung der Datenbank und Spezifikation) + 30h (Entwurf) + 62h (Client GUI) + 88h (Client Logik) + 149h (Server Logik) = 366 h betragen haben.

- Eine **hypothetische alleinige Umsetzung des JSP Ansatzes** (HYP/JSP) müsste zu den Aufwandszahlen von PTS/JSP die Zeit für den Entwurf der Domain Klassen und der Server-Schicht bei PTS/RMI hinzuzählen. Applikationsschicht, Zwischenschicht und Benutzungsoberfläche haben schätzungsweise 9 Stunden des PTS/RMI Entwurfs beansprucht. Daher wird für die Entwurfsarbeitszeit von HYP/JSP ein Wert von 9 h (Entwurf von PTS/JSP) + 17 h (Entwurf von PTS/RMI) – 9 h (abzüglich der Zeit für nicht benötigte Schichten) = 17 h geschätzt.

- Domain Klassen und Datenbankanbindung konnten nahezu unverändert von PTS/RMI nach PTS/JSP übernommen werden. Für die Arbeitszeit zur Erstellung der Server Logik in HYP/JSP wird die Entwicklung der Server Logik von

PTS/RMI (101 h) in vollem Umfang zur Entwicklung der Server Logik in PTS/JSP (39 h) hinzugezählt. Damit ergibt sich für den Aufwand zur Erstellung der Server Logik von HYP/JSP ein Wert von 101 h + 39 h = 140 h.

Zusammengezählt ergibt dies für HYP/JSP eine Entwicklungszeit von 37h (Vorbereitung der Datenbank und Spezifikation) + 17h (Entwurf) + 25h (Client GUI) + 140h (Server Logik) = 219h.

Diese hypothetischen Zahlen sind nur dazu geeignet, um eine Einschätzung der Verhältnisse zwischen den Varianten zu bekommen. Vermutlich wäre bei der Realisierung eines einzelnen Ansatzes (bei vergleichbaren Voraussetzungen wie in dieser Diplomarbeit) die JSP-Variante am schnellsten realisiert gewesen. Die Java RMI und Applet Variante baut auf einem detaillierteren Entwurf und einer Swing Benutzungsoberfläche auf und benötigt für deren Umsetzung mehr Arbeitszeit.

Vermutungen über die alleinige Umsetzung der CORBA Variante sind aus den Zahlen kaum abzuleiten, da durch Fehler beeinflusste Aufwandsdaten (wie dies bei PTS/CORBA der Fall war) stark variieren können. Es scheint jedoch so zu sein, dass der Übergang von Java nach CORBA und zurück, zusätzlichen Aufwand mit sich bringt. Einleuchtend ist jedoch, dass im Vergleich zum Verbleib im Java Umfeld wie mit Java RMI, mit der zusätzlichen Technologie CORBA auch zusätzliche Fehlerquellen bei der Software-Entwicklung drohen.

8.3 Umfang des Quellcodes

Für die Ermittlung von Aussagen über den Umfang des Quellcodes wurden drei Metriken erhoben:

- Der **Code-Umfang** der Java Quelltextdateien wurde in „Delivered Source Instructions“ (DSI) gemessen. Die DSI Zahl wurde durch das Zählen von Zeilen ermittelt, die mindestens ein Semikolon-Zeichen enthalten. Dies ist im Falle dieser Diplomarbeit eine gute Näherung an die tatsächliche Zahl der Java Anweisungen, da für die Entwicklung der Java-Quelltexte eine Programmierrichtlinie (siehe Anhang) galt, die in jeder Zeile höchstens eine Anweisung zulässt.
  Auf IDL-Dateien lässt sich mit dieser Messung auch eine sinnvolle DSI Anzahl ermitteln, da die IDL Syntax der Java Syntax im Hinblick auf den Einsatz des Semikolon-Zeichens sehr ähnlich ist.

- Der **Gesamtumfang** ist die Summe der Dateigrößen aller Java Quelltextdateien oder JSP-Dateien in Byte. Mit Hilfe dieser Metrik, die mit der DSI Zahl korreliert ist, soll der Umfang der JSP-Dateien erfasst werden.

- Die **Anzahl der Dateien** enthält alle Dateien, die bei der Ermittlung des Code-Umfangs und des Gesamtumfangs berücksichtigt wurden. Damit kann festgestellt werden, in wie viele Einzelteile eine Variante unterteilt wurde.

Um Vergleichbare Werte zu erhalten, wurden die Metriken bei jeder Variante über eine eigene Menge von Dateien erhoben:

- Bei PTS/RMI wurden alle Java Quelltextdateien einbezogen. Generierten Java Code gibt es bei dieser Variante nicht.

- Bei PTS/CORBA wurden Metriken über alle Java Quelltextdateien erhoben, ausgenommen der Java Quelltextdateien, die aus der IDL Schnittstelle `CorbaServer.idl` generiert wurden. Stattdessen wurden Metriken über die Datei `CorbaServer.idl` selbst erhoben.

- Bei PTS/JSP wurden alle Metriken über alle Java Quelltextdateien erhoben. Von den JSP-Dateien wird nur der Gesamtumfang und deren Anzahl festgestellt.

Unmittelbar vor der Erweiterung (nach Abschluss der funktionalen Tests) wurden die folgenden Werte bei den einzelnen Varianten ermittelt:

Variante	Code-Umfang	Gesamtumfang	Anzahl Dateien
PTS/RMI	1898 DSI	126 056 Byte	40
PTS/CORBA	1608 DSI	130 704 Byte	35
PTS/JSP (Java Code)	1342 DSI	74 466 Byte	17
PTS/JSP (JSP-Dateien)		16 739 Byte	6

Der Unterschied im Code-Umfang zwischen PTS/RMI und PTS/CORBA lässt sich mit der unterschiedliche Modellierung der Domain Klassen erklären. In PTS/CORBA kann durch den Inhalt eines Domain Objekts navigiert werden. Soll auf alle Elemente eines Domain Objekts dieselbe Operation angewendet (zum Beispiel ein Eingabefeld erzeugt) werden, erfolgt dies mit Hilfe einer Schleife und im Vergleich zu PTS/RMI mit weniger Java Anweisungen. Der größere Gesamtumfang von PTS/CORBA gegenüber PTS/RMI ist auf die Nutzung von Arrays und Konstanten (also Daten) in solchen Schleifen zurückzuführen. Die größere Anzahl an Prüfungen auf NULL-Zeiger durch „`if`“- und „`else`“-Anweisungen, macht sich syntaxbedingt nicht in der DSI Anzahl bemerkbar. Auch dadurch unterscheiden sich die Verhältnisse von Code-Umfang zu Gesamtumfang von PTS/CORBA und PTS/RMI.

Der große Unterschied beim Gesamtumfang zwischen PTS/JSP einerseits und PTS/RMI und PTS/CORBA andererseits kann auf die bei PTS/JSP nicht vorhandene Java Swing Benutzungsoberfläche zurückgeführt werden. Der Code für eine Benutzungsoberfläche in Form von JSP-Dateien benötigt offensichtlich weniger und kleinere Dateien als eine Java Swing Benutzungsoberfläche. Zwar stehen dem Entwickler mit Java Swing Objekten mehr Gestaltungsmöglichkeiten offen als mit HTML-Tags, im Falle des „Problem Tracking“ Systems haben HTML-Tags für die Erstellung der einfachen Benutzungsoberfläche jedoch ausgereicht.

Die Tatsache, dass der Code-Umfang von PTS/JSP nicht in gleichem Maße von den entsprechenden Werten der anderen Varianten abweicht, wie dies beim Gesamtumfang der Fall ist, zeigt eine Folge des „Thin Client“ Prinzips. „Thin Clients“ sollen kein

Wissen über die von ihnen realisierte Applikation besitzen und somit für viele verschiedene Applikationen geeignet sein. Da das Client Programm (bei PTS/JSP der Web-Browser) keinerlei Information über die von ihm durchgeführte Applikation besitzt, muss bei jeder Anfrage (nach einer neuen JSP-Seite) der gesamte aktuelle Zustand der Applikation von Client zu Server übertragen, vom Server interpretiert und ausgewertet werden. Erst im Anschluss daran kann auf Seiten des Servers entschieden werden, welche JSP-Seite als nächstes zum Client übertragen werden soll.

Die Ablaufsteuerung erfolgt bei PTS/JSP also vollständig auf Seiten des Servers. In PTS/JSP benötigt der JSP-Koordinator (die Klasse `PtsJspController`) dafür ein wenig mehr als 700 DSI, also mehr als die Hälfte aller DSI von PTS/JSP. Dies entspricht in etwa dem Code-Umfang der Zwischenschicht in PTS/RMI und PTS/CORBA. Der Umstand, dass PTS/JSP keine Java Benutzungsoberfläche benötigt, verringert den Umfang an Anweisungen nicht im selben Maß wie dies beim Gesamtumfang der Fall ist. Ursache dafür ist die durch den Entwurf von PTS/JSP bedingte Ablaufsteuerung in der Klasse `PtsJspController`. Die oftmals komplexe Aufgabe, den Ablauf einer Applikation zu Steuern, lässt sich mit einer Vereinfachung der Benutzungsoberfläche nicht einfacher und mit erheblich weniger Code lösen, als bei einer herkömmlichen Benutzungsoberfläche

Wie sich diese Art der Ablaufsteuerung auf die Entwicklung von Systemen mit sehr umfangreicher Funktionalität und Benutzerinteraktion auswirken kann, ist eine weitere interessante Fragestellung zur JSP-Technologie. Aus den Ergebnissen dieser Diplomarbeit können in dieser Hinsicht jedoch keine Aussagen gemacht werden.

8.3.1 Produktivität

Im Zusammenhang mit dem Erstellungsaufwand kann der Umfang des Quellcodes Rückschlüsse auf die Produktivität der Software-Entwicklung liefern. Unmittelbar verwendbar sind jedoch nur die Zahlen für die Entwicklung von PTS/RMI (ohne die nachträgliche Erweiterung):

Der Gesamterstellungsaufwand von PTS/RMI:	37 h + 266 h	= 303 h
Die Produktivität betrug bei PTS/RMI also:	1898 DSI / 303 h	= 6,264 DSI/h

Unterstellt man diese Produktivitätszahl auch den anderen Varianten, ergeben sich für deren Entwicklungsaufwand die folgenden Werte:

PTS/CORBA:	1608 DSI / 6,264 DSI/h	= 256,7 h
PTS/JSP:	1342 DSI / 6,264 DSI/h	= 214,2 h

Ein Vergleich mit den hypothetischen Aufwandszahlen aus Kapitel 8.2.4 (HYP/CORBA: 366 h, HYP/JSP: 219 h) zeigt, dass die Streuung der Zahlen bei PTS/CORBA recht groß ist, während PTS/JSP in beiden Fällen klar unter dem Aufwand von PTS/RMI liegt.

8.4 Erweiterbarkeit und Wartbarkeit

Für jede Variante wurde die aufgewendete Arbeitszeit zur nachträglichen Erweiterung der Funktionalität erfasst. Da die gemessenen Werte wider Erwarten sehr klein sind, können daraus aber keine verlässlichen Aussagen abgeleitet werden.

Variante	Erweiterung	Test	**Summe**
PTS/RMI	5 h	2 h	**7 h**
PTS/CORBA	3 h	1 h	**4 h**
PTS/JSP	2 h	1 h	**3 h**

Die folgende Tabelle zeigt die Zunahme der Metrikwerte für den Umfang des Quellcodes nach der abgeschlossenen Erweiterung im Vergleich zu den Metrikwerten vor Beginn der nachträglichen Erweiterung:

Variante	Code-Umfang	Gesamtumfang	Anzahl Dateien
PTS/RMI	+ 203 DSI	+ 24 997 Byte	+ 4
PTS/CORBA	+ 192 DSI	+ 14 251 Byte	+ 3
PTS/JSP (Java Code)	+ 103 DSI	+ 7 704 Byte	+ 1
PTS/JSP (JSP-Dateien)		+ 683 Byte	+ 0

Die entspricht relativen Veränderungen von:

Variante	Code-Umfang	Gesamtumfang	Anzahl Dateien
PTS/RMI	+ 10,7 %	+ 19,8 %	+ 10,0 %
PTS/CORBA	+ 11,9 %	+ 10,9 %	+ 8,6 %
PTS/JSP (Java Code)	+ 7,7 %	+ 10,3 %	+ 5,9 %
PTS/JSP (JSP-Dateien)		+ 4,1 %	+ 0 %

Die Eigenschaften der einzelnen Varianten haben ähnliche Einflüsse auf die Wartung wie bereits auf die Software-Entwicklung.

- Die prozentuale Zunahme des Code-Umfangs von PTS/RMI und PTS/CORBA deutet an, dass die nachträgliche Integration von Funktionalität bei beiden Varianten im selben Maße aufwändig ist wie die Erstellung der bereits realisierten Funktionalität. Die geringere Zunahme des Code-Umfangs bei PTS/JSP ist auch auf die Klasse `ListSpec` zurückzuführen, die es dem JSP-Koordinator erlaubt, Listen von Problemzusatzinformationen weitgehend mit bereits existierendem Code zu behandeln.

- Die stärkere Zunahme des Gesamtumfangs von PTS/RMI gegenüber PTS/CORBA begründet sich aus der (bei dieser Erweiterung extrem effizienten) Modellierung der Domain Klassen. Die Einsparung der Java Benutzungsoberfläche gleicht bei PTS/JSP diese Eigenschaft von PTS/CORBA nahezu aus.

Um zu ermitteln, welche Teile jeder Variante in welchem Umfang von der Erweiterung betroffen waren, wurden der Code-Umfang nochmals getrennt für jede Schicht oder Komponente des Entwurfs ermittelt. Die Zunahme des Code-Umfangs hat sich wie folgt auf die einzelnen Schichten und Komponenten niedergeschlagen:

Variante	Schicht	Veränderung des Code-Umfangs
PTS/RMI	Server-Schicht	+ 75 DSI
	Domain Klassen	+ 17 DSI
	Applikationsschicht	+ 56 DSI
	Zwischenschicht	+ 27 DSI
	Benutzungsoberfläche	+ 28 DSI

PTS/CORBA	Server-Schicht	+ 60 DSI
	Domain Klassen	+ 5 DSI
	Applikationsschicht	+ 57 DSI
	Zwischenschicht	+ 28 DSI
	Benutzungsoberfläche	+ 38 DSI
	IDL-Interface	+ 4 DSI
PTS/JSP (Java Code)	Applikationskoordinator	+ 60 DSI
	Domain Klassen	+ 21 DSI
	JSP-Koordinator	+ 22 DSI

Aussagen über die Wartbarkeit einer Variante lassen sich auch hieraus ableiten:

- Der Zugriff auf die Datenbank erfordert für alle Varianten eine ähnliche Zunahme des Codes. Der Unterschied zwischen der Server-Schicht von PTS/RMI und dem Applikationskoordinator von PTS/JSP wird durch die das Interface `ServerAccess` verursacht, das in PTS/JSP nicht übernommen wurde, weil es nur zur Deklaration von Methoden dient.

- Die Applikationsschicht und Zwischenschicht von PTS/RMI und PTS/CORBA sind im gleichen Maße von der Erweiterung betroffen. Bei der Benutzungsoberfläche wirken sich die Unterschiede in der Datenmodellierung zwischen PTS/RMI und PTS/CORBA aus. Bei `ProblemMoreInfo` Objekten, die nur drei Attribute besitzen, erfordert eine Schleife mehr Anweisungen als ein direkter Zugriff auf die Attribute.

- Zu den Domain Klassen von PTS/RMI und PTS/JSP kam die neue Klasse `ProblemMoreInfo` hinzu. Zusätzlich musste in PTS/JSP die Klasse `ListSpec`, die der Ablaufsteuerung von Listenfenstern dient, auf die neue Klasse `ProblemMoreInfo` angepasst werden. In PTS/CORBA musste dagegen nur die IDL Schnittstelle und der Konstruktor von `AbstractObject` um eine Konstante und deren Behandlung erweitert werden.

- Der JSP-Koordinator von PTS/JSP benötigt weniger neue DSI als die Zwischenschicht und die Applikationsschicht von PTS/RMI und PTS/CORBA. Dabei muss aber beachtet werden, dass sich Änderungen in der Ablaufsteuerung von PTS/JSP nur auf eine einzige Klasse (`PtsJspController`) auswirken. Die neuen Klassen von PTS/RMI und PTS/CORBA benötigen allein zur Definition schon eine Mindestanzahl von neuen DSI.

Zusammenfassend trifft für die Erweiterbarkeit der drei Varianten nur eine Aussage zu: Die Erweiterbarkeit und Wartbarkeit hängt stark vom Entwurf der Variante und weniger von der verwendeten Technologie ab.

Während die Modellierung der Domain Klassen von PTS/RMI zu jeder neuen fachlichen Klasse eine neue Java Klasse erfordert, hat dies bei PTS/CORBA nur eine neue Konstante zur Folge. Beide Modellierungsarten sind sowohl mit Java RMI als auch mit CORBA möglich.

Der Entwurf von PTS/JSP unterscheidet sich im Punkte Wartbarkeit insofern von PTS/RMI und PTS/CORBA, dass er keine neuen Klassen für die Ablaufsteuerung bei neu hinzugekommener Funktionalität vorsieht. Alle HTTP-Anfragen laufen beim JSP-Koordinator der Klasse `PtsJspController` zusammen und müssen von dort ausgehend die Funktionalität auslösen. Mit zunehmender Funktionalität und müsste der Umfang des JSP-Koordinator ständig zunehmen, was die Verständlichkeit und Lesbarkeit der Klasse `PtsJspController` zunehmend senken würde. Dies ist jedoch durch den Entwurf bedingt und nicht Folge der JavaServer Pages Technologie. Auf längere Sicht gesehen wäre für JSP-Applikationen ein Entwurf sinnvoll, der ähnlich wie in PTS/RMI und PTS/CORBA eine klare Trennung der einzelnen Teilmengen an Funktionalität ermöglicht.

Der PTS/RMI Entwurf kann neue Funktionalität entweder in neue Klassen aufnehmen oder eindeutig einer existierenden Klasse der Applikationsschicht zuordnen, die dann geändert werden muss. Einfache Beziehungen zwischen den Klassen wie sie zwischen Applikationsschicht, Zwischenschicht und Benutzungsoberfläche von PTS/RMI und PTS/CORBA vorliegen, und eine klare Trennung der Aufgabengebiete erleichtern das Verständnis der Systemstruktur. Die Wartbarkeit wird also auch bei PTS/RMI und PTS/CORBA nicht maßgeblich von der verwendeten Kommunikationsform beeinflusst, sondern durch den Entwurf.

Die Erstellung eines entsprechenden Entwurfs für JSP-Applikationen ist (beispielsweise durch die Zusammenarbeit von mehrere Servlets bei der Ablaufsteuerung) möglich. Dies würde vermutlich auf ähnliche Strukturen wie bei den Client-Programmen von PTS/RMI und PTS/CORBA hinauslaufen. Aber wie sich eine JSP-Applikation mit einem solchen Entwurf auf die Software-Entwicklung auswirken wird und ob solche Strukturen mit JavaServer Pages überhaupt praktikabel und sinnvoll sind, kann mit PTS/JSP nicht prognostiziert werden.

9 Performanztests

Bei Literaturrecherchen im Verlauf dieser Diplomarbeit konnten nur sehr wenige wissenschaftliche Veröffentlichungen zu genau diesem Thema gefunden werden. Auch die Autoren von [Weyuker et al. 2000] kamen bei einer Suche nach wissenschaftlichen Arbeiten zum Thema Performanztests zu dieser Erkenntnis. Es gibt unter anderem in den Bereichen Hardware und Betriebssysteme mehrere „Benchmark"-Programme, die für die Beurteilung der Leistungsfähigkeit herangezogen werden. Aber worauf sich die Auswahl der von ihnen durchgeführten Tätigkeiten stützt, um realistische Vorhersagen über die Leistungsfähigkeit eines Systems machen zu können, wird nicht oder nur ungenügend erläutert. Mit Erkenntnissen aus den gefundenen Veröffentlichungen ([Endo et al. 1996], [Seltzer et al. 1999] und [Weyuker et al. 2000]) wurden die Tests der Varianten geplant und entworfen.

9.1 Ziele

Mit den durchgeführten Performanztests in dieser Diplomarbeit soll für jede Variante das Laufzeitverhalten getestet werden. Mit den Ergebnissen aus den Performanztests soll für jede Variante festgestellt werden:

1. Wie viele Benutzer gleichzeitig ohne Beeinträchtigung aktiv sein können.
2. Welches Verhältnis zwischen Antwortzeit und Last besteht.
3. An welchen Stellen des Systems (Datenbank, Netzwerk, Benutzungsoberfläche) wie viel Rechenzeit benötigt wird.
4. Welche Optimierungsmöglichkeiten empfehlenswert sind.

Eine Beeinträchtigung der Benutzer liegt vor, wenn die Zeitspanne von der Betätigung einer Schaltfläche bis zur vollständigen Darstellung eines Fensters zu groß wird. In [Herczeg 1994] wird für die Antwortzeit (vereinfacht: die Zeitspanne zwischen Eingabezeitpunkt und Ausgabezeitpunkt) interaktiver Systeme im allgemeinen eine Höchstgrenze von 4 Sekunden gefordert. In [Endo et al. 1996] wird die Latenzzeit (die Zeit, die ein Benutzer auf eine Eingabe wartet) als bedeutender eingeschätzt als der Durchsatz (die maximale Anzahl von Operationen pro Zeiteinheit) eines Systems. Denn die Latenzzeit wird vom Benutzer direkt empfunden, während aus dem Durchsatz nur beschränkt die Latenzzeit und damit Zufriedenheit des Benutzers ermittelt werden kann. Allerdings sollte die Latenzzeit im Idealfall von der Denkzeit des Benutzers getrennt werden, was jedoch die Beobachtung von Benutzerverhalten erfordert.

- Als **Beeinträchtigungskriterium** in den Performanztests wird eine Antwortzeit von mehr als 4 Sekunden definiert.

9.2 Testumgebung

Für die Performanztests wurde die in Kapitel 3.2 beschriebenen Rechner genutzt. Die Veränderung einer DB2 Datenbankeinstellung war die einzige Abweichung von den Standardeinstellungen der beteiligten Software. Nach ersten Tests musste der Wert für „Application Heap Size" der [PTS Datenbank] auf 32 Megabytes erhöht werden. Sonst führten viele gleichzeitige Anfragen an die DB2 Datenbank zum Absturz des Java Server-Programms wegen fehlerhafter Datenbankverbindungen.

9.3 Simulation von Benutzerverhalten

Da Performanztests Vorhersagen über das künftige Verhalten eines Systems im Betrieb treffen sollen, ist die Ermittlung des künftigen Nutzungsprofils eines Systems wesentlich für die Simulation der Systemlast. Für die Nachbildung von realitätsnahem Benutzerverhalten für die Nutzung eines „Problem Tracking“ Systems, wurde ein Entwickler eines ähnlichen Systems der IBM befragt. Aus diesem Gespräch konnten einige Aussagen für die Erstellung von Szenarien herangezogen werden:

1. *„50% aller Anfragen durchsuchen den Problemdatenbestand nach großen Kategorien.“*
   Die Anfrageergebnisse sind im Falle der in dieser Diplomarbeit entstandenen Varianten, Listen mit vielen *Problemen*.

2. *„Bei 30% aller Anfragen suchen Benutzer nach Problemen, die mit ihrer Benutzerkennung verbunden sind.“*
   Dies entspräche der Anforderung von Listen, in denen alle von einem Bearbeiter erstellten oder zu lösenden *Probleme* enthalten sind.

3. *„20% aller Anfragen führen Workflow-Aktionen aus. 75% der erstellten Probleme werden akzeptiert, die anderen abgelehnt. Dabei werden etwa 40% der akzeptierten Probleme mindestens einmal neu zugewiesen. Zu 40% von allen akzeptierten Problemen werden mindestens einmal weitere Informationen angefordert. Von den 25% abgelehnten Problemen wird etwa die Hälfte wieder geöffnet.“*
   Aus diesen Aussagen können einige wahrscheinlich auftretende Sequenzen von Workflow-Aktionen, denen ein *Problem* von der Erstellung bis zur Beendigung im System unterworfen wird, definiert werden.

4. *„Die Benachrichtigung der betroffenen Benutzer bei Problemänderungen erfolgt per E-Mail.“*
   Eine Synchronisation zwischen mehreren Bearbeitern erfolgt außerhalb des Systems. Eine schnelle Reaktion auf veränderte Problemzustände ist nicht zu erwarten, da die Probleme zuerst bearbeitet werden müssen, bevor im „Problem Tracking“ System eine Workflow-Aktion durchgeführt werden kann. Deswegen wird kein großer Prozentsatz der eingetragenen Bearbeiter gleichzeitig zur Arbeit mit dem „Problem Tracking“ System erwartet.

Analyse der Datenbankinhalte

Um dementsprechende Anfrageergebnisse erhalten zu können, wurde der Inhalt von [PTS Datenbank] analysiert. Aus Vertraulichkeitsgründen werden die konkreten Werte nicht genannt, sondern lediglich umschrieben.

- Insgesamt sind ungefähr 20,000 Probleme und 600 *Bearbeiter* in der Datenbank gespeichert.
- Die Suchanfrage nach den drei wichtigsten Kategorien der Spalte `severity` ergeben Listen mit 1644 (im folgenden **S1** genannt), 10138 (**S2**) und 77 (**S3**) Problemen.

- Weitere Kategorien ergeben sich auch durch die Einteilung von *Problemen* in Problembereiche. Einem Problembereich sind im Median 12 (**Bmed**), im Durchschnitt 37 (**Bavg**) und für 95% aller Problembereiche weniger als 408 (**B_95**) *Probleme* zugeordnet.
- *Bearbeiter* haben im Median 15 (**EM**) und im Durchschnitt 41 (**EA**) *Probleme* erstellt. Es gibt 8 *Bearbeiter* mit genau 15 erstellten *Problemen* und 7 *Bearbeiter* mit genau 41 erstellten *Problemen*.
- *Bearbeiter* sollen im Median 23 (**LM**) und im Durchschnitt 34 (**LA**) *Probleme* lösen. 7 *Bearbeitern* wurden genau 23 und 4 *Bearbeitern* genau 34 *Probleme* zugeordnet.

Simulation von lesenden Benutzerzugriffen

Aus der Analyse der Datenbankinhalte wurden drei Verhaltensklassen zur Simulation von Benutzern erstellt. Die Ergebnisse der Benutzeranfragen – Listen von Problemen – sollten in ihrer Größe dem Median und dem Durchschnitt entsprechender Realer Anfragen entsprechen. Das Verhalten für die Klassen, die auf jeder Variante Performanztests durchführen werden, wird vor deren Implementierung definiert:

- In der Klasse **`BigProblemReader`** werden abwechselnd Listen von *Problemen* mit einem bestimmten `severity` Wert, und von *Problemen* mit einem bestimmten Problembereich, vom Server angefordert.

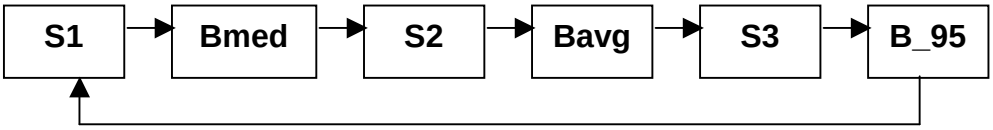


Abbildung 25 Anforderung von Problemlisten durch die Klasse `BigProblemReader`

Durch die Abwechslung zwischen den Kategorien von Anfrage zu Anfrage soll das Zwischenspeichern („caching“) von Anfragen und Ergebnissen durch die Datenbank erschwert werden.

- In der Klasse **`SmallProblemReader`** werden abwechselnd von einem *Bearbeiter* erstellte und zu lösende *Probleme* angefordert.

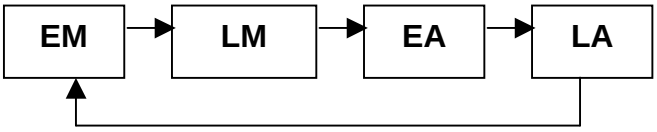


Abbildung 26 Anforderung von Problemlisten durch die Klasse `SmallProblemReader`

In dieser Klasse wechseln sich nicht nur die Kategorien ab, auch die *Bearbeiter* einer Kategorie werden zyklisch geändert.

Problemlöser-Werte = { „8 **EM** *Bearbeiter*“, „5 **EA** *Bearbeiter*“ }
Problemersteller-Werte = { „7 **LM** *Bearbeiter*“, „4 **LA** *Bearbeiter*“ }

Bei jeder nachfolgenden Anfrage nach einer Liste derselben Größe, wird ein anderer *Bearbeiter* simuliert, und damit eine andere Menge von Problemen in der Liste angezeigt.

Simulation von schreibenden Benutzerzugriffen

Für die Simulation von *Bearbeitern*, die *Probleme* erzeugen und mit Workflow-Aktionen verändern, wird die Klasse **`ProblemProcessor`** angelegt. *Probleme* werden erzeugt und mit zyklisch wechselnden Sequenzen von Workflow-Aktionen bearbeitet.

Es werden insgesamt 8 verschiedene Sequenzen von Workflow-Aktionen durchgeführt:

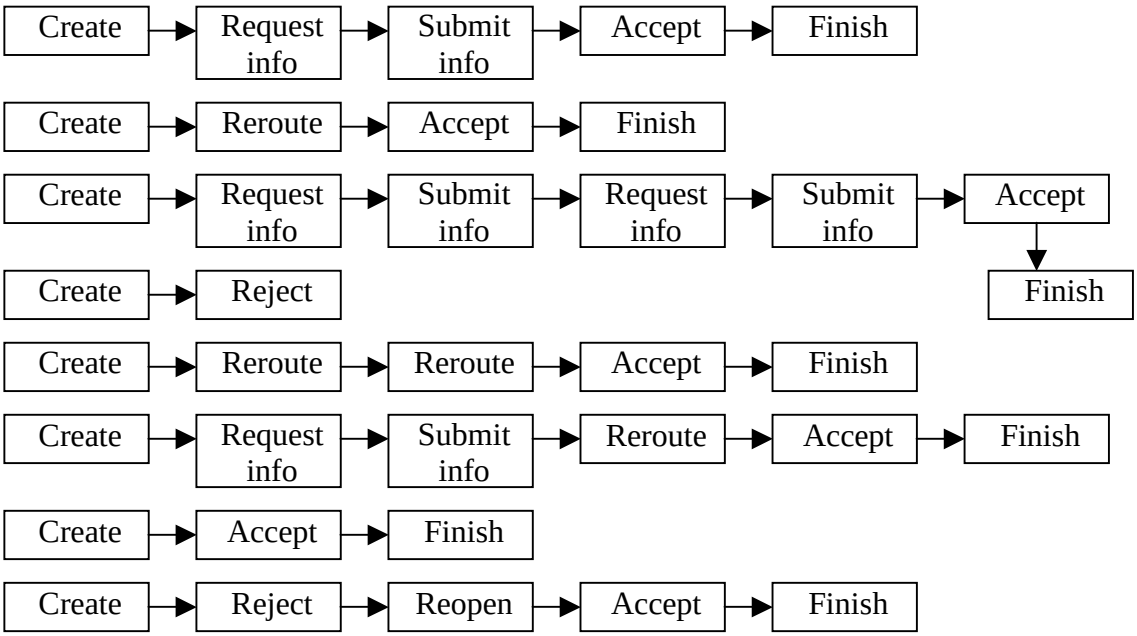


Abbildung 27 Sequenzen von Workflow-Aktionen in der Klasse `ProblemProcessor`

Die Durchführung der Workfow-Aktionen erfolgt bei PTS/RMI oder PTS/CORBA auf eine andere Art und Weise als bei PTS/JSP. Da bei PTS/JSP nur ein einziges Fenster (das des Web-Browsers) zur gleichen Zeit nutzt, muss die Durchführung der Workflow-Aktionen mit mehr Fenstern vom Server rechnen (beispielsweise für Meldungen), als Server Interaktionen bei PTS/RMI oder PTS/CORBA notwendig sind.

Für alle mit der Klasse `ProblemProcessor` erzeugten und bearbeiteten *Probleme* gilt im Durchschnitt:

- 75% der *Probleme* werden ohne Ablehnung akzeptiert.
- 25% der *Probleme* werden abgelehnt. Die Hälfte dieser *Probleme* wird erneut geöffnet.
- Ca. 43% der akzeptierten *Probleme* wird ein neuer Problemlöser zugewiesen.
- Zu ca. 43% der akzeptierten *Probleme* wird Problemzusatzinformation angefordert.
- 50% der abgelehnten *Probleme* werden erneut geöffnet.

Es wurde bei der Definition der `ProblemProcessor` Sequenzen darauf geachtet, das in den Aussagen der Befragung beschriebene Benutzerverhalten möglichst genau nachzubilden.

9.4 Messmethoden

Auf Seiten des Servers werden folgende Zeiten ermittelt:

- Die Gesamtzeitspanne, die eine Client Anfrage vom Server-Objekt oder dem JSP-Koordinator bearbeitet wird.
- Die Zeitspanne für die Suche in der Datenbank (SQL SELECT-Anweisungen) – die Suchzeit. Zur Datenzeit wird gleichzeitig die Art der Anfrage protokolliert, damit die Zeiten unterschiedlicher Methoden auseinandergehalten werden können.
- Die Zeitspanne zum Lesen oder Schreiben von Daten aus der beziehungsweise in die Datenbank – die Datenzeit.
- Die Zeit, die zum Protokollieren der Zeiten gebraucht wurde – die Protokollzeit. Diese dient nur zur Kontrolle, ob die Protokollierung die Ergebnisse verfälscht.

Dazu wird der Code an einigen Stellen instrumentiert (Zeitstempel wurden mit der Java Anweisung `System.currentTimeMillis` ermittelt):

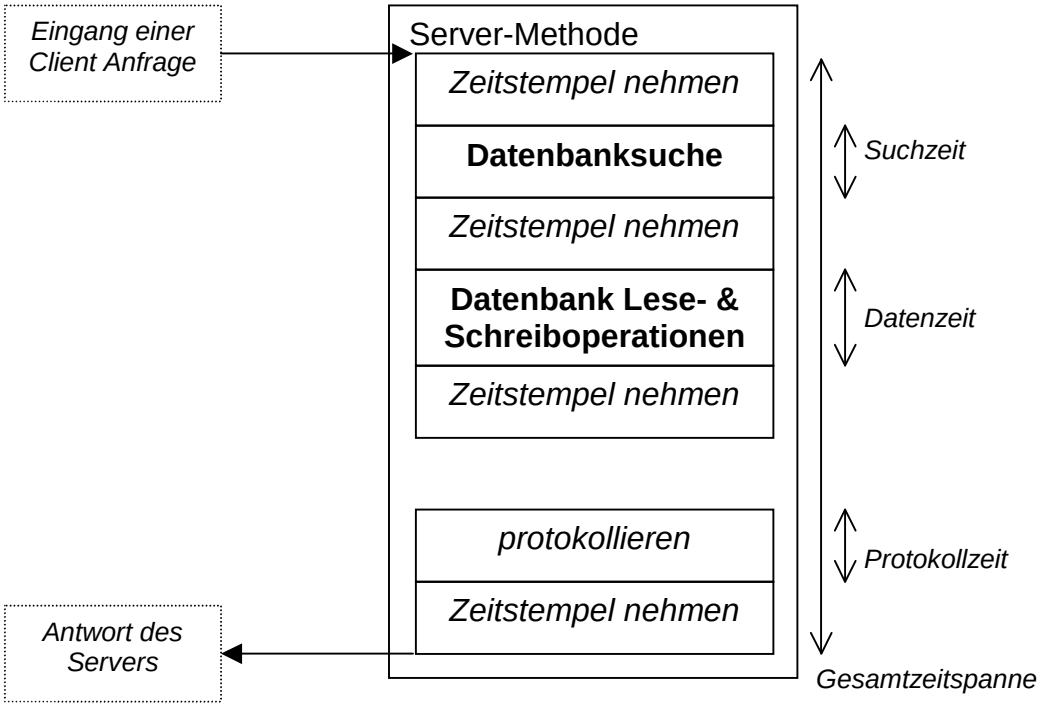


Abbildung 28 Instrumentierung des Server Quellcodes

Bei PTS/JSP werden zwei Protokolldateien benötig. Für die Gesamtzeit eine Protokolldatei des JSP-Koordinators und für Suchzeit und Datenzeit eine Protokolldatei des Applikationskoordinators.

Die Methode, die den Protokollvorgang durchführt ermittelt die Protokollzeit selbst:

Abbildung 29 Ablauf der Protokollmethode

Auf Seiten des Clients werden Server Anfragen durch unabhängig voneinander ablaufende Prozesse („Threads“) einer Java Applikation gestellt. Die Realisierung als eine Java Applikation war notwendig, weil als Applet keine Dateien für die Speicherung der Messdaten erzeugt werden können und für die Sicherheitsprüfungen und den gleichzeitig laufenden Web-Browser Rechenzeit und Hauptspeicherplatz belegt wird.

Die einzelnen Threads werden mit einem zentralen Programm sowohl erzeugt als auch in gleichmäßigen Abständen zueinander gestartet. Damit zur Laufzeit für die Datensammlung möglichst keine Rechenzeit verbraucht wird, werden genügend große Arrays vor dem Start des Tests allokiert. Diese werden dann im Verlauf des Tests mit Zeitstempeln versehen. Es werden so viele Arrays erzeugt, wie unterschiedliche Anfragen an den Server gestellt werden können, um Antwortzeiten einer Klasse von Server Anfrage zuordnen zu können.

Die Threads der Klasse `TestThread` bleiben nach einer Server Anfrage für eine einstellbare Zeit inaktiv (*Denkzeit*). Dies soll eine Denk- und Eingabezeit des Benutzers bis zur nächsten Server Anfrage simulieren. Um jegliche Synchronisation zu vermeiden, wird eine ebenfalls einstellbare feste Anzahl von Server Anfragen durchgeführt. Nach Durchführung aller Server Anfragen wird ein Objekt (`Finish Listener`), das die beendeten Threads zählt benachrichtigt. Sobald alle Threads beendet sind, liest der „`Finish Listener`“ die Zeitstempeldaten aus den Arrays aller Threads aus und speichert sie in einer Datei.

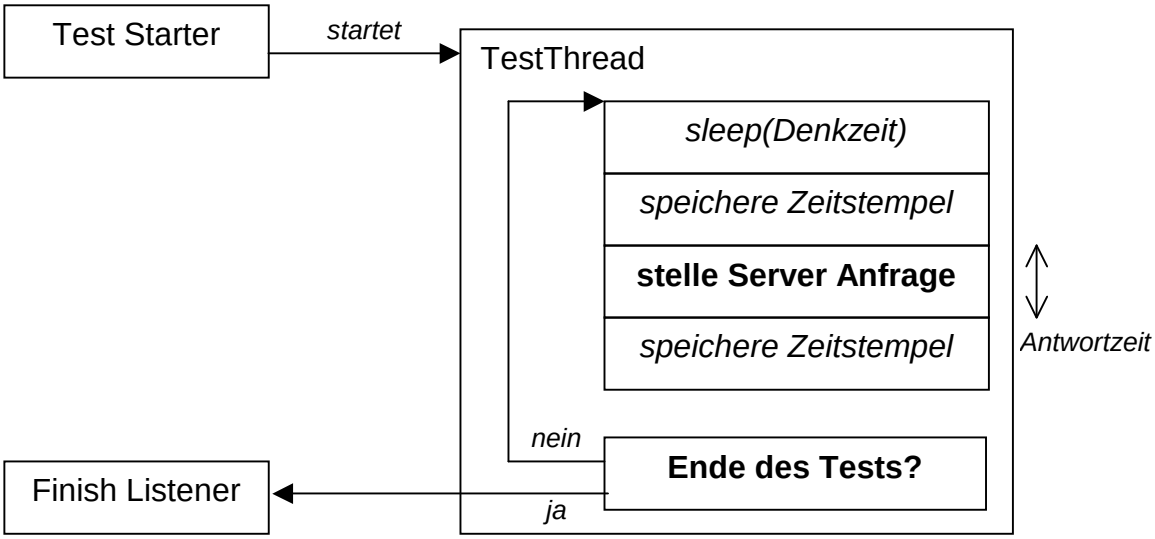


Abbildung 30 Mechanismen der Performanztest-Threads

Für jede Variante wurden drei Unterklasse von `TestThread` erstellt. Diese führten dann die in Kapitel 9.3 beschriebenen Klassen an Benutzerverhalten aus. Bei PTS/CORBA und PTS/RMI wurden Methoden des Server-Objekts aufgerufen, bei PTS/JSP war die Entwicklung eines eigenen HTTP-Clients notwendig, der ohne Analyse des Inhalts (aus Performanzgründen) und mit Unterstützung des Cookie-Protokolls (zur Identifikation der Sitzung) Anfragen mit der POST und GET Methode von HTTP den Server ansprachen.

Server Anfragen für Listenausschnitte bei PTS/CORBA `TestThread` Prozessen hatten genau wie im Client-Programm einen Kopiervorgang direkt im Anschluss an den Erhalt der Objektreferenz zur Folge.

Die Implementierung der Benutzerverhaltensklasse `ProblemProcessor` erstellt und bearbeitet *Probleme* nur so, dass die Größen der Listen bei Anfragen von Objekten der Klassen `BigProblemReader` und `SmallProblemReader` während des Testverlaufs konstant bleiben.

9.5 Planung

In einem Verhältnis der Benutzerverhaltensklassen von

- 50% `BigProblemReader` **(B)**
- 30% `SmallProblemReader` **(S)**
- 20% `ProblemProcessor` **(P)**

wird die Anzahl der simulierten Benutzer schrittweise erhöht. Dabei werden auf dem Rechner „IBM Personal Computer 300XL“ (**C1**) etwa doppelt so viele `TestThread` Prozesse ablaufen wie auf dem Rechner „IBM Thinkpad 765D“ (**C2**).

Die Datenbank wird vor jedem neuen Performanztest wird [PTS Datenbank] mit den Daten aus einer Kopie (zum Stand nach der Erweiterung von PTS/JSP) gefüllt. Dadurch sollen Effekte durch eine immer größer werdende Anzahl von *Problemen* vermieden werden. Die Hilfsprogramme `rmiregistry` für PTS/RMI und `tnameserv` für PTS/CORBA werden vor jedem neuen Performanztest der betroffenen Variante neu gestartet und nach dem Test beendet.

Die *Denkzeit* der `TestThread` Prozesse beträgt 4 Sekunden. Es werden 1000 Server Anfragen pro `TestThread` Prozess gestellt.

Durchgeführt werden die Tests mit 30, 60 und 90 simulierten Benutzern. Dies entspricht 5%, 10% und 15% aller in der Datenbank enthaltenen *Bearbeiter*, die gleichzeitig und mit minimaler Verzögerung Listen von *Problemen* anfordern oder *Probleme* bearbeiten. Vermutlich wird diese Größenordnung im realen Betrieb nie erreicht, aber zur Ermittlung der Leistungsgrenze ist diese Größenordnung geeignet.

Testfall	Variante	**B (C1)**	**B (C2)**	**S (C1)**	**S (C2)**	**P (C1)**	**P (C2)**
1	PTS/RMI	10	5	6	3	4	2
2	PTS/CORBA						
3	PTS/JSP						
4	PTS/RMI	20	10	12	6	8	4
5	PTS/CORBA						
6	PTS/JSP						
7	PTS/RMI	30	15	18	9	12	6
8	PTS/CORBA						
9	PTS/JSP						

Die mit 30, 60 und 90 simulierten Benutzern geplanten Testfälle werden höchstens zu einer Anfragefrequenz von 7,5 Anfragen/s, 15 Anfragen/s und 22,5 Anfragen/s auf Seiten des Servers führen.

9.6 Messergebnisse

Die geplanten Performanztests führten zusammengefasst zu den folgenden Ergebnissen (alle Angaben in Millisekunden).

Testfall	∅ Suchzeit	∅ Datenzeit	∅ Gesamtzeit	∅ Antwortzeit	∅ Netzwerkzeit
1	342	178	532	660	128
2	351	182	548	707	159
3	335	186	534	692	158
4	671	439	1126	1277	151
5	734	453	1205	1404	199
6	683	445	1145	1394	249
7	1958	883	2865	3036	171
8	2209	901	3139	3321	182
9	2053	996	3071	3391	320

Der Durchschnitt der Protokollierzeit betrug in allen Testfällen höchstens 4 Millisekunden. Der Protokolliervorgang hat die Testergebnisse also nur unmerklich beeinflusst und wird nicht aufgeführt.

Bei den ermittelten Messwerten fällt auf, dass Suchzeit und Datenzeit in allen Varianten besonders von der steigenden Zahl simulierter Benutzer betroffen ist. Bei Betrachtung einzelner Messreihen ist auch aufgefallen, dass es mit zunehmender Benutzerzahl öfter statistische Ausreißer gibt, die für die Bearbeitung einer Anfrage eine besonders lange Zeitspanne benötigen. In der Datenzeit besteht ungefähr ein Verhältnis von 10:1 zwischen der Zeit für schreibenden und für lesende Zugriffe. Dieser Umstand ist auch auf die hohe Zahl an Problembearbeitungen zurückzuführen. Schreibende Zugriffe auf die Datenbank müssen schließlich sicherstellen, dass eine Änderung nach Abschluss der Transaktion in allen weiteren Transaktionen wirksam ist.

Die gemessene Netzwerkzeit steigt bei der PTS/JSP Variante stärker an, als bei den anderen Varianten. Dies kann auch darauf zurückgeführt werden, dass die `TestThread` Klassenimplementierungen für die PTS/JSP Variante mehrere Operationen durchführen müssen, um die Daten eines HTTP-Datenstroms einzulesen. Erst nachdem der HTTP-Datenstrom vollständig eingelesen wurde, wird ein Zeitstempel für die Antwortzeit gesetzt.

Die Befürchtung, das Kopieren von CORBA Daten blockiere den ORB für andere Aufgaben hat sich mit dieser Performanzmessung nicht maßgeblich bewahrheitet. Die Netzwerkzeiten von PTS/CORBA sind zwar etwas höher als bei PTS/RMI, aber in Größenordnungen, die für einen Benutzer kaum bemerkbar sind.

9.6.1 Behinderung von Benutzern

Um die für den Benutzer auftretende Antwortzeit zu ermitteln, wurde die Benutzungsoberfläche jeder Variante auf jedem der drei zur Verfügung stehenden Rechner ohne anderweitige Belastung auf ihren Anteil an der Antwortzeit untersucht. Dazu wurde die Zeit von der Betätigung einer Schaltfläche bis zum Erscheinen eines neuen Fensters manuell gemessen. Die Messungen sind natürlich sehr unscharf, sie reichen jedoch aus, um eine grobe Abschätzung der Antwortzeit für die Benutzungsoberfläche liefern zu können. Dabei wurden die folgenden per Hand gestoppten Werte ermittelt:

- Auf der IBM Netfinity Server Maschine ergaben sich über alle Varianten und Fenster hinweg Wartezeiten unter 0,2 Sekunden. Der Start des PTS/RMI Applets (die Anzeige der HTML-Seite, in der das Applet integriert ist) benötigte allerdings 2,5 Sekunden.
- Auf dem IBM Personal Computer 300 XL Rechner konnte für PTS/JSP in einigen Fenstern eine Verzögerung von 0,8 Sekunden ermittelt werden. Die Fenster von PTS/RMI und PTS/JSP erschienen durchweg nach 0,4 Sekunden oder darunter. Der Start des PTS/RMI Applets benötigte auf dieser Maschine etwa 3,0 Sekunden.
- Auf dem IBM 765D Thinkpad ergaben sich für PTS/JSP Fenster Wartezeiten bis zu einer Sekunde. Die Fenster von PTS/RMI und PTS/CORBA erschienen nach etwa 0,5 Sekunden. Der Start des PTS/RMI Applets benötigte hier die längste Zeitspanne: 5 Sekunden.

Zusammen mit den Messergebnissen kann vermutet werden, dass bei einer durchschnittlichen Antwortzeit von 3,3 Sekunden, Benutzer von wenig leistungsfähigen Rechnern bei mehr als 90 aktiven *Bearbeitern* längere Antwortzeiten als 4 Sekunden zu erwarten haben. Bei leistungsfähigeren Rechnern ist ebenfalls bei 90 *Bearbeitern* jedoch nur bei Nutzung von PTS/JSP mit längeren Antwortzeiten als 4 Sekunden zu rechnen. Die Varianten PTS/RMI und PTS/CORBA benötigen auch bei 90 aktiven *Bearbeitern* nur in Ausnahmefällen mehr als 4 Sekunden zur Darstellung eines Fensters.

9.6.2 Verteilung der Rechenzeit

Der Großteil der Verzögerung einer Antwort des Servers auf Client Anfragen entsteht durch das Warten auf Datenbankzugriffe. Vor allem die Datenbankanfragen für die Kategorie S2 (mit 10138 Problemen als Ergebnis) haben lange Wartezeiten zur Folge. Lesende Zugriffe auf die Datenbank sind dagegen relativ schnell beendet. Dies liegt zum Beispiel bei der `getProblem` Methode daran, dass als Ergebnis höchstens eine Zeile zurückgeliefert wird, da nach einem Primärschlüssel gesucht wird. Suchzugriffe mit Primärschlüsseln als Suchkriterium können von Datenbanken besonders gut optimiert werden.

Der relative Anteil der vom Java Code des Servers benötigten Rechenzeit betrug bei den Messungen im Durchschnitt nie mehr als 3 % der Server Rechenzeit. Mit zunehmender Dauer der Datenbankzugriffe sinkt diese Zahl noch weiter. Die Rechenzeit, die im Java Code des Servers verbraucht wird ist im Vergleich zur Wartezeit auf die Datenbank nicht ausschlaggebend.

Die Auswirkungen auf den Performanztest konnte für die folgenden Beteiligten Systeme nicht ermittelt werden:

- Der JSP-Container [Tomcat 3.1.1] verwaltet die JavaBeans einer JSP-Seite und Sitzungsinformationen. Eventuell haben auch Konfigurationseinstellungen von [Tomcat 3.1.1] Auswirkungen auf den Performanztest gehabt.
- Die Netzwerkanbindung über 100Mbit Ethernet des Servers an 10Mbit Token-Ring können Verzögerungen oder Unregelmäßigkeiten bei der Netzwerkzeit verursacht haben.
- Für [PTS Datenbank] wurden die DB2 Standardeinstellungen bis auf den Wert der „Application Heap Size“ beibehalten. Andere DB2 Einstellungen könnten zu besseren Suchzeiten und Datenzeiten führen.

9.7 Optimierungsmöglichkeiten

Optimierungsmöglichkeiten existieren vor allem für die Beschleunigung der Datenbankzugriffe und die Übertragung der CORBA Tabellen:

- Um die Zahl der Datenbankzugriffe zu minimieren, wäre ein Cache-Zwischenspeicher oder eine Persistenzschicht sinnvoll, die einen Teil der *Probleme* für lesende Zugriffe im Speicher hält. Besonders für die lesenden Zugriffe auf die Datenbank, die 70% der Benutzerinteraktion ausmachen könnte diese Maßnahme bereits einen spürbaren Performanzgewinn bringen.

- Für einen effizientern Zugriff auf die Datenbank wäre neben „Prepared Statements“ auch die Verwendung von „Embedded SQL“, also die Nutzung von selbst definierten Prozeduren, denkbar.

- Für die Übertragung von CORBA Daten könnte ein codierter Zeichenstrom genutzt werden, der alle Informationen auf einmal enthält. Bei CORBA steht auch die Nutzung eines anderen ORB oder sogar einer anderen Programmiersprache für die Implementierung des Servers als Option zur Verfügung.

9.8 Interpretation

Die Messwerte sind nur schwer miteinander zu vergleichen. PTS/JSP besitzt beispielsweise den Nachteil, dass bei jeder Benutzerinteraktion die gesamte Benutzungsoberfläche im HTML-Text mit übertragen wird. Dennoch kann aus den Messwerten geschlossen werden, dass alle Realisierungsansätze in ihren Grenzen performant sein können.

Die Optimierung der Datenbankzugriffe sollte eher in Angriff genommen werden, als eine Umstellung auf eine der hier vorgestellten Technologien.

10 Schlussfolgerungen und Ausblick

Die in dieser Diplomarbeit untersuchten Technologien sind für den praktischen Einsatz in einer Client-Server Anwendung geeignet. Jede Technologie eignet sich für spezielle Aufgabengebiete besonders:

- Java RMI bietet Vorteile bei Anwendungen, die rein in Java entwickelt werden. Alle Eigenschaften von Java, alle für Java-Entwickler gewohnten Klassen und APIs können ohne zusätzlichen Aufwand in eine verteilte Anwendung einbezogen werden.

- CORBA/IIOP bietet sich als offener und herstellerunabhängiger Standard für die Entwicklung von Systemen in mehreren Programmiersprachen oder die Nutzung bereits bestehender Software an. Die Auswahl eines ORBs bietet bei der Software-Entwicklung zusätzliche Freiheiten.

- JavaServer Pages verbinden die Vorteile aus der Programmiersprache Java mit der einfachen Darstellung einer Benutzungsoberfläche in der Sprache HTML. JavaServer Pages Anwendungen setzen für ihre Benutzung nur einen beliebigen Web-Browser voraus.

Jedoch besitzt auch jede Technologie ihre eigenen Einschränkungen oder Schwierigkeiten bei der Software-Entwicklung:

- Java RMI beschränkt die Kommunikation rein auf die Programmiersprache Java. Zwar ist bereits eine Kombination mit dem CORBA Standard geplant, aber bis zur Realisierung dieser Pläne bleibt Java RMI ein auf Java spezialisierter Kommunikationsmechanismus.

- CORBA/IIOP birgt Risiken bei der Umsetzung zwischen der abstrakten Schnittstellenbeschreibung in IDL und der konkreten Implementierung. Diese Kontaktfläche zwischen CORBA und Programmiersprache kann eine Quelle für viele Fehler sein.

- JavaServer Pages geben durch die Benutzerinteraktion über das HTTP-Protokoll die Objektidentität der verwendeten Java-Objekte auf. Eine Verbesserung in dieser Hinsicht könnte die „Enterprise JavaBeans“ Technologie bringen, die ähnlich zum JSP-Container für JavaServer Pages die Verwaltung der JavaBeans übernimmt. Für eine gute Wartbarkeit ist auf die sorgfältige Planung des Entwurfs zu achten.

Mit der in dieser Diplomarbeit realisierten Software konnten Erfahrungen gesammelt und Risiken bei der Software-Entwicklung und Wartung entdeckt werden. Mit Hilfe dieser Erkenntnisse können Schwierigkeiten bei der Entwicklung von Software mit den verwendeten Technologien Java RMI, CORBA/IIOP und JavaServer Pages umgangen oder wenigstens angegangen werden.

Anhang

Anhang A Hilfsmittel

Studium

[GDB 1999] B. Mitschang; **Vorlesung Grundlagen von Datanbanken und Informationssystemen;** Universität Stuttgart; Wintersemester 1999/2000

[GVS 1999] K. Rothermel; **Vorlesung Grundlagen der Verteilten Systeme;** Universität Stuttgart; Sommersemester 1999

[IP2 2000] Kristian Tandjung, Markus Wolf, Dietrich Weber; **Vega-Entry Prototyp;** Industriepraktikum bei debis Systemhaus MEB; September 2000 bis November 2000

[JavaKK 1999] Ralf Reißing; **Kompaktkurs Java;** Universität Stuttgart; 12. bis 16. Juli 1999

[KAOS 1998] R. Melchisedech, S. Krauß, J. Ludewig; **Vorlesung Konzeption und Aufbau objektorientierter Software;** Universität Stuttgart; Wintersemester 1998/1999

[RE 1999] R. Koschke; **Vorlesung Reengineering;** Universität Stuttgart; Wintersemester 1999/2000

[SE 1998] J. Ludewig, P. Mandl-Striegnitz; **Vorlesung Software Engineering;** Universität Stuttgart; Wintersemester 1998/1999

[SP1 1998] **Studienprojekt 1 – „Sesamalyzer“;** Universität Stuttgart; Wintersemester 1998/1999 und Sommersemester 1999

[SP2 1999] **Studienprojekt 2 – „Riven“ (Wartung des Graphen-Editors “Rigi”);** Universität Stuttgart; Wintersemester 1999/2000 und Sommersemester 2000

[SP3 1999] **Studienprojekt 3 – Verkehrsverteilung und Verkehrsaufteilung in einem intermodalen Prognosesystem;** Universität Stuttgart; Wintersemester 1999/2000 und Sommersemester 2000

Literatur

[Eckel 2000] Bruce Eckel; **Thinking in Java;** Prentice Hall; 24. April 2000

[Endo et al. 1996] Yasuhiro Endo, Zheng Wang, J. Bradley Chen, Margo Seltzer; **Using Latency to Evaluate Interactive System Performance;** Harvard University; 1996

[Gokhale et al.1996] Aniruddha Gokhale, Douglas C. Schmidt; **Measuring the Performance of Communication Middleware on High-Speed Networks;** ACM SIGCOMM Conference 1996

[Herczeg 1994] Michael Herczeg; **Software-Ergonomie: Grundlagen der Mensch-Computer-Kommunikation;** Addison-Wesley; 1994

[Hirano et al. 1998] Satoshi Hirano, Yoshiji Yasu, Hirotaka Igarashi; **Performance Evaluation of Popular Distributed Object Technologies for Java;** 1998

[OmgIDLJava] Object Management Group; **IDL to Java Language Mapping Specification;** Juni 2000

[iX_JSP_Tutorial] Lars Röwekamp, Peter Roßbach; **JSP-Tutorial (Teile 1, 2 und 3);** Heise Verlag, iX 7/2000, iX 8/2000, iX 9/2000

[Seltzer et al. 1999] Margo Seltzer, David Krinsky, Keith Smith, Xiaolan Zhang; **The Case for Application-Specific Benchmarking;** Harvard University; 1999

[SunJavaEEGuide] Sun Microsystems; **The Java™ 2 Enterprise Edition Developer's Guide;** Version 1.2.1, Mai 2000

[SunJavaTutorial] Sun Microsystems; **The Java Tutorial;** 13. Oktober 2000

[SunJSPSpec] Eduardo Pelegrí-Llopart, Larry Cable; **JavaServer Pages™ Specification;** Sun Microsystems; Version 1.1, 30. November 1999

[SunJSPTutorial] Sun Microsystems; **JavaServer Pages™ Tutorial;** 2000

[SunRmiSpec] Sun Microsystems; **Java™ Remote Method Invocation Specification;** Revision 1.7, December 1999

[SunServletSpec] James Duncan Davidson, Danny Coward; **Java™ Servlet Specification;** Sun Microsystems; Version 2.2, Final Release, 19. Dezember 1999

[Weyuker et al. 2000] Elaine J. Weyuker, Filippos I. Voklos; **Experience With Performance Testing of Software Systems: Issues, an Approach, and Case Study;** IEEE Transactions on Software Engineering Vol. 26 No. 12; Dezember 2000

[Züllighoven 1998] Heinz Züllighoven; **Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- & Material-Ansatz;** dpunkt Verlag; 1998

Betreuung

[PTS Datenbank] Otto Wohlmuth; **SQL Datenbanktabellendefinition;** 2. Februar 2001

[IBM_JSP_Prototyp] IBM ODF-Team; **CRS2000 Prototyp Quellcode;** 2000

[PTS Workflow] Otto Wohlmuth; **Problembearbeitungs-Workflow Diagramm;** 6. Februar 2001

Dritt-Software

[Apache 1.3.14] Apache; **Apache Webserver;** Version 1.3.14; *http://www.apache.org*

[Tomcat 3.1.1] Apache; **Tomcat JSP Container;** Version 3.1.1; *http://jakarta.apache.org/tomcat*

[IBM_DB2] IBM Corporation; **IBM DB2 Universal Database for Linux;** Version 7.0, Extended Enterprise Edition

[Netscape 4.72] Netscape Communications Corporation; **Netscape Communicator;** Version 4.72

[SunHtmlConv 1.3] Sun Microsystems; **Java™ Plug-in HTML Converter;** Version 1.3

[SunJavaRun 1.3] Sun Microsystems; **Java™ 2 Runtime Environment Standard Edition;** Version 1.3.

[SunJavaSdk 1.3] Sun Microsystems; **Java™ 2 SDK, Standard Edition;** Version 1.3

[SuseLinux 7.0] SuSE GmbH; **SuSE Linux Distribution;** Version 7.0

Anhang B Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

Kristian Tandjung

Ein Musterexemplar liegt bei der die Arbeiten entgegennehmenden Stelle zur Einsicht aus.