Bericht über das

Praktikum bei der sd&m AG

vom 15. Juli 2002 bis zum 15. Oktober 2002

Jörg Rüdenauer
Matrikelnr. xxxxxxx

1. Formale Angaben

1.1 Verfasser

Name: Jörg Rüdenauer
Matrikelnummer: 1907911
Fachsemester: 9
Adresse: Im Kechler 7/2
71642 Ludwigsburg
E-Mail-Adresse: j.ruedenauer@gmx.de

1.2 Zeitraum

Ich war bei sd&m als Praktikant vom 15. Juli 2002 bis zum 15. Oktober 2002 – jeweils einschließlich – beschäftigt, also drei Monate lang. Dabei hatte ich eine wöchentliche Arbeitszeit von 40 Stunden.

1.3 Beschäftigende Firma

Die Firma ist die sd&m AG (software, design & management). Ich war in der Niederlassung Stuttgart beschäftigt. Adresse:
sd&m AG
Industriestr. 5
70565 Stuttgart
Mehr Informationen zu sd&m gibt es im Internet unter http://www.sdm.de/.

1.4 Ansprechpartner

Ansprechpartner für allgemeine Fragen bezüglich des Praktikums ist
Thorsten Feuerabend
sd&m AG
Industriestr. 5
70565 Stuttgart
Tel.: 0711-78324-161
E-Mail: thorsten.feuerabend@sdm.de

Bei Fragen zur Thematik (fachlich oder technisch) des Praktikums ist der Ansprechpartner
Dr. Kai Höfler
sd&m AG
Industriestr. 5
70565 Stuttgart
Tel.: 0711-78324-142
E-Mail: kai.hoefler@sdm.de

2. Einleitung

2.1 Das Praktikum

Ich habe dieses Praktikum zwischen dem 15. Juli 2002 und dem 15. Oktober 2002 (jeweils eingeschlossen) bei sd&m in Stuttgart absolviert. Die sd&m AG ist eine große deutsche Softwarefirma mit ca. 900 Mitarbeitern an 8 Standorten, die Individualsoftware für betriebliche Informationssysteme entwickelt – mehr Informationen gibt es im WWW unter http://www.sdm.de. Während des Großteils der Zeit war ich mit der Entwicklung einer Java-Applikation beschäftigt, die zur Automatisierung des Testprozesses in einem Projekt von sd&m dienen soll. Dabei entstanden knapp 20.000 DLOC und ca. 50 Seiten Dokumentation (ohne javadoc).
Für Nachfragen stehe ich natürlich gerne zur Verfügung, am besten per E-Mail: j.ruedenauer@gmx.de. Ansprechpartner bei sd&m sind auf der vorigen Seite genannt.

2.2 Zur Struktur des Berichtes

In Kapitel 3 folgen detaillierte Berichte zu meinen Tätigkeiten in den einzelnen Wochen bei sd&m. Um den Wochenberichten einen gewissen Zusammenhalt zu geben, habe ich in ihnen jeweils abgeschlossene Themen beschrieben, obwohl ich diese Themen häufig auch ein oder manchmal zwei Tage in der vorhergehenden oder nachfolgenden Woche bearbeitet habe. In Einzelfällen ist dann auch die Lösung für ein bestimmtes Problem in einer Woche beschrieben, obwohl sie erst einige Zeit später entwickelt wurde, um ein flüssiges Lesen des Berichtes zu ermöglichen. Ich habe an diesen Stellen auch explizit darauf hingewiesen.
Auf Auszüge aus dem Quellcode oder der Dokumentation habe ich meistens verzichtet, um den Praktikumsbericht nicht zu groß werden zu lassen und weil der Abstraktionsgrad des Berichtes generell höher ist. Nur an einzelnen Stellen sind exemplarische Programmfragmente enthalten.

3. Wochenberichte

1. Woche (KW 29): Einarbeitung; OT: Excel-Tool zur Zeitabrechnung

Die Woche begann (natürlich) zuerst damit, dass ich mich mit den Gegebenheiten bei sd&m vertraut machte. Ich lernte also einige Kollegen kennen, richtete meinen Arbeitsplatz – insbesondere auch meinen Rechner – auf meine Bedürfnisse ein, erforschte das Intranet und die Dateistruktur auf den Netzlaufwerken usw. Hierbei interessierte mich besonders das Themengebiet *Quasar* – *Qua*litäts-*S*oftware-*A*rchitektur – der internen Forschungsabteilung von sd&m (sd&m research). Da die später im Praktikum von mir entwickelte Anwendung nach den Prinzipien von Quasar aufgebaut ist, beschreibe ich es hier näher.
Hervorstechendes Merkmal von Quasar ist die Aufteilung einer Architektur in Komponenten, womit hier einfach größere, austauschbare Einheiten der Software gemeint sind, die (mindestens) eine Schnittstelle exportieren. Jegliche Software (Code, Schnittstellen, Komponenten) wird bei Quasar einer bestimmten Kategorie zugeordnet:

- A-Software ist anwendungsspezifisch und meist nicht wiederverwendbar
- T-Software ist anwendungsunabhängig, dafür technikspezifisch und häufig wiederverwendbar
- 0-Software ist unabhängig von Anwendung und Technik – ideal wiederverwendbar, aber für sich alleine nutzlos
- AT-Software ist sowohl von Anwendung als auch von Technik abhängig, schlecht wartbar und schlecht wiederverwendbar und sollte daher vermieden werden
- R-Software schließlich dient der Umwandlung von Daten in externe Repräsentationen wie Datenbanken oder XML. Idealerweise kann sie automatisch generiert werden.

Die Komponentenarchitektur ist dabei nicht nur in Hinblick auf möglichst einfache Wartung (auch Austauschbarkeit) und Wiederverwendung entworfen, sondern legt auch besonderes Augenmerk auf die Robustheit. Dafür wird der Begriff des *Desasters* eingeführt, der eine 'illegale', d.h. unvorhergesehene Ausnahmesituation kennzeichnet, die z.B. durch einen Programmierfehler herbeigeführt wurde. Indem eine Komponente Desaster explizit erkennt und behandelt, kann die Stabilität der Software stark erhöht werden.
Weiterhin definiert Quasar einen Standardweg zum Entwurf von Software-Architekturen. Die Architektur wird dabei aufgeteilt in verschiedene Sichten:

- die Architektur der technischen Infrastruktur (TI-Architektur), welche die verwendete Hardware und Systemsoftware – auch die Aufteilung auf verschiedene Tier – beschreibt
- die Anwendungsarchitektur (A-Architektur), die alle fachlichen, also auf das spezielle Aufgabengebiet der Anwendung bezogenen, Elemente beschreibt
- die Technikarchitektur (T-Architektur). Diese beschreibt die Komponenten der Software anwendungsunabhängig. Die A-Architektur wird in sie 'hineingesteckt'; die T-Architektur beschreibt also insbesondere auch das Zusammenspiel von 0-, A- und T-Software.

Schließlich definiert Quasar auch Standardarchitekturen, auf die ich hier aber nicht weiter eingehe.

Da Kai Höfler, der mir die eigentliche Aufgabe im Praktikum erläutern sollte und auch sonst mein erster Ansprechpartner werden sollte, sich in der Woche noch im Urlaub befand, bekam ich zur Überbrückung der Zeit eine andere kleine Aufgabe. Hierbei ging es um ein sd&m-internes Werkzeug zur Auswertung von Zeiterfassungsdaten. Die Mitarbeiter bei sd&m tragen ihre Arbeitszeiten in Excel-Tabellen ein; diese werden dann teils von Visual Basic-Makros, teils von Anwendungslogik auf einem zentralen Server ausgewertet. Ich sollte nun Verbesserung an einem VB-Modul vornehmen, das aus einer Tabelle mit einzelnen Zeitdaten Summen je nach Mitarbeiter und Monat erstellt und in einer neuen Tabelle anzeigt. Hierbei waren zum Einen kleinere Fehler zu beseitigen, zum Anderen sollte die neue Tabelle in Zukunft Summenformeln statt dem berechneten Ergebnis enthalten, um manuelle Änderungen zu ermöglichen.
Dazu war es leider erforderlich, den Algorithmus zu ändern: der bisherige Algorithmus lief pro Kategorie (Mitarbeiter, Monat) jeweils einmal über die Daten und sortierte sie vorher nach dieser Kategorie, um die Summation zu vereinfachen. Für Summenformeln war es aber notwendig, dass die Position der Daten sich nicht veränderte. Deshalb speicherte ich die Zellen pro Kategorie während des Durchlaufs in einer Hash-Struktur und fügte sie am Ende in die Formel ein. Somit wäre auch ein einzelner Durchlauf für alle Kategorien möglich gewesen; ich verzichtete aber auf diese Verbesserung, weil sie auch einen strukturellen Umbau des Moduls bedeutet hätte.
Leichte Probleme bei der Aufgabe hatte ich nur, weil ich vorher fast nicht mit Visual Basic gearbeitet hatte – deshalb musste ich zum Beispiel erst lernen, dass eine Änderung von Daten innerhalb einer *Collection* nicht aktualisiert wird (ein Entfernen aus der *Collection* und erneutes Einfügen ist dafür notwendig) oder dass die Summenformel entgegen der Hilfe auf

Englisch notiert werden muss und höchstens 30 Argumente haben darf. Nachdem diese Probleme jedoch behoben waren, führte ich einen Regressionstest mit Hilfe der alten Version des Moduls durch (ich konnte die Ergebnisse aber nur stichprobenhaft kontrollieren, weil eine automatische Kontrolle nicht möglich war), um sicher zu gehen, dass keine falschen Daten berechnet wurden. Durch die Abschaltung der visuellen Darstellung der einzelnen Operationen konnte ich das Modul außerdem um ca. 30% beschleunigen.

2. Woche (KW 30): Einarbeitung, Grobarchitektur

In dieser Woche lernte ich die Aufgabe kennen, die mich die restliche Zeit des Praktikums beschäftigen sollte. Das Themengebiet ist die Verbesserung des Testprozesses in einem sehr großen Projekt von sd&m, dem Projekt GO (Global Ordering). Beim GO-Projekt geht es grob gesagt um den Prozess der Bestellung von PKWs bei DaimlerChrysler – wie schon der Name sagt, weltweit, und mit verschiedensten Teilkomponenten wie einer Werksschnittstelle zur direkten Planung der Produktion oder einer Baubarkeitsprüfung beim Eingang einer Bestellung. Die Anwendungslogik ist dabei in Cobol implementiert und auf einem zentralen Server (dem sog. Host) installiert, auf den von verschiedenen Clients aus zugegriffen wird, die hauptsächlich in Java realisiert sind. Das Projekt läuft schon seit 1996 und ist in eine Reihe von Unterprojekten aufgeteilt; ich war allerdings keinem speziellen Unterprojekt zugeordnet (organisatorisch gehörte ich zum Chefdesign).

Nachdem ich einen groben Überblick über GO gewonnen hatte, machte ich mich mit dem bisherigen Testprozess vertraut, dessen Ablauf in Abbildung 3.1 dargestellt wird. Hierzu redete ich einerseits mit einigen anderen Mitarbeitern im GO-Projekt, ließ mir den Prozess erklären und den Ablauf eines typischen Tests zeigen; andererseits las ich auch einige Dokumente, die den Vorgang oder dafür verwendete Werkzeuge beschrieben.
Getestet wurden einzelne Module oder Subsysteme (Tests des Gesamtsystems wurden hier nicht betrachtet) bisher folgendermaßen:

1. Bei Subsystem-Tests sind benötigte Daten teilweise schon in einer Access-Datenbank abgelegt. Aus dieser werden Insert-Skripte automatisch generiert, die der Tester auf den Server überträgt und dort ausführt, um den richtigen DB-Zustand herzustellen
2. Bei diesen Tests werden Eingabedateien ebenfalls automatisch aus einer Access-DB generiert. Bei anderen Subsystem-Tests oder Modultests muss der Entwickler die Eingabedateien manuell erstellen.
3. Diese Eingabedatei, die sowohl die Aufrufparameter für die Operationen der Module als auch die erwarteten Ausgaben enthält, wird dann von Hand auf den Server übertragen
4. Dort muss man sie in ein spezielles, vom Betriebssystem verarbeitbares Format umwandeln
5. Jetzt wird – ebenfalls von Hand – ein spezielles Programm, der sog. Universaltesttreiber, gestartet. Dieser liest die Eingabedatei, initialisiert je nach Operation benötigte Querschnittskomponenten (z.B. einen Meldungsverwalter oder eine Prozesssteuerung; die dafür notwendigen Daten liest er aus einer eigenen DB-Tabelle), ruft die gewünschten Operationen auf, vergleicht Soll- mit Istausgaben und erzeugt eine Ausgabedatei mit den Ergebnissen.
6. Das Format dieser Datei muss wieder geändert werden; jetzt kann der Entwickler in ihr und evtl. in der Datenbank die Ergebnisse des Tests kontrollieren.
7. Schließlich wird die Ausgabedatei zurück auf den Client geschoben und der Versionskontrolle zugeführt.

Ein paar Monate vorher hatte eine sd&m-interne Analyse schon ergeben, in welcher Weise dieser Testprozess hauptsächlich verbessert werden könnte. In der Münchner sd&m-Niederlassung hatte kurz vor mir auch ein Werkstudent, Jan Czeika, mit ersten Maßnahmen begonnen. Bei seiner Arbeit ging es hauptsächlich darum, eine einheitliche (Access-) Datenbank zu schaffen, in der die Testdaten für alle Teilprojekte und alle Tests gehalten werden konnten.

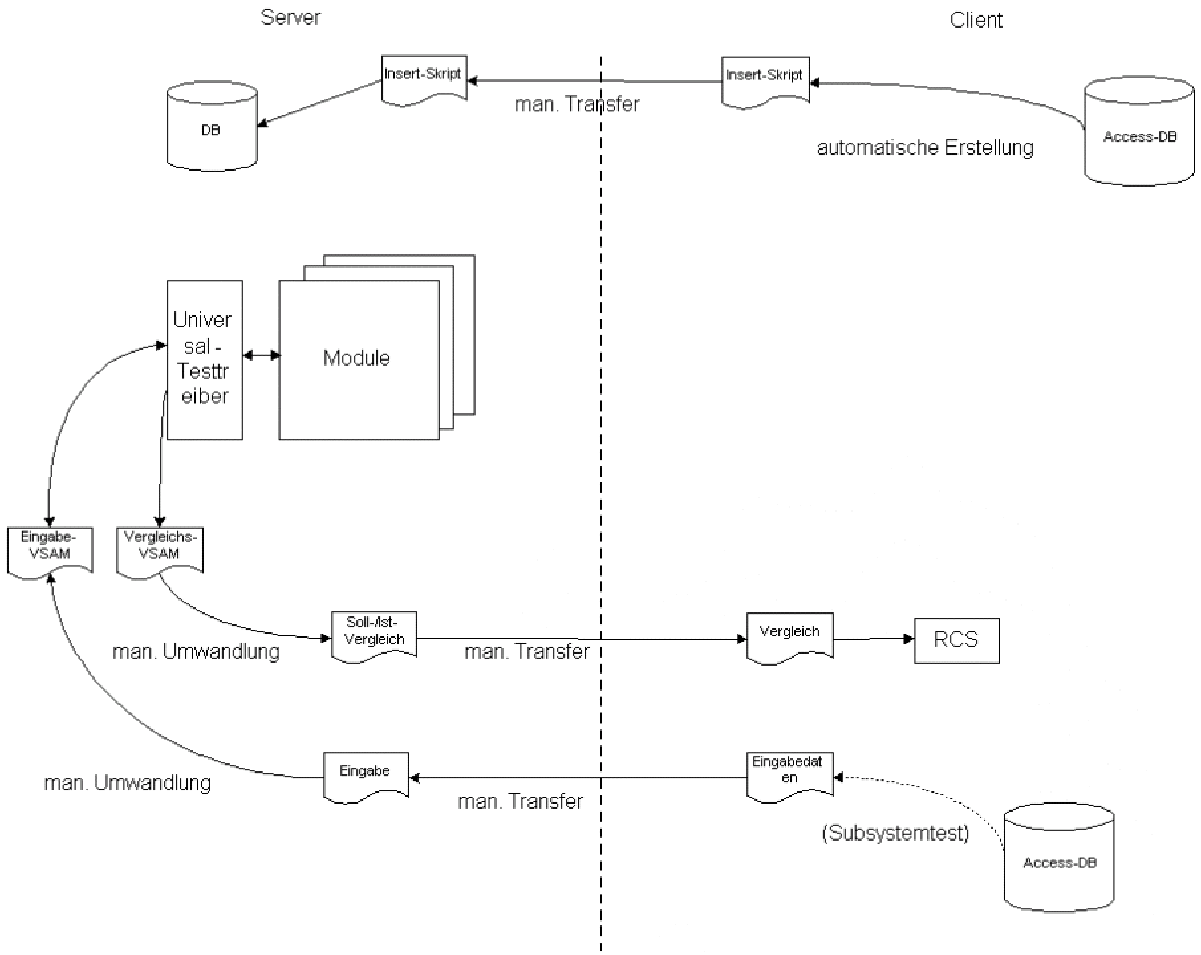

Abbildung 3.1: Bisheriger Testablauf

Meine Aufgabe bestand nun darin, den eigentlichen Testvorgang größtmöglich zu automatisieren. Hierzu sollte der Universaltesttreiber abgelöst werden durch ein spezielles Modul, einen sog. *Test-Dispatcher*. Dieses Modul sollte dann per TCP über ein Socket von außen angesprochen werden können und nicht nur die eigentlichen Testdaten, sondern auch alle Informationen bezüglich der Querschnittskomponenten vom Client geliefert bekommen. Nach dem Modulaufruf sollte der Test-Dispatcher selbst nur die Istausgaben an den Client zurück liefern und diesem den Vergleich mit den Sollausgaben überlassen.
Ich sollte aber nicht diesen Test-Dispatcher selbst entwickeln (wofür auch erst mal eine gewisse Einarbeitung meinerseits in den verwendeten Cobol-Dialekt und die allgemeine Entwicklungsumgebung auf Host-Seite erforderlich gewesen wäre), sondern das Programm auf dem Client, das den Test-Dispatcher aufruft – im Folgenden Test-Client genannt. Dieses Programm sollte dem Tester eine graphische Benutzungsoberfläche bieten, Testdaten sowohl aus den bestehenden Eingabedateien (evtl. leicht angepasst wegen der Querschnittskomponenten) als auch aus der neuen Datenbank lesen können und die Ergebnisse des Tests möglichst komfortabel visualisieren. Eine Automatisierung der DB-Vorbereitung wäre zwar ebenfalls er-

wünscht, ist aber technisch nicht machbar, weil auf die Datenbank von außen nicht zugegriffen werden kann. Den Testablauf mit dem neuen Test-Client zeigt Abbildung 3.2.

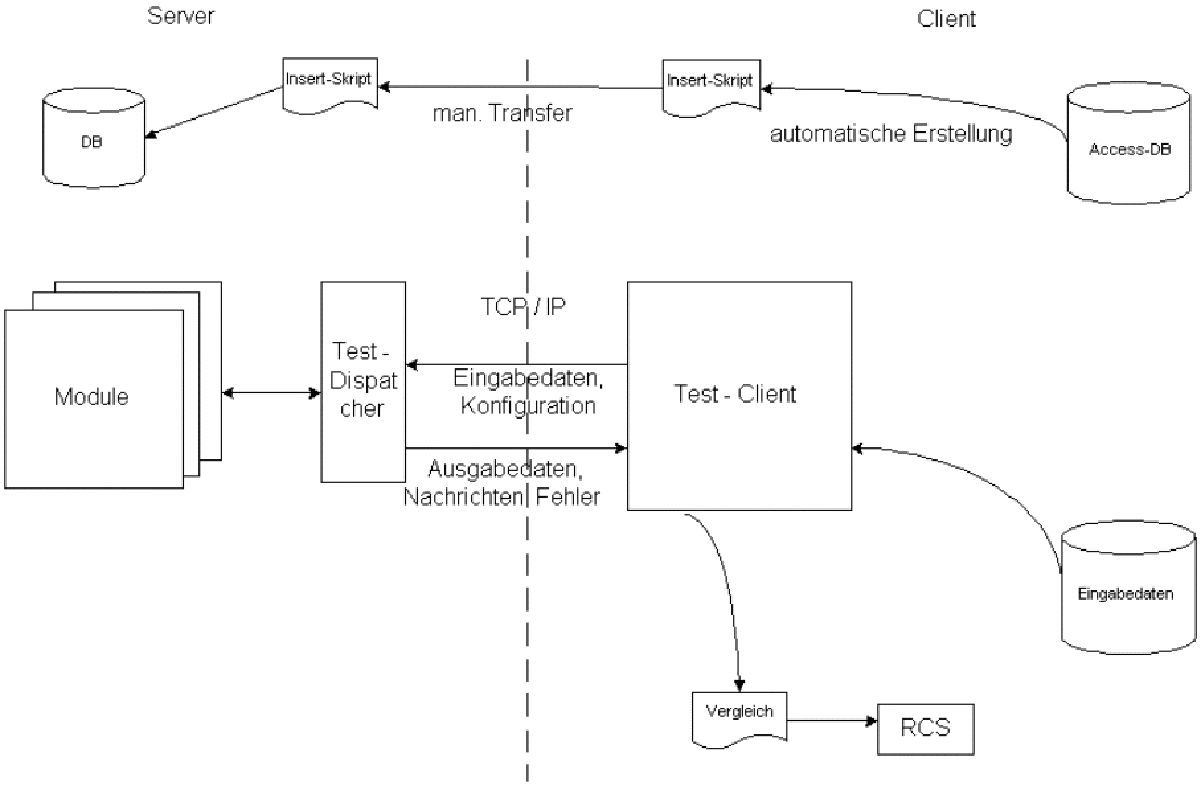

Abbildung 3.2: neuer Testablauf

Mein Beitrag zur Testverbesserung würde also hauptsächlich in der Beschleunigung des Ablaufs liegen – einerseits fallen einige langwierige Zwischenschritte weg, andererseits sind Fehler in den getesteten Modulen durch die Visualisierung (s. auch Abbildung 3.13: Ergebnisübersicht und Abbildung 3.14: Ausgabenvergleich) schneller zu erkennen und zu lokalisieren als in der bisherigen Ausgabedatei. Jan's Aufgabe lag indessen hauptsächlich in einer organisatorischen Verbesserung durch die zentrale Datenhaltung, wobei durch gewisse Skripte in der Access-DB natürlich auch einige Aufgaben bei der Datenpflege erleichtert werden können. Zum jetzigen Zeitpunkt stand noch nicht fest, ob der Test-Client innerhalb der Access-Datenbank mittels Visual Basic oder als eigene Anwendung in Java realisiert werden sollte. Meine Präferenz lag klar bei Java, u.a. weil hier mehr Möglichkeiten hinsichtlich der Dialoge bestanden, weil die Methode des Netzwerkzugriffes in Visual Basic unklar war, weil Java eine wesentlich 'sauberere' Software ermöglicht und weil ich mit Java schon viel Erfahrung hatte. Auf jeden Fall war aber abzusehen, dass eine genaue Abstimmung mit Jan erforderlich sein würde.

Am Ende der Woche hatte ich noch etwas Zeit übrig, weil weitere Entscheidungen über das Vorgehen erst Anfang der folgenden Woche getroffen werden würden. In dieser Zeit überlegte ich mir schon einmal eine grobe Architektur für den Test-Client, also seine Aufteilung in Komponenten, und sprach diese mit Kai ab. Die Architektur hatte ohne größere Änderungen Bestand, daher zeige ich sie schon hier in Abbildung 3.3.

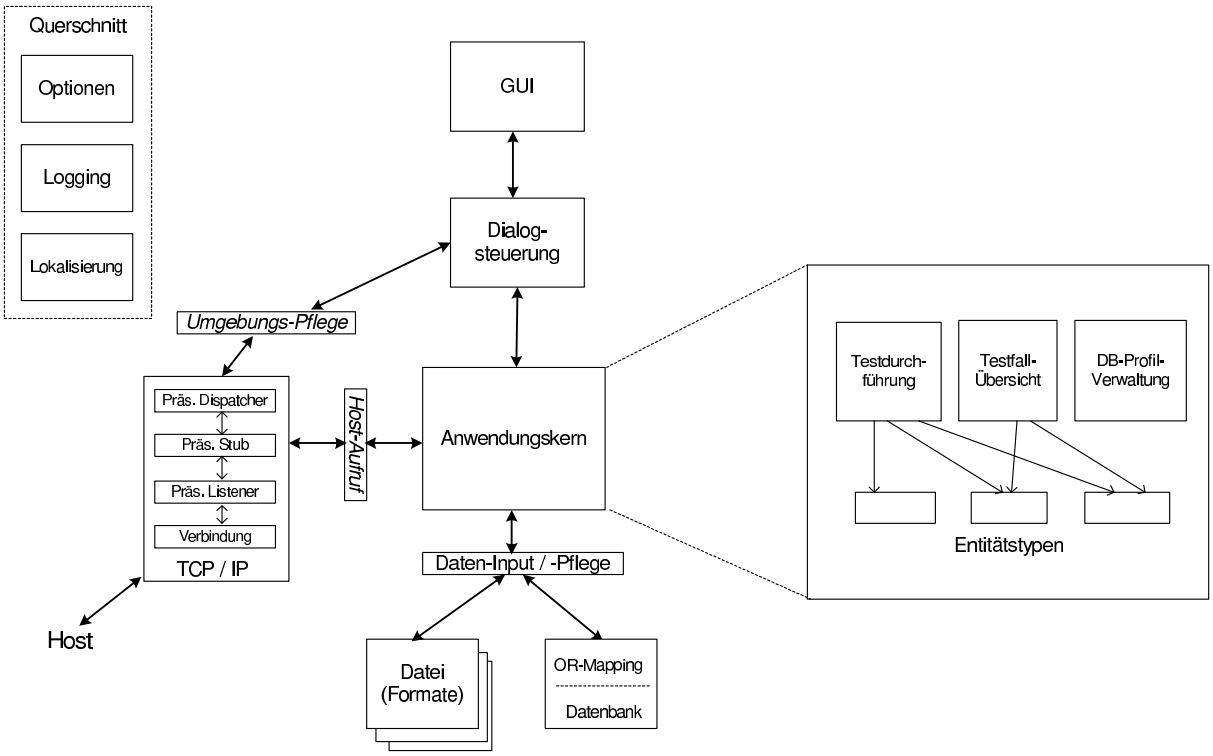

Abbildung 3.3: Architektur des Test-Clients

Wie es Quasar vorgibt, trennte ich die Dialogsteuerung und die GUI. Klar war auch die Existenz einer Komponente für die Netzwerk-Operationen und einer Komponente für die Datenbank-Anbindung. Hier schlägt Quasar die Einführung einer expliziten Zugriffsschicht vor, welche die technische Anbindung an die Datenbank kapselt. Ich entschied mich jedoch dagegen, zum Einen wegen des dann höheren Komplexitätsgrades (und Aufwandes, denn der Umfang der DB-Zugriffe ist beim Test-Client noch recht gering, so dass der feste Aufwand zur Erstellung der Zugriffsschicht die Einsparungen beim Zugriff selbst überwogen hätte), zum Anderen, weil weder bei Java mit JDBC noch bei Visual Basic ein Wechsel der Zugriffstechnik realistisch zu erwarten war. Um größtmögliche Austauschbarkeit und Einfachheit zu erreichen, sollten DB-Komponente und Datei-Komponenten eine gemeinsame Schnittstelle erhalten. Pro Dateiformat kann es eine eigene Komponente geben, wobei die richtige Auswahl und Instanziierung von einer Fabrik geregelt wird; realisiert habe ich während des Praktikums schließlich eine Komponente für ein bestimmtes Format – mehr dazu im Bericht der 10. Woche ab S. 26. Die Netzwerk-Komponente sollte eine eigene Schnittstelle zur Pflege von Umgebungen (diese enthalten die notwendigen Daten zur Verbindung mit dem Host, insbesondere IP-Adresse und Portnummer) erhalten, auf die direkt unter Umgehung des Anwendungskerns zugegriffen wird; dadurch wird dieser unabhängiger und einfacher. Intern hat die Netzwerkkomponente eine Schichtenarchitektur, bei der jede Schicht eine spezielle Präsentation der Daten für eine korrespondierende Schicht auf dem Host übernimmt: dort werden die Daten zunächst von einem *Listener* entgegengenommen, der z.B. die Benutzer-ID braucht, dann von einem *TCP-Stub*, der ein bestimmtes Modul aufruft – den Test-Dispatcher in diesem Fall.
Auf Seite der A-Architektur überlegte ich mir eine Aufteilung in zwei Anwendungsfälle: einerseits die Anzeige und Auswahl der vorhandenen Testfälle, andererseits die Durchführung und Auswertung von Tests. Später kam noch die Verwaltung von DB-Profilen hinzu, die Informationen über den Zugriff auf eine bestimmte Datenbank beinhalten (URL, Benutzer,

Kennwort usw.). Zuletzt sollte es noch ein paar Hilfskomponenten geben, und zwar zum Speichern von Einstellungen, zum Loggen von Fehlermeldungen und zur Lokalisierung des Test-Clients.
Ich war mir nicht ganz sicher, ob eine klare Komponentenstruktur in Visual Basic möglich sein würde, hoffte aber einerseits auf eine Realisierung in Java und andererseits im Falle von VB auf eine Strukturierung z.B. auf Modulbasis.

3. Woche (KW 31): Datenmodell; JEFF Views

Am Montag begleitete ich Kai erstmals in die sd&m-Hauptniederlassung in München, um an einem sog. internen CD-Meeting teilzunehmen. Dies ist ein regelmäßiges Treffen der Chefdesigner der einzelnen Teilprojekte im GO-Projekt, das der Koordination und Problemlösung dient.
Bei der Gelegenheit lernte ich dann Jan persönlich kennen, der den ersten Entwurf des Datenmodells für die Testverbesserung vorstellte. In diesem Entwurf war z.B. die rekursive Schachtelung von Datentypen noch nicht enthalten; im Folgenden beschreibe ich aber das endgültige Datenmodell, soweit es für den Test-Client von Belang ist. Abbildung 3.4 zeigt ein vereinfachtes ER-Modell, in dem nur die Schlüssel und für die Struktur wichtige Attribute dargestellt sind.

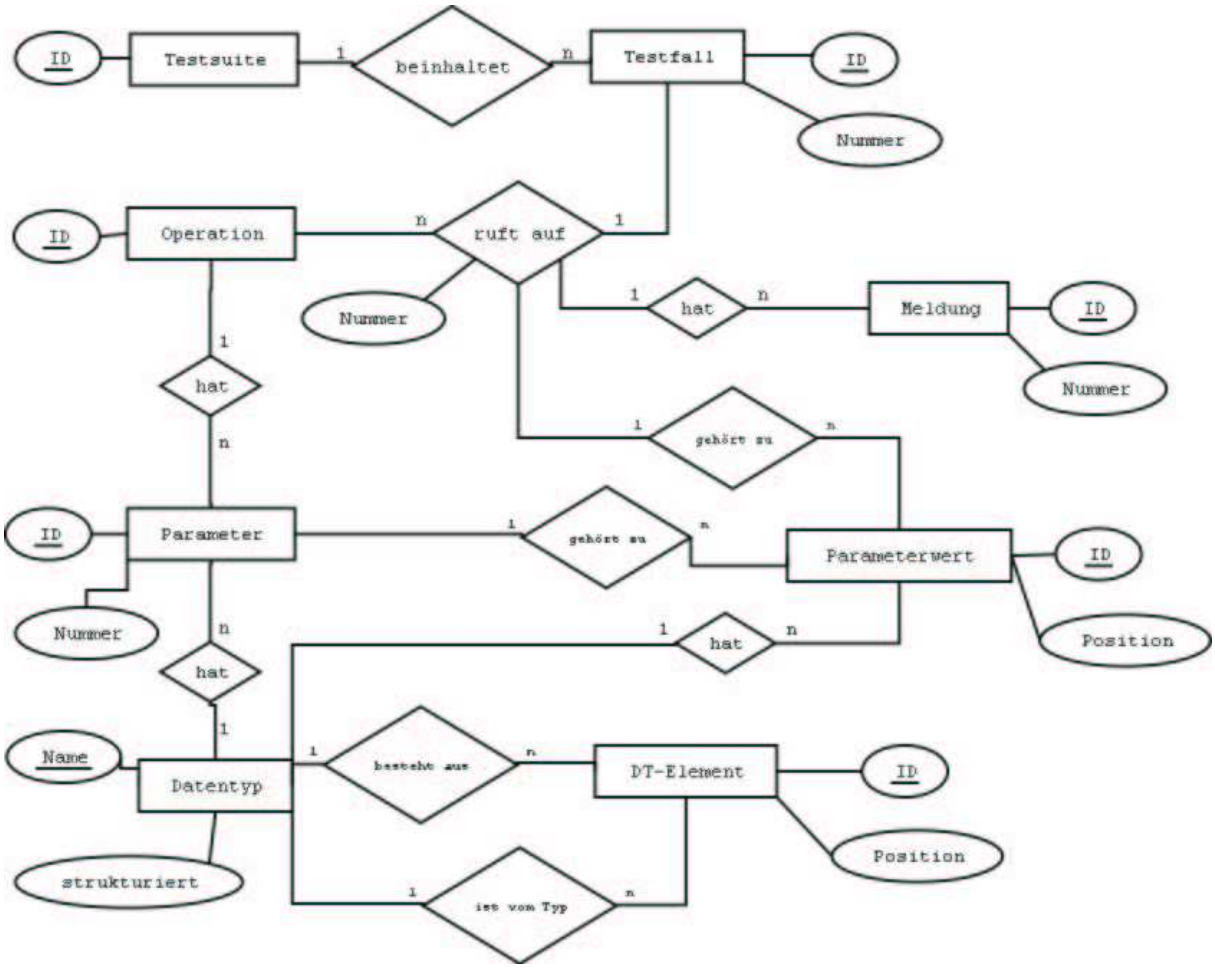

Abbildung 3.4: ER-Modell

In der Datenbank werden zunächst Informationen über die Operationen gespeichert, die getestet werden sollen. Diese Operationen haben jeweils eine Anzahl Parameter, von den wiederum jeder einen bestimmten Datentyp hat. Manche Datentypen sind strukturiert / zusammengesetzt, was bedeutet, dass sie aus einer Reihe von Elementen bestehen, die selbst wieder einen Datentyp haben – so werden Record-Strukturen nachgebildet. Ein Testfall besteht dann aus dem Aufruf einer Reihe von Operationen, wobei zum Einen die Eingabeparameter mit Werten belegt werden müssen, zum Anderen bestimmte Werte für die Ausgabeparameter erwartet werden. Diese Werte werden auf der Ebene der unstrukturierten Datentypen gespeichert. Beim Aufruf einer Operation können außerdem Meldungen generiert werden; jeder Operationsaufruf hat weiterhin einen Return-Code und einen Systemfehler-Code. Die Testfälle werden thematisch in Testsuiten gruppiert.

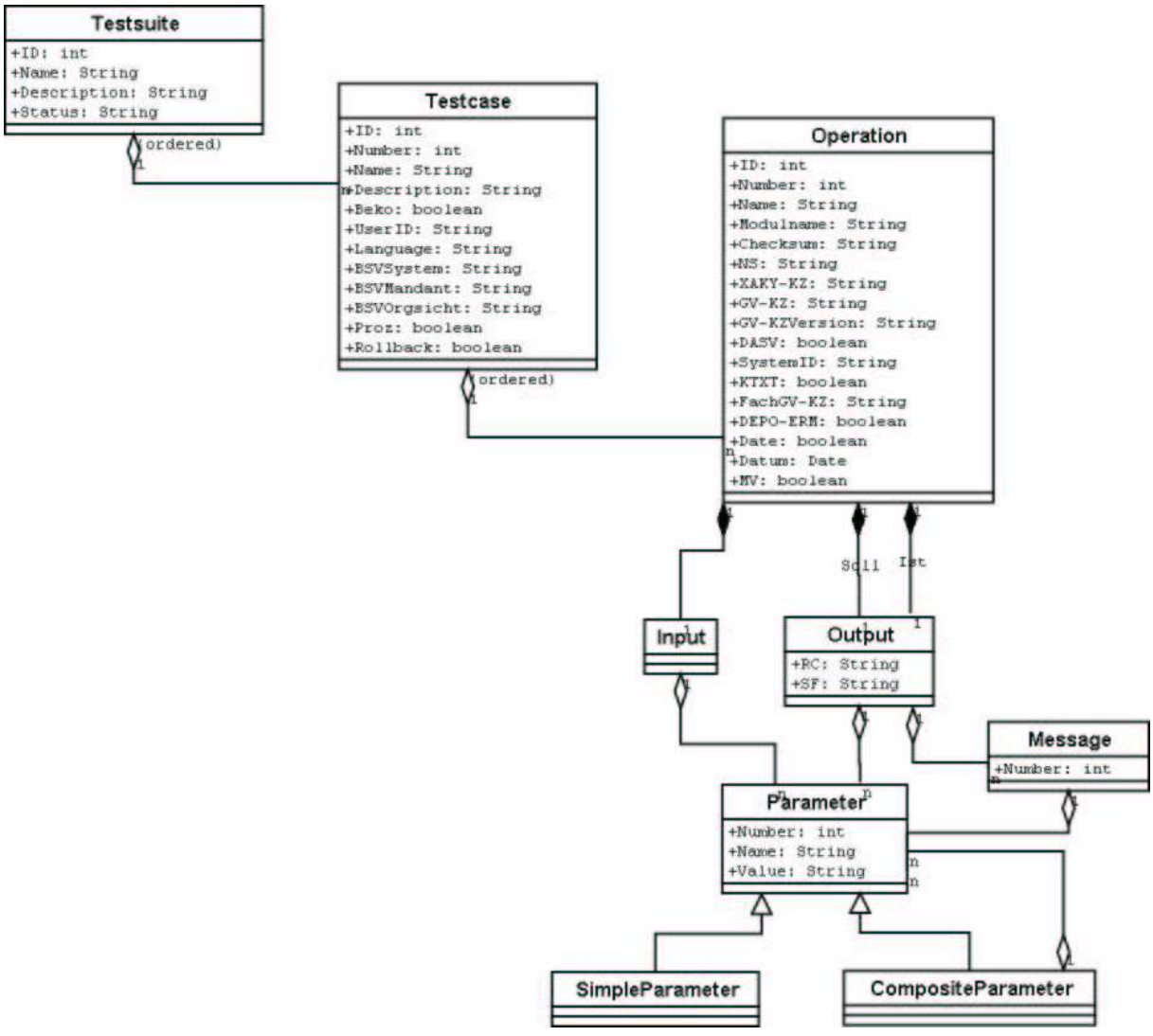

Abbildung 3.5: Entitäten

Mit den nun vorhandenen Informationen über das Datenmodell entwarf ich die Entitätsklassen des Test-Clients, wie sie in Abbildung 3.5 gezeigt werden. Operation und ihren Aufruf fasste ich in einer Klasse zusammen, denn die Metadaten über vorhandene Operationen interessieren den Client nur in Verbindung mit den Operationsaufrufen. Da der Test-Client auch die Istdaten speichern muss, um einen Vergleich anstellen zu können, führte ich weiterhin

Zwischenklassen für Eingabe und Ausgabe einer Operation ein. Die Parameterwerte speicherte ich zusammen mit den Daten des zugehörigen Parameters (die Datentyp-Informationen waren dagegen nach dem Einlesen aus der Datenbank unwichtig), wobei die rekursive Struktur durch ein Kompositum-Muster nachgebildet wurde. Hier sah ich auch gleich den Einsatz von Besucher-Mustern vor. Die Daten einer Meldung schließlich modellierte ich ebenfalls in Form von Parametern.

In der restlichen Woche las ich mich in das Konzept der *JEFF Views* ein. Dies ist ein bei sd&m entwickeltes Framework für die Erstellung von Benutzungsoberflächen in Java. Zwar stand noch gar nicht fest, ob der Test-Client tatsächlich in Java realisiert werden würde, aber für diesen Fall wollte ich untersuchen, ob das Framework für meine Zwecke geeignet wäre – eine Tatsache, die die Entscheidung für oder gegen Java übrigens auch beeinflussen konnte.
Das Hauptziel bei der Entwicklung der Views war es, den Software-Entwickler dazu zu bringen, eine saubere, mit den Prinzipien von Quasar konforme Architektur zu verwenden. Sie bringen es weiterhin mit sich, dass ein Wechsel der Dialoge von Swing zu HTML (über JSP mit einer zusätzlichen Library) sehr gut unterstützt wird. Auch der Aufruf von Funktionalität über das Netz wird von ihnen automatisch unterstützt, diese Möglichkeit habe ich allerdings für den Test-Client nicht gebraucht.
Eine sog. *View* ist eine - möglichst wiederverwendbare – GUI-Komponente. Sie passt üblicherweise zu einem Dialog oder zu einem Teil eines Dialoges, denn Views können weitere Views beinhalten. Die Daten des Dialoges hält die View in einem ihr zugeordneten, speziellen Objekt.
Zur Anzeige verwendet die View einen sog. *Visualizer*, der dann je nach Ausgabekanal – Swing oder HTML – ausgetauscht werden kann. Dieser Visualizer erstellt die visuelle Repräsentation der View auf dem Bildschirm des Benutzers und füllt sie mit Daten. Er ist außerdem für eine einfache Überprüfung der Korrektheit von eingegebenen Daten zuständig (z.B. Formatprüfungen). Abbildung 3.6 verdeutlicht dieses Zusammenspiel.

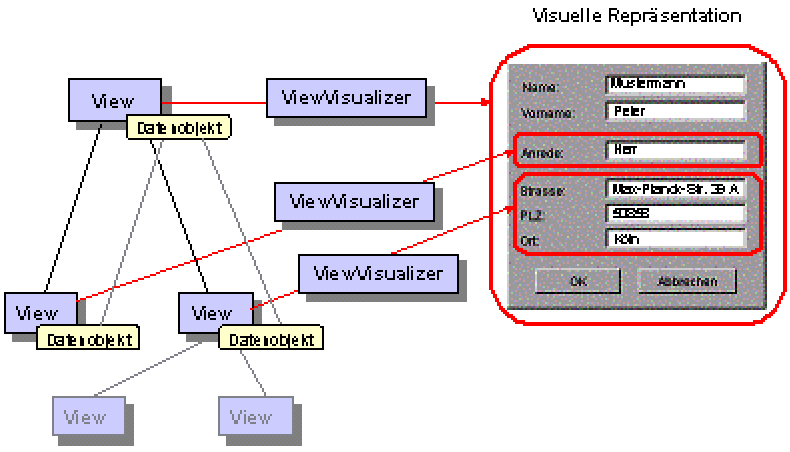

Abbildung 3.6: View-Struktur

Die View definiert *Kommandos*, mit denen der Benutzer Aktionen auslösen kann. Über sog. *Kommando-Aktivatoren* werden diese Kommandos dann GUI-Elementen im Dialog (z.B. Schaltflächen) zugeordnet, die abhängig vom Zustand des Datenobjektes automatisch aktiviert oder deaktiviert werden können. Diese Struktur wird von Abbildung 3.7 verdeutlicht.

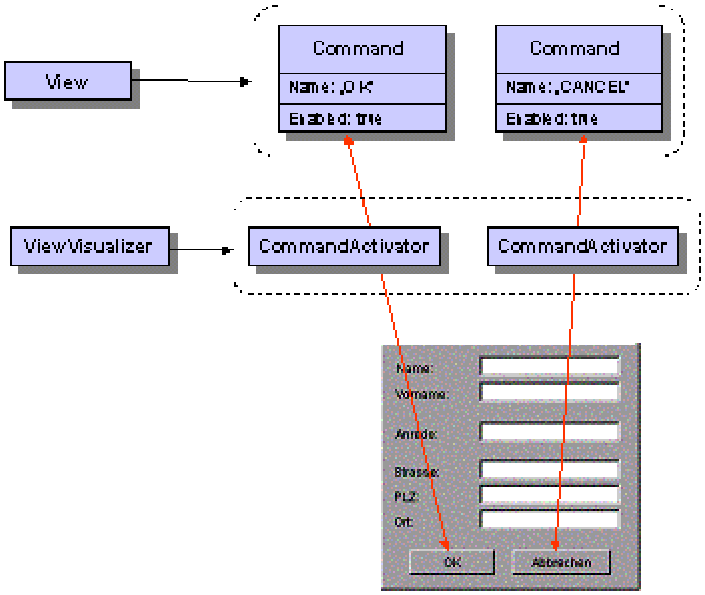

Abbildung 3.7: Kommandos

Ich kam zu dem Schluss, dass die JEFF Views für die Zwecke des Test-Clients sehr gut geeignet seien, und sie wurden bei der Realisierung dann auch eingesetzt.

4. Woche (KW 32): Komponenten, Prototyp

Jetzt fing ich an, die Schnittstellen der vorgesehenen Komponenten detaillierter zu entwerfen, um das Zusammenspiel und den richtigen Zuschnitt überprüfen zu können. Ich legte dafür ein Dokument an, in dem ich die Architektur des Test-Clients beschrieb. Das Dokument war zunächst hauptsächlich für mich selbst gedacht; ich überlegte mir die Struktur daher selbst und kam schließlich zu folgendem Inhaltsverzeichnis:

1. Einleitung
2. TI-Architektur
3. T-Architektur
4. A-Architektur
5. Anhang

Der größte Teil lag in der T-Architektur (siehe auch Abschnitt 0), wo ich die Schnittstelle der einzelnen Komponenten in Form von Prozedurdefinitionen in Pseudo-Code festlegte. Außerdem fügte ich zu jeder Komponente einen Abschnitt hinzu, in dem ich ihre Funktion beschrieb, einen Abschnitt zur Initialisierung der Komponente und schließlich einen Abschnitt, in dem ich mir erste Gedanken zur internen Realisierung machte, so dass ich

feststellen konnte, ob eine solche Komponente eigenständig existieren kann oder ob sie (andererseits) zu kompliziert wäre, was für eine Aufteilung in mehrere Komponenten spräche. Auch konnte ich so die allgemeine Machbarkeit abschätzen. Glücklicherweise musste ich jedoch die vorgesehene Komponentenstruktur nicht ändern.
Im Abschnitt über die A-Architektur überlegte ich mir auch die möglichen Anwendungsfälle. Ich fertigte aber keine detaillierte Beschreibung im Sinne eines Fachkonzeptes an (wie ich es aus den Studienprojekten gewohnt war), sondern konzentrierte mich mehr auf die notwendigen Operationen auf Entwurfsebene, denn es ging mir hauptsächlich darum, festzustellen, ob die Architektur des Test-Clients den Erfordernissen der Anwendung gewachsen sein würde. Beim Entwurf der Schnittstellen des Anwendungskerns achtete ich außerdem auf eine allgemeinere Verwendbarkeit – also darauf, dass er auch durch andere Dialoge und eventuell über ein Netzwerk aufgerufen werden konnte.
Die A-Architektur des Test-Clients zeigt Abbildung 3.8. Sie besteht aus zwei Komponenten, einer kleinen zur Verwaltung der DB-Profile und einer großen für den eigentlichen Test. Der Anwendungsfall zur Testdurchführung hat außerdem zwei kleine Hilfskomponenten, eine zum Vergleich von Soll- mit Istdaten und eine zum Formatieren von Testprotokollen. Die Testdatenverwaltung hat eine Lese- und eine Schreibschnittstelle, wobei der Testfallübersichts-AF nur die Leseschnittstelle nutzt.

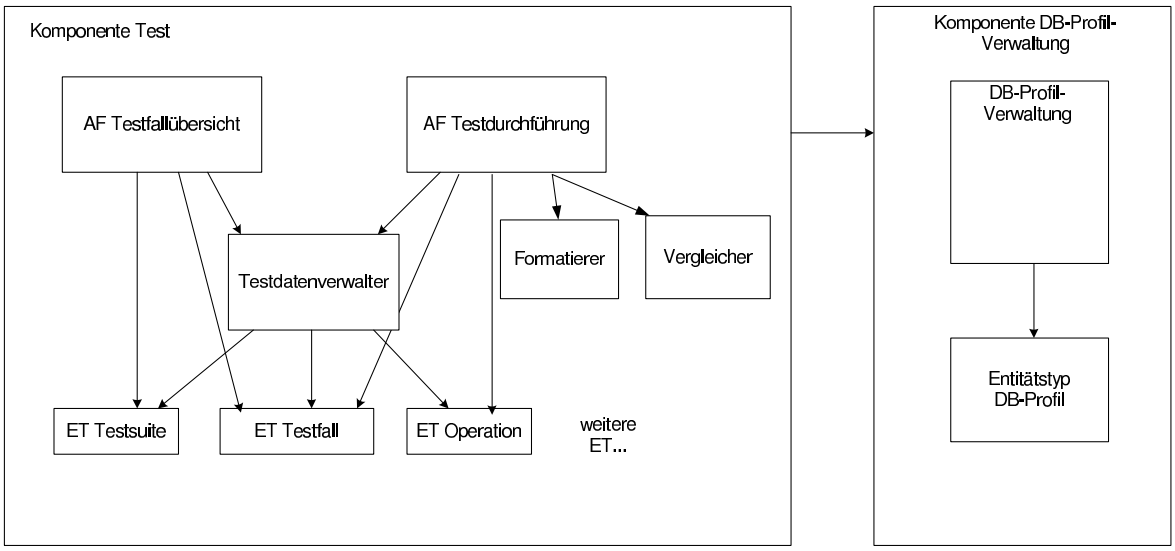

Abbildung 3.8: A-Architektur

Die restliche Woche verbrachte ich mit der Erstellung eines GUI-Prototyps der Anwendung, um eine gute Kontrolle zu haben, ob ich die Anforderungen an den Test-Client richtig verstanden hatte. Dazu verwendete ich den GUI-Builder der IDE von Borland, JBuilder. Hier sprach ich mich wieder intensiver mit Kai ab, der eine Reihe von Verbesserungsvorschlägen einbrachte. Zum Beispiel hatte ich zuerst eine Auswahl der Datenquelle – also Datenbank oder Datei – im ersten Dialogfenster vorgesehen. Die Datenquelle wird aber recht selten geändert, so dass wir diese Auswahl ins Menü verlegten und festlegten, dass der Test-Client die zuletzt verwendete Quelle beim erneuten Start automatisch wieder öffnet. Kai schlug auch vor, Soll- und Istwerte in einem Dialogfenster in einer nicht-proportionalen Schriftart untereinander darzustellen (s. Abbildung 3.14), was vor allem bei sehr langen Parameterwerten von Vorteil ist. Screenshots der schließlich entstandenen Dialoge sind in den späteren Wochenberichten enthalten.

5. Woche (KW 33): Beginn der Realisierung

Am Anfang der Woche war ich wieder in München zum internen CD-Meeting. Jan stellte das geänderte Datenmodell und ich den Prototypen vor. Es kamen noch eine Reihe von Verbesserungsvorschläge, insbesondere zur Anzeige des Soll-/Ist-Vergleiches. Außerdem wurde beschlossen, dass der Test-Client als Java-Applikation und nicht in Form von Visual Basic-Skripten innerhalb der Access-Datenbank realisiert werden sollte. Dies kam mir auch deshalb gelegen, weil ich schon am Ende der vorherigen Woche die ersten Schritte zur Realisierung der Querschnittskomponenten unternommen hatte ;-).

Mit den Querschnittskomponenten (Optionen, Logging, Lokalisierung) setzte ich meine Arbeit dann auch fort. Hier am Anfang überlegte ich mir auch die allgemeine Zugriffsstruktur, die ich im Regelfall für die Komponenten einsetzen wollte. Ich entschied mich dafür, die Schnittstellen und Ausnahmen in ein Java-Paket zu legen, die Implementierung der Komponente in ein Unterpaket. Außerdem legte ich in das Schnittstellen-Paket eine Klasse mit statischen Methoden: einerseits zur Definition der implementierenden Klasse durch den Glue-Code der Anwendung, andererseits zum Zugriff auf die Komponente durch die Anwendung, teilweise auch als 'Abkürzung' Stellvertreter-Methoden aus der Komponentenschnittstelle. Diese Struktur garantiert eine einfache Austauschbarkeit der Implementierung und Wiederverwendbarkeit der Komponenten. Sie ist beispielhaft anhand der relevanten Klassen der Netzwerk-Komponente in Abbildung 3.9 gezeigt. Zu diesem Zeitpunkt definierte ich außerdem Oberklassen für die Ausnahmen im Test-Client; diese stellen hauptsächlich die Möglichkeit zur Lokalisierung der Fehlermeldung zur Verfügung.

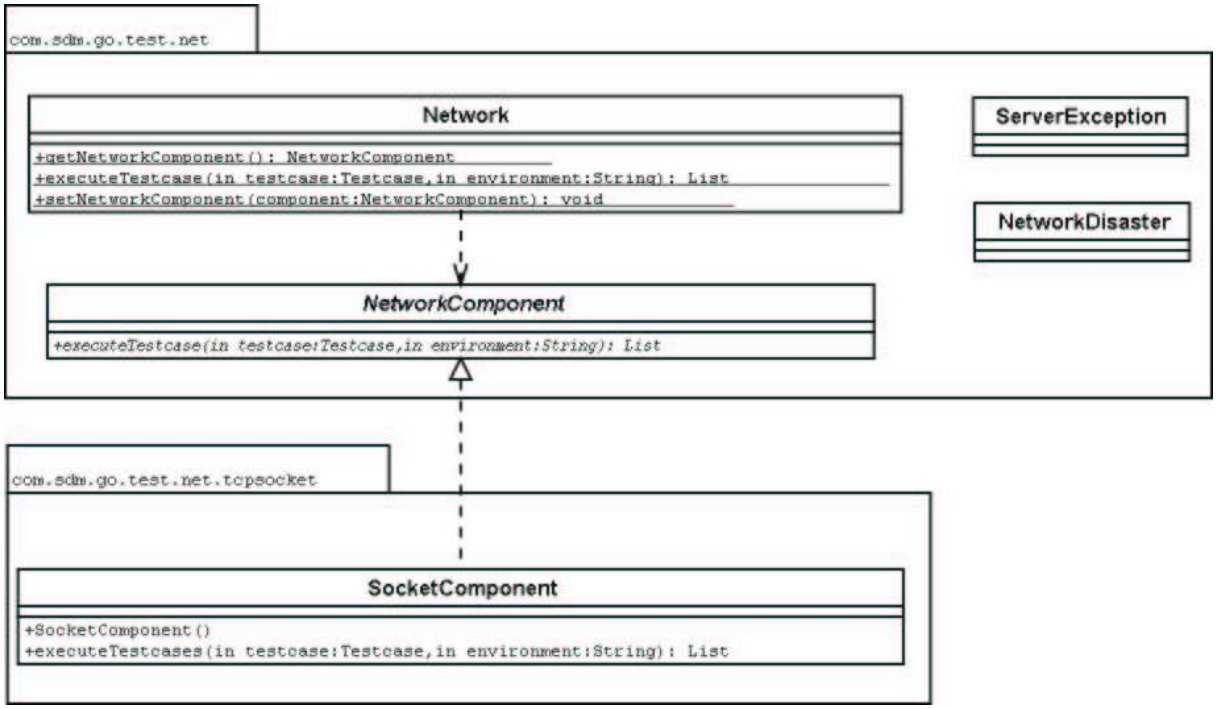

Abbildung 3.9: Komponentenstruktur

Für die Implementierung der Querschnittskomponenten setzte ich jeweils bestehende Java-Funktionalität ein, für die ich dann nur noch Adapter schreiben musste. Für das Logging ist

dies das Java-Paket java.util.logging, das eine äußerst große Flexibilität bietet. Die Optionen speicherte ich mit Hilfe des Java-Paketes java.util.prefs, das ebenfalls genau auf diese Aufgabe ausgerichtet ist. Für die Lokalisierung setzte ich java.util.ResourceBundle ein. Das bedeutet, dass die lokalisierten Strings in einfachen Textdateien der Form 'Name=Wert' gespeichert werden – für eine andere Sprache muss man nur eine weitere Datei mit der richtigen Endung anlegen. Teilweise war die Funktionalität erst im neuesten Java-Release eingeführt worden, so dass eine kurze Einarbeitung notwendig war. Probleme traten aber nicht auf. Ich schrieb dann noch für jede Komponente einen Testtreiber zum automatisierten Test, wobei ich mir die Anweisungsüberdeckung zum Ziel setzte.

Nach den Querschnittskomponenten definierte ich die Entitätsklassen, was hauptsächlich eine Fleißarbeit war. Damit hing auch der Vergleicher eng zusammen, so dass ich seine Realisierung direkt danach machte. Für das Durchlaufen der Datenstrukturen einer Ausgabe (insbesondere der zusammengesetzten Parameter) bereitete ich die Entitäten für die Anwendung einer abgewandelten Form des Besucher-Musters vor – statt einer Methode `visit(Visitor)` eine Methode `compare(Comparator, Comparable)`, so dass das zweite Objekt im Vergleich mit übergeben werden kann. Die Ergebnisse eines Vergleiches legte ich in einen Cache (gebildet durch eine HashMap), damit er nicht bei jedem Abruf wiederholt werden muss.

Danach ging ich an die Realisierung der unteren Schichten in der Netzwerkkomponente, weil mir das Format der auf Hostseite entsprechenden Schichten (IBM-Listener und TCP-Stub) noch gut in Erinnerung war und mich der saubere Entwurf einer Schichtenarchitektur interessierte. Ich definierte also eine gemeinsame Schnittstelle für Schichten, so dass ein möglichst flexibles 'Zusammenstecken' einer Gesamt-Komponente aus den Schichten möglich wurde. Um dabei ein zeitraubendes Kopieren der Strings, die an den Host geschickt oder von ihm empfangen werden, zu vermeiden, setzte ich die Klasse StringBuffer als Parametertyp ein, die sich dynamisch erweitern oder verkürzen lässt. Die unterste Schicht wird durch eine Adapter-Klasse gebildet, die den Socket-Zugriff kapselt. Abbildung 3.10 zeigt einen größeren Überblick über die Klassenstruktur.

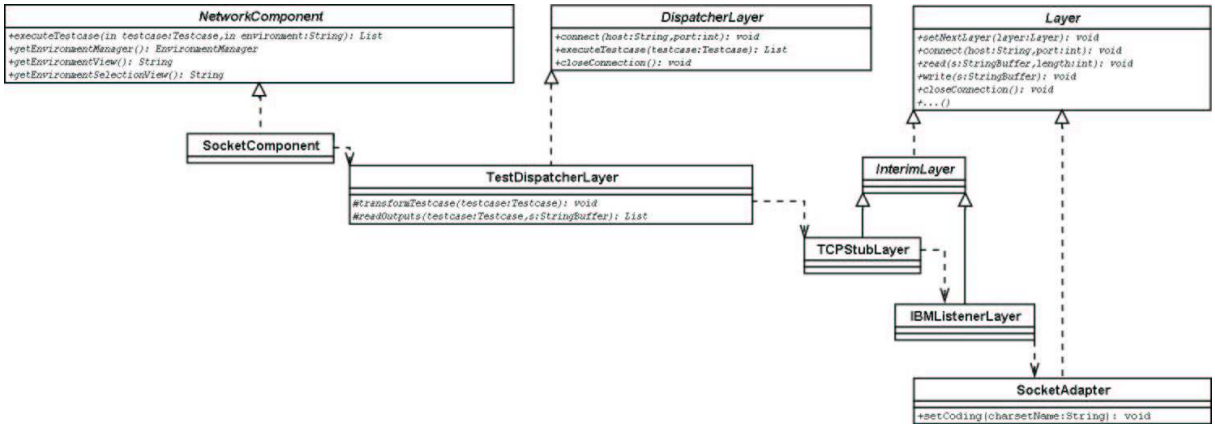

Abbildung 3.10: Klassen der Netzwerk-Komponente

Um diese Klassen testen zu können, verwendete ich zunächst eine einfache Java-Klasse, die auf einem Socket als Server wartet, Verbindungen annimmt und erhaltene Daten unverändert zurückschreibt. Diese Klasse erweiterte ich im Laufe der Entwicklung des Test-Clients bis hin

zu einem einfachen Stub, der den Test-Dispatcher auf Host-Seite simulieren kann. Dadurch wurde ein Test des Test-Clients auch ohne Netzwerk und Host möglich.
Beim Test der Klassen traten nur einfache Formatierungsfehler auf, die schnell behoben waren. Ein anderes Problem war die Festlegung der richtigen Codierung des Zeichenstroms (für den Host ist ein EBCDIC-Format notwendig). Beim Anlegen eines Zeichenstroms kann man mehr Codierungen angeben, als die Java-Routine, welche die Korrektheit einer Codierung überprüft, zulässt – das scheint ein Bug in Java zu sein. Daher musste ich die ursprüngliche Idee, die Codierung vor der Herstellung der Verbindung festzulegen (das hätte die Fehlerbehandlung vereinfacht) aufgeben.

6. Woche (KW 34): Erste Dialoge

Nachdem die Netzwerk-Tests keine Fehler mehr aufzeigten, begann ich damit, mich mit der konkreten Anwendung des Frameworks der JEFF Views zu beschäftigen. Ich wählte als einfachen und typischen Dialog denjenigen zum Verwalten der verschiedenen Umgebungen (Abbildung 3.11). Die ComboBox ist editierbar, so dass dadurch neue Umgebungen angelegt werden können.

Abbildung 3.11: Umgebungspflege

Die Programmierung von Views teilt sich üblicherweise in drei Klassen auf: die View selber, die die Dialogsteuerung übernimmt und unabhängig von Swing ist; ihr Visualizer, der die Felder im Dialog füllt und ausliest und die Eingaben auf Typkorrektheit prüft; und ein Panel, das die Anzeige enthält. Letzteres stellte kein Problem dar, ich musste nur die richtigen LayoutManager richtig ineinander schachteln. Mit dem Framework selbst hatte ich aber zu Anfang einige Schwierigkeiten. So war es mir unklar, wo die Speicherung der Daten erfolgen konnte. Ich hatte dafür zuerst ein eigenes Kommando definiert; aber der Dialog (ein vom Framework bereitgestellter ViewContainer) führte beim Schließen über die Windows-Fensterfunktionen offensichtlich ein anderes Kommando aus – er zeigt vorher automatisch eine

Nachfrage an, ob die Änderungen gespeichert werden sollen. Ich versuchte, meinen Code an dieses Kommando zu hängen, was aber auch nicht das gewünschte Resultat brachte. Schließlich erkannte ich, dass ich ihn in die Methode `close` der View stecken musste, die innerhalb ihrer Argumente übergeben bekommt, ob der Dialog bestätigt oder abgebrochen wurde. Die Argumente von Methoden innerhalb des Frameworks werden stets in einer speziellen Form einer HashMap übergeben – das macht den Aufruf sehr flexibel, behindert aber eben teilweise das Verständnis; ich hatte diese Methode nur als Nachfragemethode, ob die View geschlossen werden darf, verstanden.
Ein weiteres Problem war die Aktualisierung der Daten in der ComboBox, z.B. nach dem Löschen einer Umgebung. Die Befüllung der ComboBox geschieht durch das Framework in einem CommandActivator, weil die restlichen Daten im Dialog nach Auswahl einer Umgebung geändert werden müssen. Das Framework hat aber keine Methode, um eine Aktualisierung der Daten in der ComboBox anzustoßen. Nach längerem Suchen im Quellcode des Frameworks rief ich von Hand zwei Methoden des CommandActivators auf. Dies führte zwar zum gewünschten Ergebnis, hatte aber teilweise Instabilitäten zur Folge. Diese konnte ich erst einige Wochen später beheben, indem ich meinen eigenen CommandActivator schrieb.

Als ich den Dialog fertig hatte, implementierte ich auch gleich die Verwaltung der Umgebungen auf der Basis von Optionen. Dies bereitete überhaupt keine Probleme. Ich speichere die Namen aller vorhandenen Umgebungen in einer Option, konkateniert durch ';'. Mit einem StringTokenizer können sie beim Auslesen einfach wieder getrennt werden. Für die einzelnen Daten einer Umgebung verwende ich dann jeweils eigene Optionen, durch den Namen der Umgebung eindeutig bestimmt (z.B. "LTU.port").

Zur Einübung dessen, was ich nun über das Framework gelernt hatte, realisierte ich gleich darauf den sehr ähnlichen Dialog zur Pflege von Datenbank-Profilen (Treiber, URL, Name, Kennwort) und die DB-Profil-Verwaltung. Hierfür konnte ich große Teile des Codes wiederverwenden.

7. Woche (KW 35): Restliche Dialoge

Ich hatte jetzt ein Gespür dafür entwickelt, wie das Framework funktioniert und begann mit der Entwicklung der größeren Dialoge. Dies sind ein Dialog zur Auswahl der Testfälle (Abbildung 3.12), ein Dialog zur Übersicht über die Testergebnisse (Abbildung 3.13) und schließlich ein Dialog zum detaillierten Vergleich zwischen Soll- und Istausgaben (Abbildung 3.14).

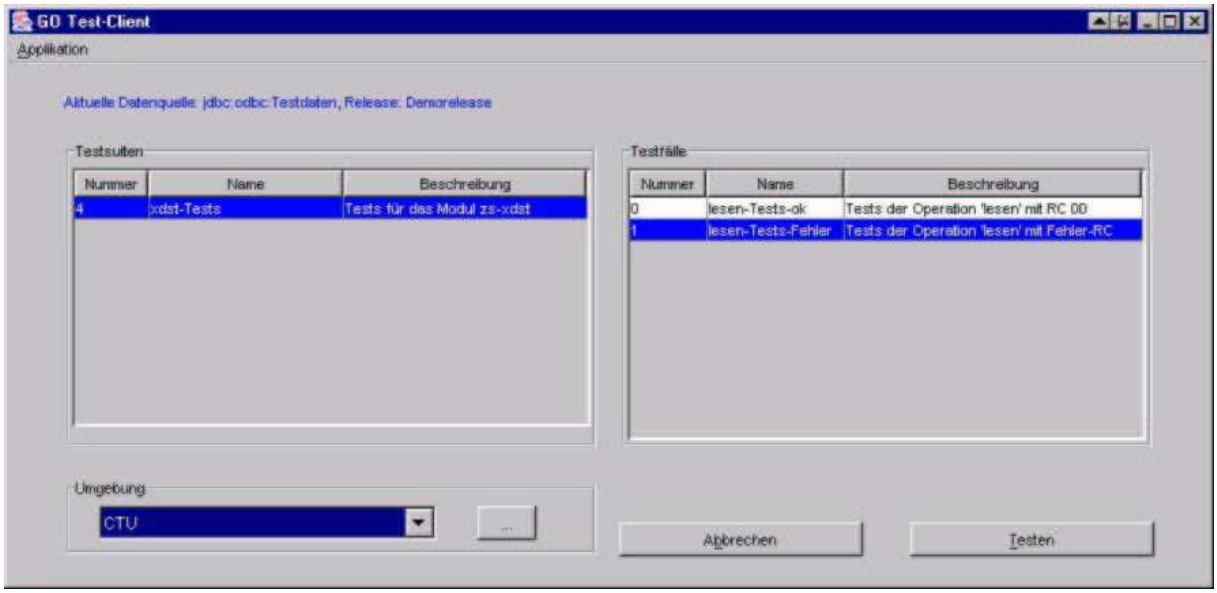

Abbildung 3.12: Testfallselektion

Die Testfallselektion war relativ einfach. Ich musste für das Layout leider ein GridBagLayout benutzen, um die richtige Platzierung der Elemente zu erreichen, kam aber gut damit zurecht. Weil auf die Selektion von Zeilen in der Tabelle reagiert werden muss, brauchte ich hier einen eigenen CommandActivator, da ein solcher im Framework nicht vorhanden war. Ich benutzte zur Vorlage den aus dem Framework, der auf Listen reagiert. Hierbei habe ich aber vermutlich etwas falsch gemacht, denn das Framework reagierte mit sporadischem Fehlverhalten bezüglich der Aktivierung der Schaltflächen. Ich konnte die Fehler größtenteils umgehen, indem ich die Fokus-bezogene Aktivierung aller Kommandos abschaltete. Erst einige Wochen später schrieb ich den CommandActivator auf tieferer Ebene neu, wodurch die Probleme ganz verschwanden.

Die Anzeige der Resultatsübersicht stellte hinsichtlich der Tabellen eine größere Herausforderung dar. Um die Textfarbe der einzelnen Zeilen zu ändern, brauchte ich ebenso einen eigenen TableCellRenderer wie für die Anzeige der Icons in den ersten Spalten. Das größte Problem war jedoch, eine Anzeige der gesamten Tabelle bei beliebiger Breite zu garantieren. Swing schneidet trotz Einbettung in eine ScrollPane die Spalten so ab, dass die gesamte Tabelle gezeigt wird, wenn man die Spaltenbreite veränderbar macht. Andernfalls kann der Benutzer aber ebenfalls nicht die Spaltenbreite ändern, und außerdem müsste man sie schon beim Anlegen der Tabelle groß genug machen – wobei man die später eingefügten Daten noch gar nicht kennt. Zur Lösung dieses Problems musste ich meine eigene Spezialisierung der JTable schreiben, die auf Änderungen des Models reagiert und außerdem in der `doLayout()`-Methode den autoResize-Modus je nach Erfordernis an- oder abschaltet, so dass ein Scrollbalken

angezeigt wird, sobald es notwendig ist. Für die Teilviews, die die Operationen und die Meldungen anzeigen, konnte ich viel Code wiederverwenden. Auch die Ergebnisübersicht benötigte ein GridBagLayout.

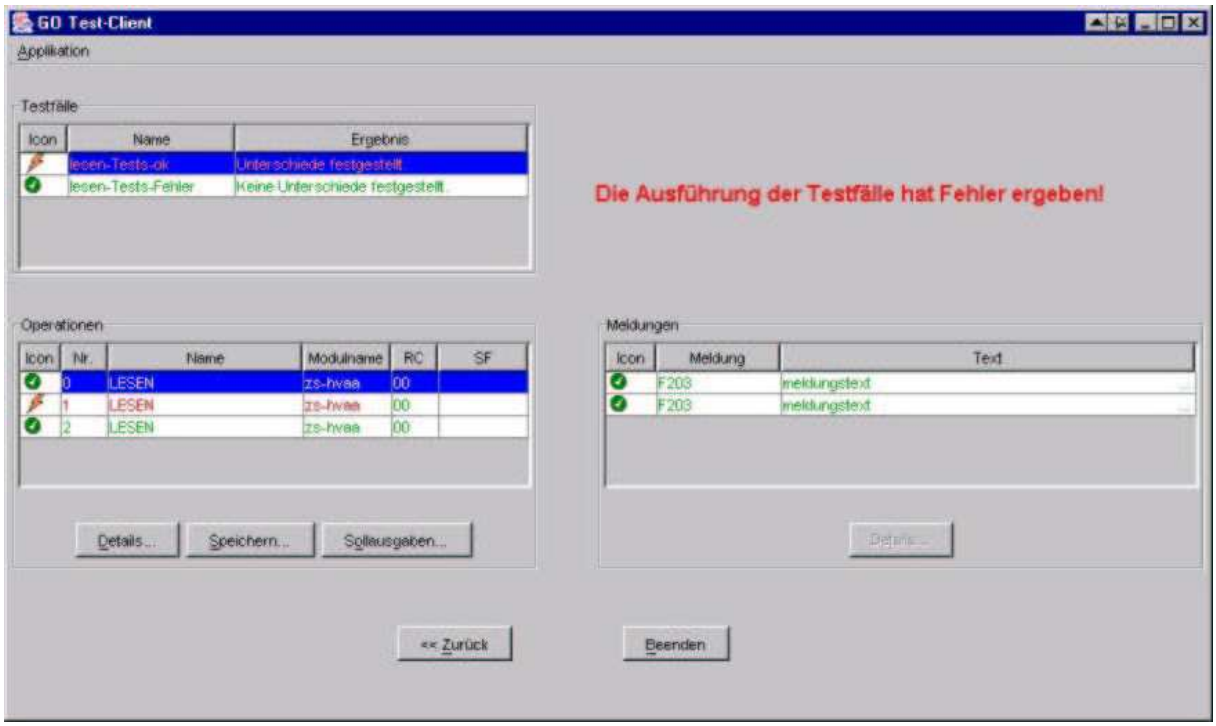

Abbildung 3.13: Ergebnisübersicht

Der Dialog zum direkten Vergleich von Soll- und Istwerten wird sowohl zur Anzeige der Details einer Meldung als auch der Details (hauptsächlich der Parameter) einer Operation benutzt. Hier war eine ganze Reihe von Problemen zu lösen. Für den Vergleich muss innerhalb einer Tabellenzelle eine weitere Tabelle angezeigt werden. Dies ging recht einfach mittels eines weiteren TableCellRenderers; allerdings gelang es mir nicht, die Zeilenhöhe innerhalb der großen Tabelle unterschiedlich zu setzen, so dass jetzt auch bei Parametern ohne Unterschiede die doppelte Zeilenhöhe verwendet wird. In diesem Dialog kann sich die Schriftfarbe auch innerhalb einer Zelle ändern (in der Abbildung nicht zu sehen); dies erforderte einen weiteren TableCellRenderer, der keinen JLabel, sondern ein JTextField benutzt, das etwas komplizierter in der Verwendung ist. Beim Doppelklick auf eine Zeile mit einem zusammengesetzten Parameter wird eine neue Instanz des Dialogs geöffnet, der die Unterelemente des Parameters anzeigt, es war also noch ein CommandActivator notwendig. Schließlich ist die Initialisierung des Dialogs nicht völlig trivial, da entschieden werden muss, ob alle Parameter oder nur die unterschiedlichen Parameter angezeigt werden sollen.
Ein weiteres Problem ergab sich daraus, dass das Framework am Ende der Ausführung eines Kommandos den Fokus auf die ursprüngliche View zurücksetzt. Das bedeutete, dass die Ergebnisübersicht regelmäßig das kleinere, neu geöffnete Fenster des Ausgabenvergleichs verdeckte. Ich brauchte erst einige Zeit, um die Ursache zu erkennen, und musste mir schließlich mit einem Trick behelfen: indem ich einen Timer starte, der dem neuen Fenster nach kurzer Zeit (einigen Zehntelsekunden) den Fokus zurück gibt.

Abbildung 3.14: Ausgabenvergleich

8. Woche (KW 36): Anwendungskern, Präsentationsschicht für den Dispatcher

Als ich nun die gesamten einzelnen Dialoge erstellt hatte, fing ich an, eine Rahmenapplikation – sozusagen *den* Test-Client – zu bauen. Ich wollte auch hier möglichst eine Trennung zwischen den Swing-spezifischen Teilen der Anwendung und den davon unabhängigen Teilen erreichen. Somit definierte ich zwei Klassen: einen ViewContainer, der die Haupt-Views (Testfallselektion und Resultatsübersicht) anzeigt und das Menü verwaltet, und einen CommandController, der die Haupt-Dialogsteuerung übernimmt.
Den ViewContainer konnte ich von einer vorhanden Klasse des Frameworks ableiten, die schon eine Unterstützung für die wechselnde Anzeige mehrerer Views enthält. Ich musste hier also nur zwei kleinere Aufgaben bearbeiten: zum Einen die Definition von Kommandos für die Menüeinträge, die ihre Ausführung aber nur an Kommandos des CommandControllers delegieren, zum Anderen die Definition des Menüs. Das Menü muss für das Framework schon von der Hauptinitialisierungsklasse vorbereitet sein und man kann es dann über eine spezielle Methode des ViewContainers erweitern. Leider ist hier theoretisch auch nur eine Erweiterung am Ende vorgesehen, ich wollte aber meine Menüeinträge vor dem Ende (dem 'Beenden'-Eintrag) einfügen. Das Problem konnte ich umgehen, indem ich die bisherigen Menüeinträge speichere, sie entferne, meine Menüeinträge hinzufüge und dann die alten wieder hinzufüge. In der 11. Woche fügte ich noch Menüeinträge für den Schnellzugriff auf die zuletzt verwendeten Datenquellen hinzu (die ich als Optionen speichere).

Auch für den CommandController war eine abstrakte Basisklasse im Framework vorhanden, die mir viel Arbeit abnahm. Daher konnte ich mich ganz auf die Programmierung des Steuercodes konzentrieren. Hier war hauptsächlich darauf zu achten, stets alle Ausnahmen abzufangen und dem Benutzer aussagekräftige Fehlermeldungen zu geben. Während der Ausführung von Testfällen wollte ich einen Fortschrittsbalken anzeigen und entdeckte dafür die Klasse ProgressMonitor von Java. Hier musste ich nur ein wenig tricksen, um eine Anzeige des Dialogs zu erzwingen (normalerweise entscheidet der Dialog nach einer gewissen Zeit, ob genügend Zeit vergehen wird, damit sich seine Anzeige lohnt; diese Zeit setzte ich auf 0 und legte schon vorher einen minimalen Fortschritt fest).
Zuerst wusste ich nicht, wie ich Kommandos von untergeordneten Views – insbesondere das Test-Kommando – an den übergeordneten CommandController übergeben sollte, den die View nicht kennen darf. Ich fand die Lösung dann ein paar Tage später in einem zu dem Zeitpunkt herausgegebenen Tutorial zu den JEFF Views. Sie besteht darin, dem Kommando der untergeordneten View einen Listener hinzuzufügen, dessen Methode nach Ausführung des Kommandos aufgerufen wird. Schließlich programmierte ich den Glue-Code der Anwendung, der die Komponenten initialisiert, in der Hauptklasse mit der main-Methode.

Daraufhin beschloss ich, rasch die (restliche) Funktionalität des Anwendungskerns zu realisieren. Diese besteht eigentlich hauptsächlich aus der richtigen Delegation der Aufrufe an andere Komponenten (Datenquellen und Netzwerk), so dass ich schnell damit fertig wurde. Nur die Kontrolle der Testausführung war etwas interessanter, weil hier richtig auf die unterschiedlichen Ausnahmen reagiert werden muss – bei manchen, die sich wahrscheinlich auf falsche Eingabedaten zurückführen lassen, kann die Ausführung der restlichen Testfälle fortgesetzt werden; bei anderen, wie z.B. einem Fehlschlagen der Verbindung zum Server, sollte die komplette Verarbeitung abgebrochen werden. Den Ablauf eines Testvorgangs auf Komponentenebene zeigt Abbildung 3.15.

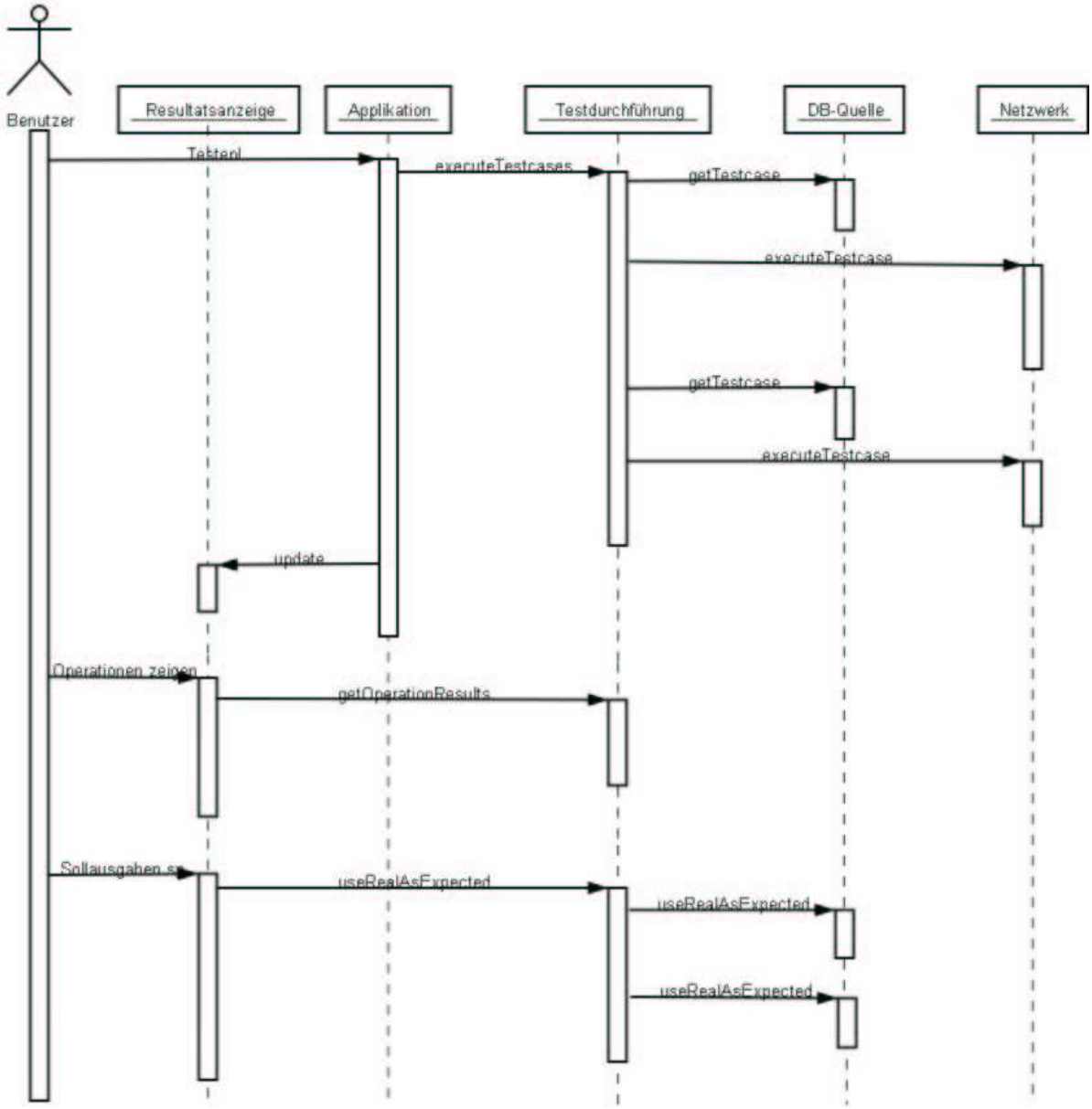

Abbildung 3.15: Testdurchführung

Am Ende der Woche programmierte ich dann die Präsentationsschicht für den Test-Dispatcher in der Netzwerkkomponente. Sie bildet die oberste Schicht der Komponente und erfüllt zwei Aufgaben: zum Einen das Umwandeln der Daten eines Testfalls in einen langen String, zum Anderen das Auslesen der Ist-Daten aus einem String(Buffer) in die Ausgabe-Instanz. Für die Erzeugung des Strings war dabei hauptsächlich darauf zu achten, dass die einzelnen Teilfelder die richtige Länge bekommen, sie sind also je nach Typ entweder hinten mit Leerzeichen oder vorne mit Nullen aufzufüllen. Beim Auslesen erkannte ich (jetzt erst), dass bei unerwarteten Meldungen mehr gelesen werden muss als geschrieben wurde; damit beim 'Nachlesen' die tieferen Schichten nichts ändern, musste ich ihre Schnittstelle um eine weitere Methode ergänzen.
Um die Verarbeitung schnell zu halten, teilte ich den Buffer nicht auf, sondern sprang beim Lesen jeweils zur richtigen Stelle; dadurch wird unnötiges Kopieren von Daten vermieden. Die Parameter erforderten jeweils eine spezielle Behandlung, weil Ein- und Ausgabeparameter in verschiedenen Listen gehalten werden, die Schnittstelle einer Operation aber immer beide enthält. Ich habe einen typischen Misch-Algorithmus verwendet, um die Parameter zu-

sammenzuführen, wie er z.B. im MergeSort-Verfahren vorkommt (ermöglicht durch die Speicherung der Nummer des Parameters):

```
...
while ((inputParam != null) && (outputParam != null)) {
  if (inputParam.getNumber() <= outputParam.getNumber()) {
    // include parameter value
    inputParam = inputIt.hasNext() ? (Parameter) inputIt.next() : null;
  }
  else {
   // reserve space for output parameter
    outputParam = outputIt.hasNext() ? (Parameter) outputIt.next() : null;
}
// now only input params or only output params remain
...
```

Für den rekursiven Abstieg innerhalb von zusammengesetzten Parametern leistete mir das Besucher-Muster gute Dienste:

```
/**
 * Reads a parameter from the output buffer
 * @param s the output buffer
 * @param nr the number of the parameter
 * @param index the starting index of the parameter in the buffer
 * @param expectedParam the parameter that serves as a structure blueprint
 * @return the new parameter
 */
private Parameter readParameter(final StringBuffer s, final int nr,
                                final int index,  Parameter expectedParam) {
  // different methods for reading simple or composite parameters.
  // visitor pattern is used here
  ParamVisitor reader = new ParamVisitor() {

    public Object visitSimpleParameter(SimpleParameter param) {
      String value = s.substring(index, index + param.getLength());
      return Entities.getFactory().createSimpleParameter(
          param.getName(), value, nr);
    }

    public Object visitCompositeParameter(CompositeParameter param) {
      CompositeParameter result =
        Entities.getFactory().createCompositeParameter(param.getName());
      result.setNumber(nr);
      List subParameters = result.getParameters();
      List expectedParameters = param.getParameters();
      int subIndex = index;
      int subNr = 0;
      Iterator it = expectedParameters.iterator();
      while (it.hasNext()) {
        Parameter subParam = (Parameter) it.next();
        // recursion
        subParameters.add(subNr, readParameter(s, nr, subIndex, subParam));
        subNr++;
        subIndex += subParam.getLength();
      }
      return result;
    }
  };
  return (Parameter) expectedParam.visit(reader);
}
```

Die Testdispatcher-Schicht prüft außerdem den Return-Code des Dispatchers selbst und verpackt als oberste Schicht die verschiedenen Java-Ausnahmen, die bei der Socketverbindung auftreten können, in meine eigenen, von der Verbindungsmethode unabhängigen Ausnahmen. Zum Test der Schicht erweiterte ich den schon erwähnten Mirror-Stub.

9. Woche (KW 37): DB-Zugriff, Gesamttest

Ich hatte jetzt die Realisierung des DB-Zugriffs so lange wie möglich herausgezögert, weil Jans Betreuer im Urlaub war und Jan deswegen an anderen Aufgaben arbeitete. Nun wurde eine bessere Abstimmung aber wieder möglich und ich ging an die Implementierung der DB-Komponente.

Zuerst klammerte ich das Schreiben von Sollausgaben aus – was meine Schnittstelle ja ermöglichte – und konzentrierte mich auf das Einlesen der Testdaten. Dank der gehörten Datenbankvorlesungen und JDBC-Erfahrungen aus anderen Softwareprojekten war ich zuversichtlich, vor keine großen Probleme gestellt zu werden, was sich auch bestätigte.
Viele der DB-Abfragen werden mehrfach hintereinander ausgeführt (z.B. das Auslesen der Meldungen für jede Operation eines Testfalls), wodurch sich der Einsatz der Klasse 'PreparedStatement' anbot. Hierbei wird eine bestimmte Art Abfrage von der Datenbank vorkompiliert, dann nur noch parametrisiert und mehrfach ausgeführt. Es verwirrte mich zunächst, dass bei der Parametrisierung die Zählung der Parameter entgegen der üblichen Konventionen bei 1 beginnt, doch dank der Java-Dokumentation war der Fehler rasch gefunden.
Ein Problem ergab sich nur bei der Zuordnung von Parameterelementwerten zu den richtigen Parametern. Durch die rekursive Struktur der Datentypen reicht eine Verknüpfung der Parameterelementwert-Tabelle mit der Parameter-Tabelle, der Operationsaufruf-Tabelle und der Datentyp-Tabelle nicht aus; man braucht zur korrekten Abfrage die gesamte akkumulierte Positionsinformation des Datentyp-Elements, zu dem der Wert gehört. Man hätte hierfür nun auch die Werte-Tabelle rekursiv erweitern können oder bei Begrenzung der Verschachtelungstiefe explizite Felder für jede Vorgängerposition definieren können (also Position im höchsten Datentyp, Position im zweithöchsten, usw.). Weil es dem rekursiven Methodenaufruf entgegenkam und die Anzahl der DB-Abfragen begrenzte, sah ich eine Codierung der gesamten Positionsinformation in einem akkumulierten Textfeld vor – für das 3. Element im 4. Element des Datentyps eines Parameters z.B. '0.3.2'. Ich stimmte die Änderung mit Jan ab, ebenso ein paar kleinere Änderungen die Querschnittskomponenten auf dem Host betreffend (für einige fehlten noch Felder in der Datenbank).

Jetzt war ich soweit, dass ich erstmals den gesamten Test-Client integriert testen konnte. Ich legte mir eine lokale Kopie der Datenbank an, die ich (relativ mühsam, vor allem wegen den rekursiven Elementwerten der Parameter) mit einer Anzahl an Daten für den Test befüllte. Der Test deckte dann auch eine Reihe von Fehlern auf, vor allem innerhalb der Testdispatcher-Schicht in der Netzwerk-Komponente. Zum Beispiel war die Reaktion des Clients auf zusätzliche Meldungen falsch, die Verarbeitung von Operationen ohne Parameter stimmte nicht und ein paar Längenangaben mussten angepasst werden. Durch einen Belastungstest entdeckte ich, dass nach ca. einem Dutzend Testvorgänge die Performance der DB-Komponente dramatisch einbrach. Dies lag daran, dass ich die benutzten Statements nicht explizit geschlossen hatte, sondern mich darauf verlassen hatte, dass sie nach Verlassen der Methode durch den Garbage Collector automatisch geschlossen würden. Jedoch hält das Connection-Objekt intern noch eine Liste der offenen Statements, und weil ich die Verbindung zur Daten-

bank offen hielt, trat der Garbage Collector nicht in Aktion. Ein Umschließen der jeweiligen Zugriffe durch Code des Musters

```
Statement statement = null;
try {
  statement = connection.createStatement();
  ...
}
finally {
  try {
    if (statement != null) statement.close();
  }
  catch (SQLException e) {
    Logger.javaExceptionCaught(...);
  }
}
```

schaffte Abhilfe (die SQLException darf nicht weitergeworfen werden, weil sie eventuell aufgetretene frühere Ausnahmen überdecken würde).

Schließlich schrieb ich noch den Code für die Speicherung von Istausgaben als zukünftige Sollausgaben. Hierfür verfolgte ich zwei unterschiedliche Strategien: Parameterwerte und Returncodes änderte ich durch 'update'-Statements an den entsprechenden Stellen, wobei ich für die Parameter wiederum ein PreparedStatement und das Besucher-Muster einsetzte. Bei den Meldungen konnte sich jedoch die Anzahl ändern, was in manchen Fällen das Löschen oder Hinzufügen von Datensätzen erforderlich machte. Daher entschied ich mich, nicht zuerst umständlich zu prüfen, welche Meldungen schon vorhanden waren, sondern die Meldungen einer Operation komplett zu löschen und für die aus der Istausgabe immer neue Datensätze einzufügen.

Beim Schreiben der Sollausgaben traten nur zwei kleinere Probleme mit dem ODBC-Treiber von Microsoft auf: zum Einen scheint er ein 'insert'-Statement mit expliziter Angabe der Spaltenreihenfolge à la 'insert into messages (var1, var2, ...) values (....)' nicht zu unterstützen, zum Anderen konnte ich das Auto-Commit-Verhalten des Treibers nur direkt nach der Herstellung der Verbindung ändern, danach nicht mehr (JDBC-Connections haben per default ein gesetztes Auto-Commit, was ein Commit nach jedem DB-Zugriff bedeutet. Ich wollte jedoch die gesamte Änderung der Sollausgaben einer Operation in einer Transaktion ablaufen lassen und explizit Commit oder Rollback anfordern, wofür das Auto-Commit ausgeschaltet werden muss). Beide Fehler konnten leicht durch die Einhaltung der Spaltenreihenfolge in der Tabelle bzw. durch ein Ausschalten des Auto-Commits direkt nach Herstellung der Verbindung umgangen werden.

Am Ende der Woche programmierte ich noch den Dialog zur Einstellung von Optionen. Das Design zeigt Abbildung 3.16; mittels einer JTabbedPane werden verschiedene Einstellungsseiten ermöglicht. Ich definierte Schnittstellen für die Views (und ihre Visualisierung), die die einzelnen Seiten verwalten. Die übergeordnete View fragt beim Schließen mit Ok in einem zweiphasigen Prozess erst alle Seiten, ob die eingetragenen Werte korrekt sind und fordert sie dann zum Speichern der Werte auf.

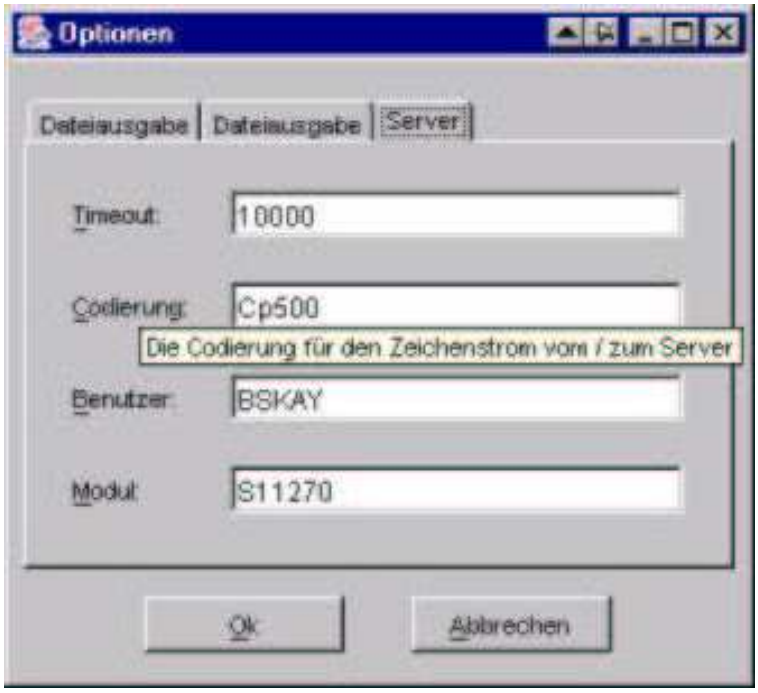

Abbildung 3.16: Optionsdialog

10. Woche (KW 38): Verarbeitung von Dateien

Die Woche begann mit einem neuerlichen internen CD-Meeting. Hier demonstrierte ich zum Einen den derzeitigen Stand des Clients, zum Anderen stellten Jan und ich die letzten Änderungen am Datenmodell vor. Außerdem diente das Meeting zur Festlegung des Dateiformats für den Test-Client. Das Format sollte kompatibel zum Format der Dateien sein, die vom Universalttesttreiber verwendet werden, und es stellte sich dabei die Frage, ob dieser teilweise erweitert werden sollte. Ich schlug vor, für die notwendigen Erweiterungen (die hauptsächlich die Initialisierung der Querschnittskomponenten betreffen), spezielle Zeileneinleitungen zu verwenden, die der Universaltesttreiber als Kommentar ansieht. Indem ich den Test-Client beim Einlesen der Datei diese Einleitungen wegschneiden lasse, kann eine spätere Erweiterung des Universalttesttreibers ohne Änderungen am Test-Client geschehen. Diesem Vorschlag wurde zugestimmt. Eine Eingabedatei besteht somit aus drei Abschnitten: einem Header mit Format und Formatversion, einem Header pro Testfall und einem Abschnitt pro Operation. Das Format ist zeilenorientiert, wobei die verschiedenen Zeilen an Schlüsselworten erkannt werden, die am Anfang der Zeile stehen – z.B. "_TESTFALL" oder "_GVKZ". Kommentarzeilen fangen mit "#" an, die nur vom Test-Client erkannten Schlüsselworte werden durch die Einleitung "#§" maskiert.
Nach dem Meeting wurden ein paar kleinere Änderungen im Test-Client erforderlich (z.B. die zusätzliche Anzeige des Modulnamens in der Operationsergebnis-Übersicht), die mich aber nicht lange aufgehalten haben.

Nachdem der Test-Client nun im Prinzip funktionierte und mit Zustimmung aufgenommen worden war, war die letzte noch fehlende große Funktionalität die Verarbeitung von Eingabedateien. Das Dateiformat stand jetzt ebenfalls fest, und daher konnte ich direkt zur Konstruktionsphase übergehen.
Von Anfang an setzte ich mir das Ziel, auch Sollausgaben in die Datei schreiben zu können (sonst wäre es zu einfach gewesen :-)). Dabei sollte so wenig wie möglich von der ursprüng-

lichen Datei verloren gehen, z.B. sollten Kommentare nach Möglichkeit erhalten bleiben. Dies schloss ein komplettes Neuschreiben der Datei aus – die neuen Werte mussten exakt an die Stellen geschrieben werden, an denen die alten gestanden hatten. Java bietet für diese Anforderung die Klasse RandomAccessFile an, mit der direkt an eine bestimmte Stelle in einer Datei gesprungen werden kann. Leider ist diese Klasse aber byte- und nicht zeilenbasiert, weshalb ich für das zeilenbasierte Eingabeformat einen Adapter schreiben musste. Dieser Adapter speichert drei HashMaps: eine Abbildung von logischen Zeilennummern auf Byte-Positionen, eine Abbildung von logischen Zeilennummern auf reale Zeilen, und die Inverse zu letzterer Abbildung. Die Unterscheidung zwischen logischen und realen Zeilen wurde nötig, weil beim Schreiben von Meldungen Zeilen hinzugefügt oder gelöscht werden können, ich aber außerhalb des Adapters die Zeilennummern nicht daraufhin anpassen wollte. Der Adapter kann dann Methoden wie `readLine`, `deleteLines` oder `replaceLine` anbieten, die auf logischen Zeilennummern basieren.

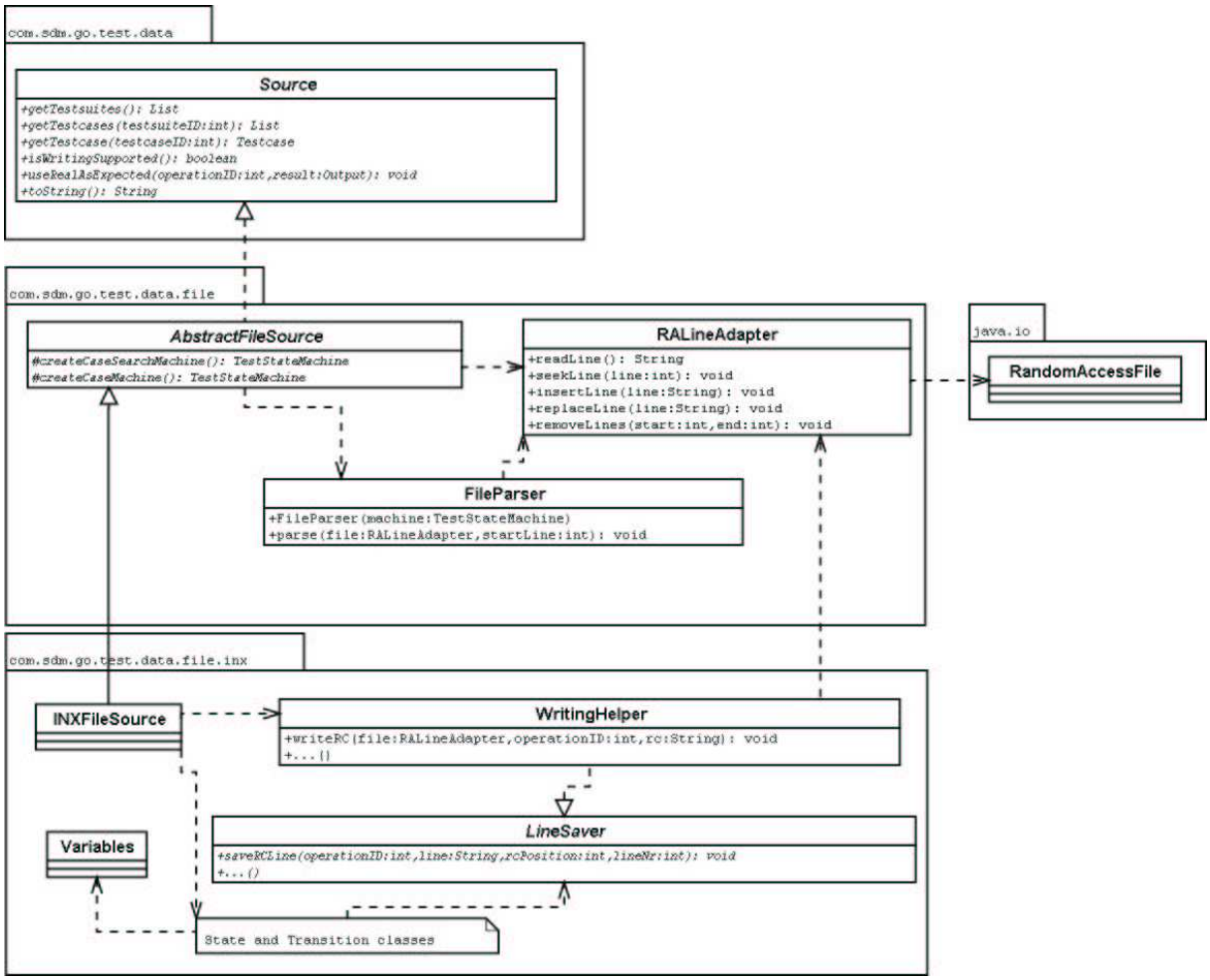

Abbildung 3.17 : Klassen zur Datei-Verarbeitung

Beim Einlesen speicherte ich dann die erforderlichen Informationen (bei Parameterwerten z.B. die Zeilennummer, die Zeile ohne den Wert und die Position des Wertes in dieser Zeile), so dass das Schreiben eines neuen Parameterwertes sich sehr einfach gestaltete:

```
String line = info.getLine();
line = line.substring(0, info.getPosition())
       + parameter.getValue()
       + line.substring(info.getPosition());
try {
  file.seekLine(info.getLineNr());
  file.replaceLine(line);
}
catch (java.io.IOException e) {
  throw new IOException(e.getMessage(), e);
}
```

Leider bereitete das Schreiben der Meldungen, das die Zeilennummern veränderte, erheblich mehr Probleme, weil ich häufig kleine Denkfehler machte (z.B. um die logische Zeilennummer der nächsten Zeile zu bekommen, die bisherige Nummer um eins erhöhte – falls Zeilen eingefügt werden, erhalten sie jedoch neue (hohe) Nummern, so dass die Abfolge der logischen Zeilennummern nicht mehr der Abfolge der realen Zeilennummern entspricht). Außerdem muss das Programm natürlich die (wenn auch sinnlose) mehrfache Speicherung der Sollausgaben hintereinander unterstützen, und schließlich darf auch eine erneute Ausführung der Testfälle (bei der die Datei neu gelesen wird) nicht zu Problemen führen. Die Behebung dieser Fehler kostete mich einen ganzen Arbeitstag.

Bevor gespeichert werden kann, muss jedoch die Datei erst einmal eingelesen werden. Aufgrund der Kenntnisse aus verschiedenen Informatik-Vorlesungen (z.B. Compilerbau) entschied ich mich dafür, einen endlichen Automaten für das Parsen zu bauen. Schnell wurde jedoch klar, dass bei strikter Einhaltung der Automatenprinzipien die Zahl der Zustände und Transitionen ausufern würde, und ein Werkzeug, das einen Automaten generieren würde, stand mir nicht zur Verfügung. Daher entschied ich mich dafür, manche für den Automaten 'globale' Informationen – wie z.B., ob GV-Daten bereits gelesen wurden – in einer extra Klasse als Attribute zu speichern, statt sie in Zuständen zu modellieren. Außerdem ließ ich viele Transitionen aufgrund dieser Informationen selbst entscheiden, in welchen Zustand sie führen sollten. So kam ich mit einer Transition pro Zeilenart aus. Ich entwarf zwei Automaten: einen, der schnell über die Datei läuft und nur die Testfälle sucht (und die Zeilennummer ihres Anfangs speichert), und einen, der komplette Testfälle parsen kann. Der eigentliche Parser wurde so vom Automaten und damit vom Dateiformat unabhängig und besteht hauptsächlich aus einem Auseinanderschneiden der einzelnen Zeilen mittels eines StringTokenizers. In vereinfachtem Code – ohne Fehlerbehandlung und Sonderfälle – sieht er so aus:

```
Integer lineNr = new Integer(startLine);
String line = null;
String token = null;
Transition t = null;
try {
  file.seekLine(startLine);
  do {
    line = file.readLine();
    if (line == null) line = "EOF"; // special transition for EOF
    line.trim();
    StringTokenizer st = new StringTokenizer(line, " ");
    token = st.nextToken();
```

```
    t = machine.getTransition(token);
    if (machine.isAllowed(t)) {
      String[] tokens = new String[st.countTokens()];
      for (int i = 0; i < tokens.length; i++) tokens[i] = st.nextToken();
      Keywords arguments = t.checkLine(tokens, lineNr);
      arguments.addValue("LineNumber", lineNr);
      machine.trigger(t, arguments);
    }
    lineNr = new Integer(file.getCurrentLine());
  } while (!line.equals("EOF") && !(machine.getState().isEndState()));
```

Abbildung 3.17 zeigt ein vereinfachtes Klassendiagramm der Datei-Komponente, Abbildung 3.18 den Automaten zum Parsen von Testfällen.

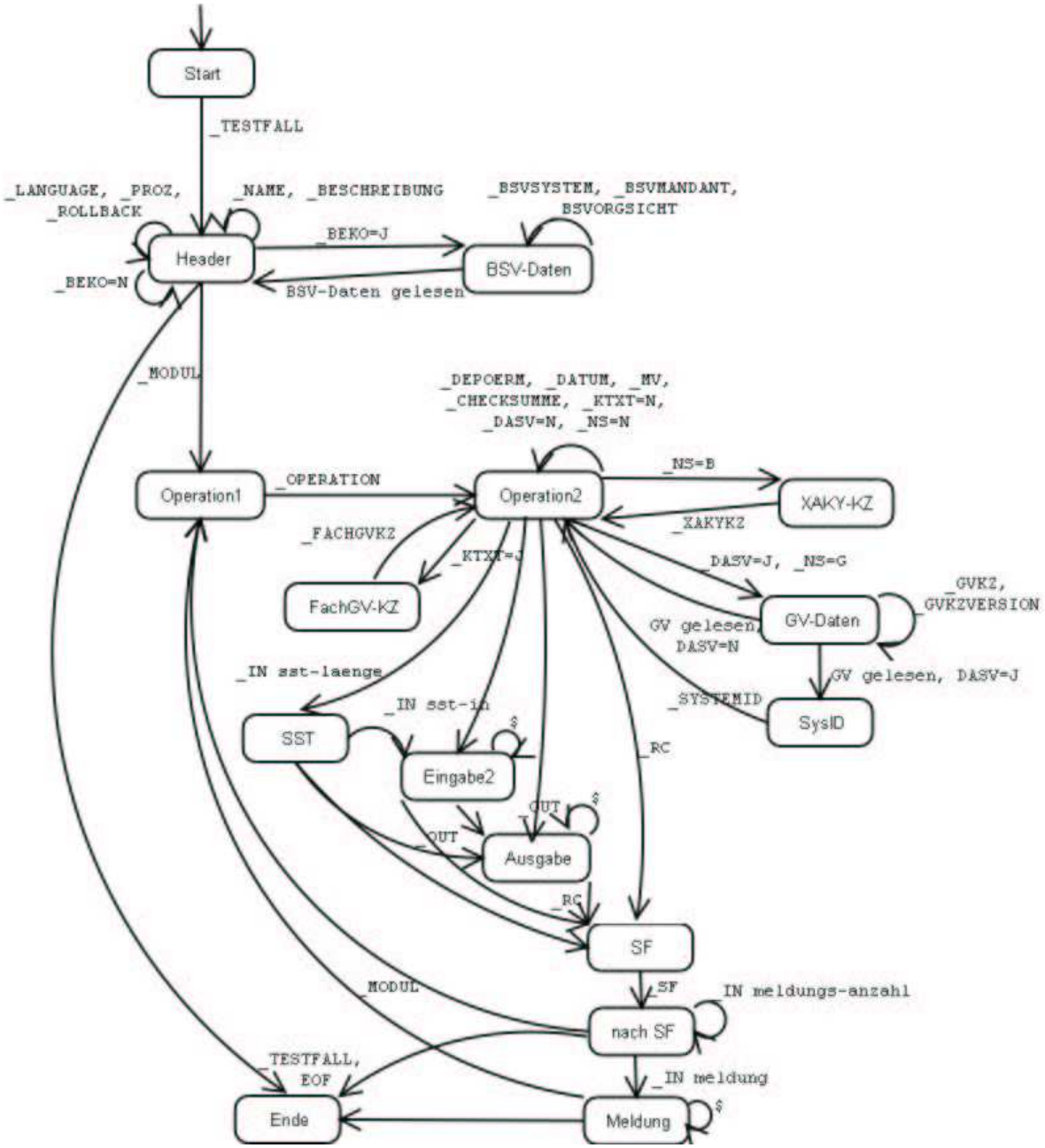

Abbildung 3.18: Automat zum Parsen von Testfällen

Selbstverständlich war für mich die Ausgabe von aussagekräftigen Fehlermeldungen zusammen mit einer Zeilennummer, falls der Inhalt der Datei unkorrekt war. U.a. geben viele Fehlermeldungen aus, welche korrekte Eingabe hätte folgen können (anhand der erlaubten Transitionen).

11. Woche (KW 39): Installation, Dokumentation

Nachdem die Dateiverarbeitung endlich funktionierte, fügte ich noch die letzte fehlende Funktionalität hinzu: das Schreiben von Testprotokollen. Dies war einfach und ich brauchte nur einen Tag dafür. Ich definierte eine neue Seite im Optionsdialog, wo eingestellt werden kann, welches Format verwendet werden soll, ob auch die Sollausgaben ausgegeben werden sollen und ob nur die Fehler oder alle Ausgaben protokolliert werden sollen. Das Format definiert eine ein Interface implementierende Klasse, die dann zur Ausgabe dynamisch nachgeladen wird. Ich stellte zwei Formate bereit: ein plain-text Format, das für Menschen gut lesbar ist, und ein XML-Format, das durch andere Programme weiterverarbeitet werden könnte.
Danach habe ich mich mit dem Deployment und der Installation des Test-Clients beschäftigt. Es stand für mich von vorneherein fest, dass er als jar-Archiv ausgeliefert werden sollte. Zusätzlich müssen die Archive der JEFF Views mitgeliefert werden. Es gelang mir nicht, die Ressourcen-Dateien ins Archiv zu integrieren, so dass sie immer noch vom Programm gefunden werden; ich habe für sie also noch ein weiteres Archiv erstellt. Danach informierte ich mich bei sun über den Inhalt der Meta-Informationsdateien in einem jar-Archiv (Manifest) und erstellte eine solche für den Test-Client. Insbesondere nützlich sind die Angaben

- Class-Path: hier kann man zusätzliche Verzeichnisse und Archive für den classpath definieren. Das war für die JEFF-Views und die Ressourcen erforderlich
- Main-Class: hier kann man die Hauptklasse der Anwendung festlegen, daraufhin kann die Anwendung durch einen simplen Aufruf von `java -jar <archiv>.jar` gestartet werden

Weiterhin kann man für jedes Paket – bei mir nahezu äquivalent zur Komponente – den Namen, die Spezifikations-Version, den Hersteller und das Implementierungsdatum angeben, wovon ich im Sinne einer guten Dokumentation auch Gebrauch gemacht habe.

Daraufhin stellte sich die Frage nach der Installationsprozedur. Der Aufwand für ein eigenes Installationsprogramm erschien mir unnötig hoch. Die erforderlichen Aufgaben bestehen in der Einrichtung der Datenbank als ODBC-Quelle und der Vorbelegung von Optionen mit Default-Werten, insbesondere im Hinblick auf die Umgebungen und DB-Profile, die ja auch als Optionen abgespeichert werden. Ich habe mich dafür entschieden, die Einrichtung der Datenbank nicht zu automatisieren (zu kompliziert), sondern statt dessen eine detaillierte Anleitung im Nutzungshandbuch zu geben. Außerdem schrieb ich ein einfaches Programm (eine Klasse), das Default-Optionen aus einer Datei liest und abspeichert. Die Ablage der Default-Optionen in einer Datei hat den weiteren Vorteil, dass sie einfach geändert werden können, ohne am Programmcode Änderungen vornehmen zu müssen. Mit Hilfe des Programms kann man auch schnell diese Default-Optionen wieder restaurieren, falls durch eine Fehlbedienung des Test-Clients falsche Einstellungen entstanden sind.

Das Dateiformat habe ich sehr einfach gehalten: Leerzeilen und Zeilen, die mit '#' beginnen (Kommentare) werden ignoriert, die restlichen Zeilen haben folgendes Format:
<Komponente><Leerzeichen (LZ)><Optionsname><LZ><Typ><LZ><Wert>, z.B.
`com.sdm.go.test.application askAtResultsLeaving boolean true`

Beim Parsen leistete mir die schon aus der Datei-Komponente bekannte Klasse java.util.StringTokenizer wieder gute Dienste. Auch hier müssen die Leerzeichen mit zurückgeliefert werden, weil der Wert Leerzeichen enthalten darf (falls er ein String ist). Bei Fehlern in einer Zeile wird das Parsen mit einer Warnmeldung fortgesetzt, um zumindest die korrekten Zeilen einzulesen.

Als ich mich dann dem Nutzungshandbuch zuwenden wollte, entdeckte ich im Intranet, dass vor ein paar Tagen eine neue Version der JEFF Views herausgegeben worden war. Ich las mir die Release Notes durch und entschied, dass eine Umstellung zwar etwas Zeit kosten würde, weil die neue Version nicht hundertprozentig mit der alten kompatibel war, aber teilweise zu einfacherem Code führen würde und vor allem Lösungen für ein paar Probleme brachte, die ich bisher von Hand umgehen musste – z.B. die Generierung eigener Argumente für automatisch ausgelöste Kommandos oder die Aktualisierung eines Objektes durch den ViewVisualizer. Der hauptsächliche Aufwand in der Umstellung lag dann zum Einen darin begründet, dass der ViewManager (eine allgemeine Verwaltungsklasse für die Views, die u.a. Methoden zur Anzeige von Standard-Meldungsdialogen bietet) kein Singleton mehr war und die statische Methode getViewManager() verloren hatte. Statt dessen haben die Views eine eigene Methode getViewManager(). Zum Anderen musste ich die Überprüfung der Datentypen bei den Objekteingabe-Views aus der View in ihren Visualizer verlegen – offenbar hatte ich vorher falsch verstanden, wo diese Überprüfung statt finden sollte. Die neue Methode zur Objektaktualisierung erzwang nun eine Überprüfung direkt während der Aktualisierung.
Beide Umstellungen waren jedoch recht schnell erledigt. Längere Zeit benötigte ich, weil der ViewManager auch eine andere Initialisierung brauchte, die aber leider in der Dokumentation nicht explizit beschrieben war (abgesehen vom javadoc). Nach einigen Fehlversuchen, die zu Abstürzen des Programms führten, gelang es mir jedoch, die richtigen Argumente für den Konstruktor zu finden.

Insgesamt kostete mich die Umstellung ca. einen Arbeitstag. Danach machte ich mich an die Erstellung des Nutzungshandbuches. Dieses sollte wie erwähnt eine detaillierte Installationsbeschreibung enthalten; außerdem sah ich einen kurzen Durchlauf durch einen typischen Testvorgang vor. Besonders wichtig für das Handbuch war die große Anzahl von Screenshots, deren Aufnahme mit Gimp jedoch kein Problem darstellte.
Das Nutzungshandbuch hat schließlich folgendes Inhaltsverzeichnis erhalten:

A. Installation
    1. Voraussetzungen (z.B. Plattenplatz)
    2. Einrichtung der Datenbank
    3. Installation des Programms
    4. Start des Programms
B. Überblick über das Testen
    1. Auswahl der Datenquelle
    2. Auswahl der Testfälle
    3. Testdurchführung und Ergebnisanzeige
    4. Weiterverwendung der Ergebnisse
C. Anhang
    1. Bekannte Fehler und Einschränkungen
    2. Weiterführende Dokumentation

12. Woche (KW 40): Dokumentation

Diese Woche bestand nur aus drei Arbeitstagen, weil der 3. Oktober Nationalfeiertag ist und ich mir den darauf folgenden Freitag im Zuge von Überstundenausgleich frei nahm.

In dieser Woche habe ich mich hauptsächlich mit der Erstellung des als Systemdokumentation und Wartungsdokument dienenden DV-Konzeptes des Test-Clients beschäftigt. Weil der Test-Client im Rahmen des GO-Projektes in mehrerer Hinsicht einen Sonderfall darstellt, musste ich mir hierfür zunächst grob überlegen, welchen Inhalt das DV-Konzept eigentlich haben sollte.
Ich habe mir dazu die Vorlage für DV-Konzepte und Systemdokumentation aus dem GO-Projekt angesehen und ein paar reale DV-Konzepte aus GO-Teilprojekten überflogen. Schnell wurde mir klar, dass viele Teile dieser Dokumentstruktur für mich nicht von Nutzen sein würden, weil der Test-Client zwar mit einem Modul auf dem Host kommuniziert, aber selbst keine Host-Software beinhaltet – während in einem typischen GO-Teilprojekt nur die Dialoge in Java realisiert werden (und dies so 'dünn' wie möglich), die Anwendungslogik aber komplett auf dem Host liegt. Auch ist die vom Test-Client verwendete Datenbank nicht mit den üblichen GO-Datenbanken zu vergleichen, und schließlich fehlte mir in den DV-Konzepten ein expliziter Abschnitt mit Hinweisen zur Wartung der Software. Eine Vorlage für DV-Konzepte von reinen Java-Anwendungen konnte ich auch im Intranet nicht finden; dies wird wohl bei sd&m von Projekt zu Projekt unterschiedlich gehandhabt.

Ich habe daher nur einzelne Punkte aus der Vorlage übernommen und mir den restlichen Inhalt – u.a. basierend auf meinen Erfahrungen in den Studienprojekten – selbst überlegt. So kam ich schließlich zu folgendem Inhaltsverzeichnis:

1. Einleitung und Systemüberblick (insbesondere Architektur)
2. Externe Schnittstellen (Datenbank, Dateiformat, Kommunikation mit dem Host)
3. Beschreibung der Komponenten (Aufgabe, Aufrufschnittstelle, benutzte Komponenten, Hinweise zur Implementierung)
4. Lösung typischer Wartungsaufgaben
5. Hinweise an Entwickler (Beschreibung der Projektdateien, der Testtreiber usw.)
6. Anhang (Entitätendefinition, Referenzen etc.)

Es war sd&m leider nicht möglich, mir für die Dokumentation ein marktübliches CASE-Werkzeug wie z.B. Together oder Rose zur Verfügung zu stellen, weil die Lizenzen der Programme an die einzelnen Teilprojekte des GO-Projektes gebunden sind und ich keinem dieser Teilprojekte zugeordnet war. Ich habe die nötigen UML-Diagramme deshalb mit einem Freeware-Werkzeug (dia) gezeichnet, das Objektbibliotheken dafür zur Verfügung stellt, allerdings leider keine dahinterliegende Logik besitzt und schon gar kein Re-Engineering unterstützt. Die Diagramme sind in den früheren Wochenberichten schon abgebildet.

Für das DV-Konzept konnte ich ein paar Abschnitte aus dem in der 4. Woche erstellten Architekturdokument übernehmen. Dieses wurde hingegen durch die nun vorhandene Dokumentation obsolet.

13. Woche (KW 41): Abschlussarbeiten

In der letzten Woche meines Praktikums wurde endlich (und gerade noch rechtzeitig) eine erste Version des Test-Dispatchers fertig, so dass der Test-Client mit der realen Gegenstelle getestet werden konnte. Außerdem bekam ein neuer Werkstudent, Stefan Scherer, in München die Aufgabe, vor allem Jan's Arbeit fortzuführen (Jan hatte sd&m schon am Ende der vorigen Woche verlassen), aber auch die Wartung des Test-Clients zu übernehmen. Ich fuhr deshalb nochmals für einen Tag nach München, um ihm das Datenmodell und die Architektur des Test-Clients zu erklären und ihm auch sonst generell zu zeigen, an welchen Stellen er zur Lösung bestimmter Aufgaben suchen musste.
Außerdem erstellte ich in dieser Woche diesen Praktikumsbericht anhand der Notizen und Dokumente, die ich schon im Verlauf des Praktikums angelegt hatte.
Die verbleibenden 2 Arbeitstage in Kalenderwoche 42 konnte ich als Überstundenausgleich frei nehmen – dies hatte ich auch von Anfang an so geplant, weil die Vorlesungszeit hier wieder begann.

4. Anhang

4.1 Begriffslexikon

Hier sind einige Begriffe kurz erklärt, die im Bericht häufiger vorkommen und außerhalb des Berichtes unbekannt sind oder häufig eine andere Bedeutung haben.

Client	s. Test-Client
Host	hier: ein zentraler Server im GO-Projekt, mit dem der Test-Client kommuniziert
JEFF Views	Framework von sd&m research zur GUI-Entwicklung
Kommando	im Kontext der JEFF Views ein vom Benutzer aufrufbarer Befehl, den eine View anbietet
Komponente	hier: eine größere, zusammenhängende und austauschbare Software-Einheit
Modul	hier: eine Software-Einheit auf dem Host, die verschiedene Operationen anbietet
Quasar	die Standard-Softwarearchitektur bei sd&m
Querschnitts-Komponenten	Komponenten, die von vielen anderen Komponenten genutzt werden. Auf dem Host: müssen vor dem Test von manchen Operationen initialisiert werden. Beim Test-Client: Optionen, Logging und Lokalisierung
Test-Client	Das von mir entwickelte Java-Programm, das den Test im GO-Projekt automatisiert
Test-Dispatcher	Modul auf dem Host, das Operationen aufruft und ihre Ausgabe zurück liefert
View	GUI-Komponente für einen (Teil-)Dialog
(View-)Visualizer	Teil einer View, der die Anzeige der Daten verwaltet

4.2 Weitere Literatur

Zur Erläuterung der verschiedenen angesprochenen Entwurfsmuster empfehle ich das Buch 'Design Patterns' von Gamma, Helm, Johnson und Vlissides (Addison-Wesley, 1995). Zu Quasar soll bald ein neues Buch erscheinen; einstweilen verweise ich auf eine erste Broschüre im Internet (http://www.sdm.info/download/pdf/1/sdm_quasar_teil-1.pdf) und das Buch 'Software-Technik' von Brössler und Siedersleben (Hanser, 2000), in dem schon einige der Prinzipien von Quasar beschrieben wurden.