Perl-Gestützte Softwarelieferprozesse

Gerd Aschemann (*gerd@aschemann.net*)

25. Februar 2003

Zusammenfassung

In einem großen Projekt wird Software für eine Multi-Tier-Architektur entwickelt. Die Versionskontrolle obliegt Rational ClearCase, das Change-Management Rational ClearQuest. Die entstehende Software muss in verschiedenen Teststufen (Assembly-, System-, Technischer-, Integrations-, Abnahme-Test) und später für die Produktion auf verschiedene Servertypen (Backend, Application-/Web-Server, Application-Proxy), die teilweise mehrfach für jede Stufe vorhanden sind (Verfügbarkeit, Lastausgleich), geliefert werden. Dabei ist ein hoher Qualitätsstandard einzuhalten, der beispielsweise

- die Konformität der Versionen, die auf die einzelnen Schichten geliefert werden,
- die Nachvollziehbarkeit der Lieferungen,
- die Einhaltung aller Schritte des Lieferprozesses,
- Optimierungen der Liefermengen (nur Komponenten, deren Bestandteile im Rahmen von Bugfixes geändert wurden, müssen neu geliefert werden)

sicherstellt.

Der Prozess ist durch ein Zustandsübergangsmodell in ClearQuest spezifiziert. Die zu jedem Zustandsübergang erforderlichen Aktionen sind in Form einer Klassenbibliothek und darauf aufbauender Skripte in Perl implementiert. Die Basisfunktionen von ClearCase und ClearQuest werden über entsprechende Perl-Schnittstellen eingebunden.

Der Beitrag zeigt die Erfahrungen bei der Erstellung der objektorientierten Perlbibliothek und der Skripte ($>$ 20.000 Zeilen Perlcode). Neben den klassischen Stärken von Perl (String-Verarbeitung, Pattern-Matching, Hashes, Integration von/mit anderen Werkzeugen, automatisches Testen, ...), steht vor allem die Kombination von Perl mit einem zeitgemäßen Softwareentwicklungsprozess (OOA/D, Entwurfsmuster, Team-Arbeit) im Mittelpunkt des Beitrags.

1 Einleitung

In großen Software-Projekten, die aus mehreren Teams und zahlreichen Entwicklern bestehen, muss mit angemessenen Entwicklungswerkzeugen und wohldefinierten Prozessen die Qualität des Endproduktes sichergestellt werden. Ein Teil der Prozesse betrifft die Zusammenführung und Lieferung der Software-Komponenten in Test, Abnahme und Produktion. Das Software-Konfigurationsmanagement sorgt mit einem Versionskontrollsystem für die konsistente Zusammenführung der parallel entwickelten Komponenten, das Changemanagement für die Erst-Lieferung in verschiedene Test-Instanzen und die Produktion, bzw. später auch für die Lieferung von Updates und Bugfixes.

Bei jedem Software-Projekt wird der Entwicklungs- und Testprozess unterschiedlich definiert. Es gibt zwar häufig unternehmensweit allgemeine Richtlinien, Erfahrungen und Vorgaben. Trotzdem muss der Prozess jeweils auf das zu entwickelnde Softwaresystem abgestimmt sein. Werkzeuge zum Konfigurations- und Changemanagement bieten daher in der Regel einen allgemeinen Rahmen für die Implementierung der Prozesse an, erlauben jedoch durch eigene Konfigurationsmöglichkeiten und APIs (Application Programming Interface) eine individuelle Gestaltung dieses Prozesses.

Die in unserem Projekt eingesetzten Werkzeuge Rational ClearCase (Konfigurationsmanagement, [RAT99b]) und ClearQuest (Changemanagement, [RAT99a]) bieten unter anderem eine Perl-API zur Implementierung der Prozesse. Der Beitrag beschreibt eine Perl-Klassenbibliothek, die, aufsetzend auf den bestehenden Werkzeugen und ihren APIs, den Lieferprozess eines Software-Projektes realisieren.

1.1 Aufbau des Beitrages

Der Beitrag beschreibt zunächst das eigentliche Softwaresystem mit seiner Architektur und seinen verschiedenen Entwicklungs- und Testumgebungen. Anschließend geht er auf den Konfigurations- und Changemanagement-Prozess und dessen Realisierung vor Einführung der Klassenbibliothek ein. Eine Bedarfsanalyse zeigt Schwachstellen des bisherigen Prozesses auf und definiert Anforderungen für seine Weiterentwicklung. Im Hauptteil des Beitrags werden Entwurf und Realisierung der Klassenbibliothek diskutiert, bevor er in einem zusätzlichen Kapitel die Erfahrungen, Erweiterungen, usw. bespricht, und weitere interessante Details aufzeigen. Eine Zusammenfassung und ein Ausblick auf zukünftige Aktivitäten runden den Artikel ab.

2 Softwareumgebung und -lieferprozess

Der Softwarelieferprozess definiert, wie die entwickelten Komponenten von der Entwicklungsumgebung über verschiedene Teststufen bis in die Produktion geschoben werden. Das betrifft sowohl komplett neue Releases (Volllieferungen), wie auch Bugfix-Lieferungen (Teillieferungen). In der Regel sind die Release-Zyklen so kurz (ca. sechs Monate), dass zwischen zwei Volllieferungen nur ein oder maximal zwei Teillieferungen in die Produktion erfolgen. Zunächst werden die Software-Umgebungen mit den enthaltenen Hardware- und Software-Komponenten beschrieben, sowie die verschiedenen Teststufen. Anschließend wird genauer auf den Prozess des Konfigurations- und Changemanagements eingegangen, bevor abschließend die bestehenden Skripte, in die die weitere Entwicklung der Perl-Bibliothek eingebettet wird, kurz beschreiben werden.

2.1 Zielumgebung

2.1.1 Softwarearchitektur

Bei der Anwendung handelt es sich um ein verteiltes Vertriebssystem eines großen Verkehrsbetriebes, das als Mehrschichtenarchitektur (Multi-Tier-Architektur), aus — je nach Sichtweise — mindestens drei Schichten aufgebaut ist (vgl. Abb. 1):

1. Das Backend enthält neben der klassischen Datenbank mit entsprechenden Zugriffsdiensten auch Teile der Geschäftslogik und insbesondere die Integration nachgelagerter Verfahren, wie Angebotserstellung, Buchung, Platzreservierung und Ticketing. Diese werden von einem Alt-System bis auf weiteres noch weiter benutzt und um Komponenten zum personalisierten Verkauf (Kunden-Stammdaten, -Präferenzen etc.) mit entsprechender Datenhaltung ergänzt.
2. Im (vom Projekt so genannten) Frontend gibt es einen sog. Distributionskern (DBK), der verschiedene Vertriebskanäle (Call Center, Direkter Verkauf, Reisebüros, Großkunden) bedienen kann. Für manche Vertriebskanäle, insbesondere Call-Center und in Zukunft auch den direkten Verkauf, enthält das Frontend noch eine Präsentationsschicht, die als Web-Applikation auf Basis von HTML und JavaScript realisiert ist. Andere Vertriebskanäle werden über geeignete BusinessToBusiness (B2B)-Protokolle an externe Partner angebunden, z.B. als WebServices über das Simple Object Access Protocol (SOAP).
3. Je nach Vertriebskanal enthält die Client-Schicht nur Präsentations-Clients (Windows NT, Internet-Explorer) oder auch Umsetzungen der jeweiligen B2B-Protokolle und Integrationen mit dem Ver-

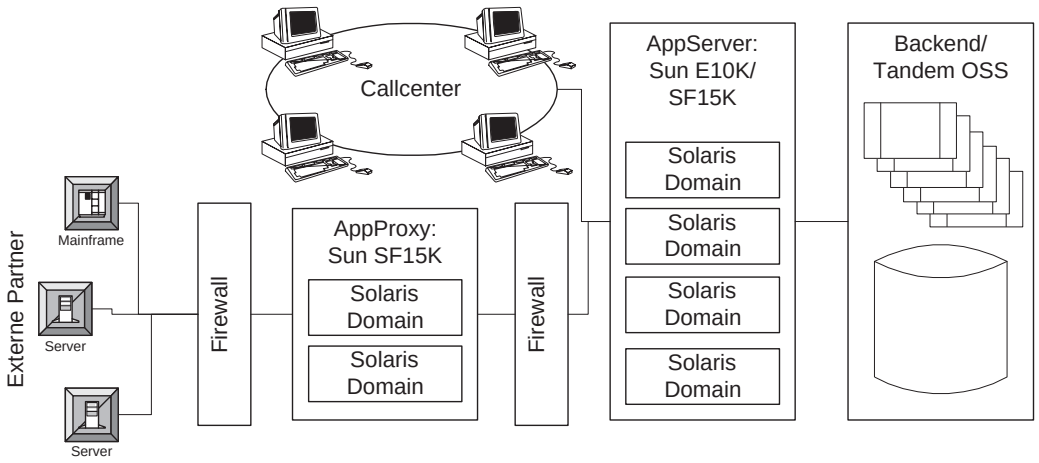

Abbildung 1: Software-Architektur

kaufssystem des externen Vertriebspartners. Für externe Vertriebspartner ist noch ein Application Level Gateway in einer Demilitarized Zone (DMZ) zwischen Client und Frontend geschaltet.

Die Anwendung soll auf Dauer das bisherige Vertriebssystem, welches hauptsächlich Direktvertrieb mit sogenannten (unter SCO-Unix realisierten) Intelligenten Terminals, die über X.25 mit einem Hostsystem kommunizieren, erlaubt, ersetzen. Daneben sollen neue Vertriebskanäle (Call-Center, Agenturen, etc.) erschlossen werden. Für eine mehrjährige Übergangszeit müssen alte und neue Systeme nebeneinander laufen und das alte System muss in das neue integriert werden. Auch nach Abschluß der Migration werden Komponenten des Alt-Systems im neuen enthalten bleiben.

2.1.2 Heterogene Hardware- und Softwarelandschaft

Die Komponenten von Backend und Frontend kommunizieren über Remote Procedure Calls (RPCs) auf Basis von Tuxedo als Transaktionsmonitor. Die entsprechenden Stubs und Skeletons der selbst entwickelten RPC-Schicht werden aus XML-Darstellungen der Schnittstellen mittels eines Perl-Skriptes generiert, was aber nicht Gegenstand dieses Artikels ist.

Eine Zielumgebung besteht aus jeweils einem zentralen Tandem System im Backend, mehreren BEA WebLogic (BEA WLS) Applikationsservern unter Solaris im Frontend und zur Anbindung Externer Parteien aus mehreren Applikations-Proxyservern (ebenfalls BEA WLS/Solaris), die die Applikationsserver als Application Level Gateway abschotten.

Ein Tandem-System bietet Hochverfügbarkeit durch redundante Auslegung aller Systemkomponenten (CPUs, Busse, Plattenspiegelung etc.) mit Hot-Plugin und -out aller Komponenten. Die Backend-Services werden in C++ realisiert, für die Datenbankschicht kommen generierte C-Module zum Einsatz. Die Services sowie Batches der Applikation laufen unter Tandem Open System Services (OSS, POSIX-Umgebung), während einige wenige Komponenten unter dem proprietären Tandem NonStop-Kernel (NSK) eingesetzt werden.

Die Frontend-Systeme, sowohl im Applikationsserver, wie auch im Applikationsproxy sind in Java realisiert. Dabei wird zum Teil auf die Standard Java-Enterprise-Architektur (J2EE, Java 2 Enterprise Environment) gesetzt, zum Teil auf eigene Komponenten zur Verteilten Implementierung, welche unter anderem auch den RPC-Zugriff zum Backend per WebLogic Tuxedo Connector (WTC) einschließt.

Als Web-Clients dienen speziell zusammengestellte Windows-Systeme unter NT mit Internet-Explorer. Für ausgewählte Clients im sogenannten Fullfillment Center, in dem Tickets und andere buchbare Leistungen zum Postversand ausgedruckt werden, sind die NT-Clients um einfache Webserver zur

Ansteuerung eines lokalen Druckers durch das Frontend erweitert. Externe Parteien realisieren SOAP-Clients unter verschiedenen Umgebungen, neben Unix-Systemen auch unter BS 2000.

2.1.3 Entwicklungsumgebung

Die Modellierung der Software erfolgt mittels Objektorientierter Analyse und Design (OOA&D) mit Rational Rose. Entwickelt wird unter Windows NT (künftig evtl. auch unter Linux oder Solaris), wofür im Bereich Java ein entsprechendes Integrated Development Environment (IDE, Borland JBuilder), sowie BEA WebLogic unter NT verwendet wird. Für die in C++ realisierten Backend-Komponenten kommt die Tandem-Developer-Suite (TDS), ebenfalls von Borland, zum Einsatz, da diese einen Cross-Compiler und Remote-Debugger für Tandem-Systeme enthält. Dahinter steht ein Tandem-Rechner für Entwicklung und die ersten Teststufen (s.u.).

2.1.4 Test: Client- und Servertypen

Als Web-Client werden entweder Entwicklerarbeitsplätze genutzt oder es kommen die gleichen NT-Clients zum Einsatz, wie sie auch in der Produktion genutzt werden. Als Clients zum Test der Anbindung Externer Parteien werden ebenfalls NT-Systeme, oder kleinere Solaris-Systeme verwendet. Im "Technischen Test" werden für Last- und Performancetests spezielle NT-Systeme, die als Lastgeneratoren unter Mercury LoadRunner konfiguriert sind, verwendet.

Als Server kommen sehr unterschiedliche Modelle zum Einsatz. Im Frontend reichen die verwendeten Server von einfachen NT-Systemen oder Sun UltraSparc 5 bis zu Partitionen von Sun E15K-System.

Im Backend werden Tandem-Server partitioniert, so dass gewisse Ressourcen für einzelne Testumgebungen zur Verfügung stehen. Hier sprechen wir von Backend-Umgebungen, da nicht alle Ressourcen exklusiv für einzelne Teststufen reserviert werden, sondern z.T. zwischen mehreren Teststufen gemeinsam genutzt werden. Zusätzlich muss die Backend-Umgebung auch die Einbindung nachgeschalteter Systeme (s.o.) realisieren.

2.1.5 Teststufen und Produktion

Wir unterscheiden zwischen verschiedenen Teststufen, sowie der Produktionsumgebung (vgl. Abb. 2. Jede dieser Stufen hat eine spezielle Bedeutung, die im folgenden kurz skizziert werden und enthält in der Regel sämtliche Teilsysteme der Anwendung.

Assemblytest. Der Modulgruppentest erfolgt im Frontend in einer Umgebung, die der Entwicklungsumgebung entspricht, also unter NT läuft. Im Backend steht ein Tandemsystem für die Tests bereit, das auch für Entwicklung und andere Teststufen benutzt wird.

Fachliche Systemtestumgebung Die fachliche Systemtestumgebung setzt als Präsentationsclients produktionsgleiche NT-Systeme ein, als Frontend kleinere Solaris-System (etwa E 250), die auch nur einfach ausgelegt sind, im Backend das gleiche Tandemsystem wie Entwicklung, Assemblytest und technischer Systemtest. Fachliche Tests der Web-Services zur Anbindung Dritter erfolgen über einen in Java geschriebenen Testclient.

Technische Systemtestumgebung Die technische Systemtestumgebung, in der unter anderem auch Last- und Performancetests, Installations- Systemverwaltungs-, Backup- und technische Exceptiontests durchgeführt werden, entspricht von ihrer Auslegung schon weitgehend der Produktionsumgebung. Das Backend-Tandemsystem ist zwar das gleiche wie in den vorhergehenden Teststufen, einige Ressourcen sind jedoch exklusiv für den Technischen Test reserviert und bei Last- und Performancetests, die fast ausschliesslich Nachts und am Wochenende stattfinden, wird das gesamte System für diese Tests reserviert. Im Frontend stehen zweimal vier Domains aus jeweils zwei Prozessoren auf zwei Sun E10K oder

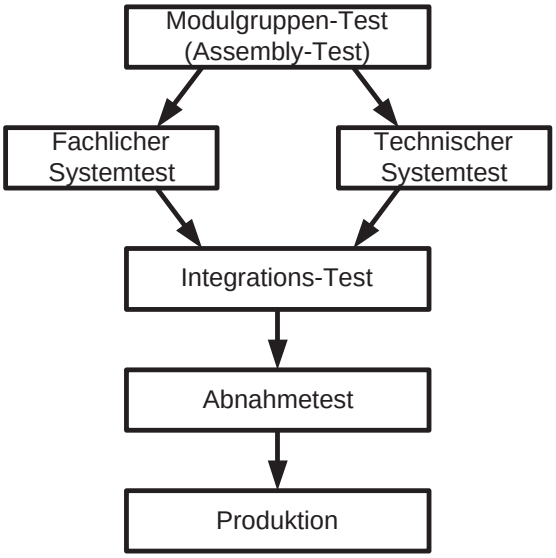

Abbildung 2: Teststufen

SF15K zur Verfügung (Application-Server bzw. Application-Proxies, wobei letztere in einer DMZ stehen). Auf jeder Domain läuft jeweils ein BEA WLS, die zusammen einen Cluster bilden und ein Apache-Server. Als Clients werden für Last- und Performancetests Lastgeneratoren (Windows-NT-Systeme) eingesetzt, die nach einem vorgegebenen Lastprofil einen Betrieb von mehreren hundert Call-Center Agents bzw. mehreren tausend externen Mitarbeitern simulieren. Typische Transaktionen (Verbindungsauskünfte, Angebote, Buchungen/Reservierungen, Änderungen etc.) wurden dabei mit einem entsprechenden Werkzeugen (Mercury LoadRunner) zunächst aufgezeichnet und zur Lasterzeugung später mit geeigneten Abfragedaten wieder abgespielt.

Integrations- und Abnahmetest Die Teststufen Integrations- und Abnahmetest dienen den Tests zur Integration mit parallelen Verfahren, die auf den gleichen Systemkomponenten laufen, bzw. natürlich der Abnahme des Gesamtsystems durch den Auftraggeber. Die jeweiligen Umgebungen entsprechen der Produktionsumgebung.

Produktion Die Produktionsumgebung besteht aus mehreren hundert Call-Center-Systemen, zwei Sun SF15K-Systemen im Frontend, wovon eine für den Application Proxy in einer DMZ steht, den Klienten der externen Vertriebspartner und einem dedizierten Tandem-System im Backend.

2.2 Konfigurations- und Change-Management

Diese komplexe Welt von verschiedenen Testumgebungen, muss durch ein ausgefeiltes Konfigurations- und Change-Management kontrolliert werden. Die Komplexität wird dadurch noch erhöht, dass zeitweise bis zu vier Releases (oder noch mehr) parallel existieren: das zuletzt aus der Produktion genommene Release, auf das man ggf. zurückfallen möchte, das aktuell in Produktion befindliche, das in der Implementierung befindliche und das nächste oder die nächsten geplanten, deren prototypische Implementierung auf Testsystemen oder in der Konzeptphase befindlichen Releases.

Aufgabe des Konfigurationsmanagements ist es unter anderem, die zahlreichen, von verschiedenen Teams erstellten Softwarekomponenten so bereitzustellen, dass sich auf jeder Umgebung jeweils eine funktionsfähige, konsistente Gesamtversion installieren läßt. Dabei müssen nicht nur die verschiede-

nen Komponenten geeignet zusammengestellt werden, sondern insbesondere auch umgebungsspezifische Einstellungen (Benutzer, Adressen etc.) zu berücksichtigen.

Das Change-Management dient dazu, Fehler und Änderungswünsche zu erfassen, ihre Bearbeitung zu verfolgen und dafür zu sorgen, dass diese in konsistente Konfigurationen einfließen. Das Change-Management ermöglicht letztendlich die Lieferung konsistenter Konfigurationen.

2.2.1 Lieferpakete und Änderungslieferungen

Um den Konfigurations- und Änderungs-Prozess, der weitgehend kein technischer, sondern ein organisatorischer ist, durch automatisierte Abläufe zu unterstützen, wird die Software-Entwicklung grob in zwei Phasen unterteilt:

- Jedes Team entwickelt in der Entwicklungsphase für ein neues Release eines oder mehrere Release-Pakete (RELP), die idealerweise jeweils eine Komponente des Systems beinhalten. Alle RELPs zusammen bilden ein abgeschlossenes und (hoffentlich) in der Assembly-Testumgebung lauffähiges und integriertes Entwickler-Release.
- Mit Auslieferung der RELPs geht der Entwicklungsprozess in die Testphase über, in der idealerweise keine funktionalen Änderungen mehr vorgenommen werden, sondern nur noch Fehler gefunden werden, die von den Entwicklern repariert werden[1]. Die Fehler und Änderungswünsche werden als sogenannte Software-Investigation-Request (SIR) verfolgt.

Allgemein bezeichnen wir ein RELP oder einen SIR als Software-Änderungs-Antrag (SAA).

2.2.2 Liefer-Baselines

Das Konfigurationsmanagement wird durch das Tool “ClearCase” von Rational Software unterstützt. Dieses versioniert nicht nur die Quellcodes der Komponenten, sondern auch daraus generierte Bibliotheken, Executables, Jar-Files etc. Eine konsistente Konfiguration aller Bestandteile einer Komponente wird durch eine sogenannte Baseline definiert: Die Baseline erfasst wie bei einem Schnappschuss die zugehörigen Bestandteile der Komponente in einem bestimmten Zustand. Im Lieferprozess werden immer Komponenten gemeinsam ausgeliefert, die zu einer bestimmten Baseline gehören. Die Baseline(s) eines Teams werden von Teams, deren Komponenten auf den Komponenten dieser Baseline aufbauen, in ihre eigene Konfiguration übernommen. Dadurch ist erkennbar, wie die Baselines der verschiedenen Teams zusammengehören.

2.2.3 Lieferungen

Bei einer Lieferung (wir sprechen auch von Migration) von der Entwicklung bzw. von einer Teststufe in die nächste werden die Komponenten aller zusammengehörigen Baselines aller Teams gemeinsam in die jeweilige Umgebung geliefert. Die Lieferung beinhaltet für jede Komponente ggf. ihre Generierung, evtl. mit anderen Generierungsoptionen (z.B. mit Debug-Optionen für den Assembly-Test, ohne Debug-Option/mit Optimierung ab Systemtest) und den Transfer in die jeweilige Zielumgebung. Manche Komponenten werden dabei einfach per FTP in die Zielumgebung kopiert, andere werden erst noch zu größeren Objekten zusammengefasst (Java Web-Applikationen z.B. als “war”-File).

2.2.4 Zustandsübergangsmodell

Das Change-Management wird durch Rational ClearQuest unterstützt. In der Entwicklungs- und Test-Phase durchläuft jeder SAA nochmal mehrere Zustände (vgl. Abb. 3):

[1] Im real existierenden Software-Entwicklungsprozess werden auch in dieser Phase noch funktionale Änderungen vorgenommen oder zur Fehlerbehebung nötig, was das gesamte Verfahren nicht gerade vereinfacht.

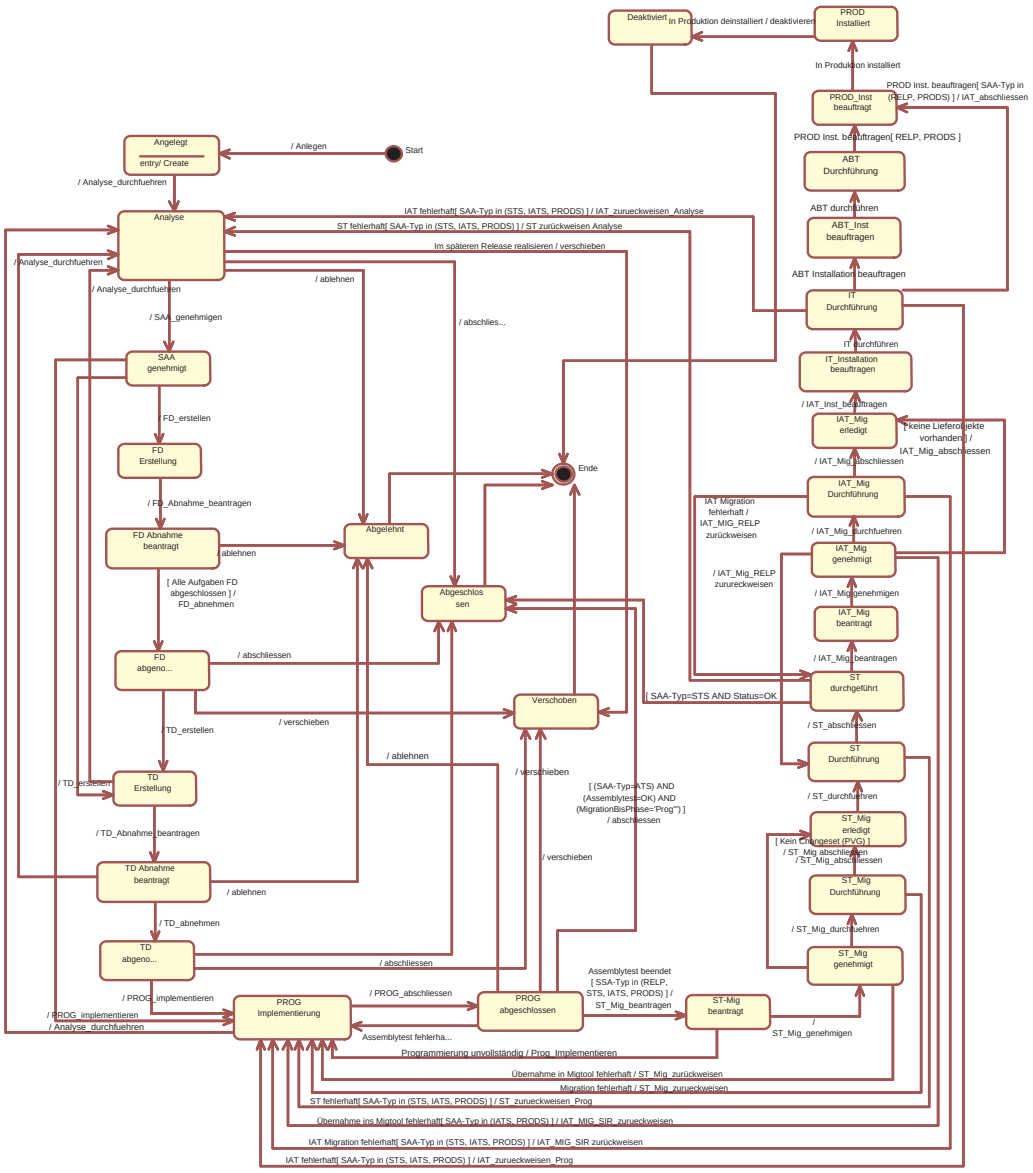

Abbildung 3: Zustandsmodell in ClearQuest

1. Angelegt
2. Analyse
3. Fachliches/Technisches Design
4. Implementierung
5. Programmierung abgeschlossen
6. Migration beantragt
7. Migration durchführen
8. Migration abgeschlossen
9. Systemtest
10. Systemtest abgeschlossen
11. Integrationstest
12. etc.

Alle Zustände sind in ClearQuest modelliert und pro Komponente in einer SQL-Datenbank abgelegt. Jedem RELP bzw. jedem SAA lassen sich in ClearQuest eine oder mehrere Aufgaben zuordnen, die jeweils einen Bearbeiter (Entwickler) haben und zu genau einem Zustand gehören. Nur im Rahmen dieser Aufgaben lassen sich in ClearCase Dateien (Quellcode) neu ins Projekt aufnehmen bzw. ändern (Checkout/Checkin). Damit ist idealerweise auch die Zugehörigkeit einer (Quell-) Datei zu einer Komponente implizit geklärt. SIRs beziehen sich auf RELP, so daß theoretisch auch hier eine Zuordnung von geänderten und insbesondere neu aufgenommenen Dateien zu RELPs automatisch getroffen wird. Prinzipiell kann man mit einer Aufgabe aber auch beliebige andere Dateien einer anderen Komponente bearbeiten, was auch nötig ist, da ein Fehler sich zum Beispiel auf das Zusammenspiel zweiter Komponenten beziehen kann. Erst wenn alle Aufgaben, die z.B. im Rahmen des Implementierungszustandes geöffnet waren, abgeschlossen werden, kann der Zustand des SAAs in ClearQuest weitergeschaltet werden: So ist sichergestellt, dass niemand mehr Erweiterungen/Änderungen am Quellcode vornimmt, wenn der SAA bereits in einem weitergehenden Zustand ist.

2.2.5 Der Lieferprozess aus ClearQuest

Der gesamte Lieferprozess läßt sich aus der graphischen Benutzungsoberfläche von ClearQuest steuern, weil an jeden Zustandsübergang nicht nur Datenbank-Trigger geknüpft werden können, die Änderungen im Zustandsmodell vornehmen oder Konsistenzprüfungen durchführen, sondern auch externe Programme, deren Resultate Auswirkungen auf den Zustandsübergang haben. Während der ersten zwei Releases waren die Zustandsübergänge und die Migrationsschritte nur organisatorisch gekoppelt. Zu jedem migrationsrelevanten Zustandsübergang musste der Teamleiter oder eine ausgewählte Person im Team (Migrateur) den jeweiligen Migrationsschritt manuell starten, bzw. nach den durchgeführten Migrationsschritten die Zustandsübergänge durchführen. Dies setzte zum einen eine gute Schulung, gute Übersicht und hohe Sorgfalt voraus, und war zum anderen dennoch ein sehr fehlerträchtiger Prozess. Vor allem aus Qualitätssicherungsgründen musste er daher weiter automatisiert werden. Ein anderer Grund war z.B. die Performance: Da eine komplette Übersicht über alle Komponenten bei keinem der Beteiligten zu jedem Zeitpunkt vorhanden war, wurde im Zweifelsfalle das gesamte Projekt neu gebaut und geliefert, was mehrere Tage dauerte. Ein weiterer Aspekt war die Durchführung des Prozesses: Wenn er durch Programme automatisiert wird, können viele Zustandsübergänge von beliebigen Projektmitarbeitern in ClearQuest durchgeführt werden. Folgerichtig wird der zu automatisierende Migrationsprozess *SAA-basierte Migration* genannt.

2.3 Vorhandene Skripte

In den ersten beiden Releases waren bereits eine Reihe von (Perl-) Skripten entstanden (s.u.), die im manuell gestarteten Migrationsprozess genutzt wurden. Die Skripte hatten zum Teil eine Doppelfunktion: Bauen der Komponenten und Lieferung in die jeweilige Zielumgebung. Durch eine entsprechende Parametrisierung konnte auch nur jeweils eine der beiden Funktionen aufgerufen werden, was die Skripte jedoch ziemlich komplex gemacht hat.

Die Skripte sollten für die Implementierung der SAA-basierten Migration erhalten bleiben und um weitere ergänzt werden. Sie waren zum Teil dadurch entstanden, dass bestehende Skripte kopiert und getrennt zu verschiedenen Werkzeugen weiterentwickelt worden waren, ohne Modularisierung und Codesharing gemeinsam benötigter Funktionen zu beachten. Das führte nicht nur zu unnötig parallelem Pflegeaufwand, sondern zum Teil sogar zu inkompatiblem Verhalten.

Kurz nach Beginn der Implementierung der SAA-basierten Migration, und dem Start der "Implementierung" eines weiteren Skripts zur Ermittlung der Lieferobjektmengen (vgl. Abschnitt 2.3.5) durch Kopieren und Ändern zeigte sich dann, dass neben den bereits anvisierten Skripten ein weiteres Skript benötigt wurde. Dieses sollte zum reinen Liefern umgebungsspezifischer Dateien für das gerade in Entwicklung befindliche dritte Release dienen. Da diese Funktionen sich weitgehend mit der bei der SAA-basierten Migration im Schritt "Migration abschließen" deckte, wurde entschieden, dieses Skript nicht direkt durch die herkömmliche Implementierungstechnik "Kopieren" und "Anpassen" zu erstellen. Statt dessen sollte diesmal etwas mehr Aufwand in eine grundsätzliche Analyse und das Design einer Klassenbibliothek mit Basisfunktionen für alle Skripte gesteckt werden. Darauf aufbauend, sollte dann zunächst das neue Transferwerkzeug erstellt werden. Anschließend sollten die vorhandenen Skripte überarbeitet werden, wobei gemeinsamer Funktionsumfang entfernt und durch Aufrufe der Klassenbibliothek ersetzt werden sollte.

2.3.1 Label-Vergabe

Eine wichtige Aufgabe im Migrationsprozess ist die Vergabe sogenannter Labels in ClearCase: Jedes Objekt (Datei oder Verzeichnis) bzw. jede Version eines Objektes kann eines oder mehrere Labels tragen. Es kennzeichnet einen bestimmten Zustand, beispielsweise sind Baselines durch ein einheitliches Label gekennzeichnet. ClearCase ist u.a. in der Lage, bestimmte (konsistente) Sichten (Views) auf eine Menge von Objektversionen zu erlauben, beispielsweise alle Objekte einer Baseline. Der Zustandsvergabe in ClearQuest entsprechen zum Teil die Zuordnung bestimmter Labels in ClearCase. Sobald zum Beispiel eine Menge von Objektversionen (eine Baseline) erfolgreich den Assemblytest durchlaufen haben, wird bei ihnen das sog. ASS-Label gesetzt, um dies zu kennzeichnen. Nur Objektversionen mit ASS-Label werden für den folgenden Migrationsschritt (Systemtest) berücksichtigt. Nach erfolgreichem Systemtest erhalten sie das SYS-Label und dürfen in den Integrationstest etc. Für die Vergabe des ASS-Labels gibt es ein eigenes Skript `label.pl`.

2.3.2 Backend-Generierung

Die Generierung (und zunächst auch der Transfer) von Backend-Komponenten wird im Wesentlichen durch `make` bzw. eine ClearCase-eigene Variante davon (`omake`) durchgeführt. Allerdings sollten die Entwickler aus verschiedenen Gründen keine `Makefiles` pflegen, u.a. wegen der Fehlerträchtigkeit für ungeübte Entwickler (die eigentliche Entwicklung von Basisbibliotheken und Services lief ausschließlich über TDS), oder der benötigten unterschiedlichen Parametrisierung des Build-Prozesses (Debug vs. Optimierung).

2.3.3 Setfiles und Templates

Daher beschreiben sogenannte *Setfile*s die Komponenten. Sie und die in den Skripten implementierten Parser und Interpreter definieren eine einfache Makrosprache, von der Syntax her ähnlich wie Variablen

in Makefiles. Es gab unter anderem

- ein Makro, das das eigentliche Lieferobjekt (Bibliothek, Batch oder Service) benennt,
- ein Makro, das den Typ des Lieferobjektes beschreibt,
- ein Makro mit einer Liste von Quelldateien,
- ein Makro mit einer Liste von Bibliotheken,
- ein Makro mit dem (ClearCase) Verzeichnis in das das generierte Objekt eingecheckt wurde,
- ein Makro für das Verzeichnis mit Logfiles,
- etc.

Die Makros konnten Referenzen auf andere Makros enthalten, die dann textuell eingesetzt wurden, z.B. gemeinsame Pfade[2].

Aus dem Setfile wurden durch das Backend-Werkzeug `make_gen.pl` mit Hilfe von Templates, in die die Werte der Makros eingesetzt wurden, Makefiles generiert und anschließend `omake` aufgerufen. Nach erfolgreichem Build hat `make_gen.pl` die generierten Objekte per FTP in die Zielumgebung transferiert und ihnen, sowie auch den zugrundeliegenden Quellobjekten ein SYSMIG-Label zum Zeichen der erfolgreichen Migration in den Systemtest gegeben, sowie Logfiles eingecheckt.

Ein weiterer Grund für die Verwendung des Werkzeuges anstelle von `make` direkt ist Qualitätssicherung: Alle am Build-Prozess beteiligten Objekte müssen die entsprechenden Labels haben (ASS) bzw. bekommen (SYSMIG), außerdem muss von Build und Transfer ein Log angelegt werden, das ebenfalls unter Versionskontrolle gestellt werden muss. Ein Parameter des `make_gen.pl`-Aufrufes war zudem die Identifikation des jeweiligen SAAs in ClearQuest. Das Werkzeug musste zunächst prüfen, ob der SAA im Zustand "Migration durchführen" war, um zumindest eine minimale Konsistenzprüfung zwischen durchgeführtem Prozess und dahinterstehenden Zustandswechseln im Change-Management-Modell zu erreichen.

Auch im o.g. Label-Skript wurden Setfiles verwendet, um die zu labelnden Objektversionen zu beschreiben. Hier wurde sich jedoch auf die Angabe (eines Makros mit) einer Menge von Verzeichnissen in ClearCase beschränkt, unterhalb derer alle vorhandenen Objekte gelabelt wurden. Somit hatten wir hier bereits zwei "Dialekte" Setfiles.

2.3.4 Frontend-Generierung

Im Frontend waren die Lieferobjekte im wesentlichen Java jar-Files, die mit Hilfe eines Compileraufrufs (`javac`) und einem Aufruf von `jar` erstellt wurden. Auch hier wurden Setfiles verwendet, die vom Aufbau her zwar ähnlich, aber beispielsweise auf jeweils ein Lieferobjekt (jar-File) beschränkt. Außerdem hießen die Makros zwar ähnlich (z.B. `LIBS` vs. `LIBRARIES`), hatten jedoch teilweise leicht unterschiedliche Semantik, oder auch Syntax, z.B. "," als Trennzeichen vs. ";" bei Aufzählungen. Daneben gab es weitere Unterschiede in der Ausstattung an Makros, beispielsweise, da im Backend Include-Pfade benötigt wurden (C/C++), oder zusätzliche Makros für SQL-Compilierung. Im Frontend wurden Mengen von Quellcodes nicht durch explizite Aufzählung der Dateinamen beschrieben, sondern durch Makros, die Verzeichnisse angeben, die wahlweise rekursiv oder auch nicht rekursiv durchsucht werden sollten, oder auch Muster von .java-Dateien, die bei der Generierung ein- oder auszuschließen sind.

Hier hatten wir im Skript `migrateFrontend.pl` also schon den dritten Dialekt der Setfiles und somit drei verschiedene Parser in den Skripten mit zum Teil überlappender Funktion, zum Teil divergierender. Andere Funktionen überlappen ebenfalls zwischen Frontend und Backend, beispielsweise das Labeln, die Konsistenzprüfung gegen ClearQuest-Zustände oder das Einchecken von Logfiles.

[2] Zunächst konnte ein Setfile im Backend auch noch eine ganze Menge von Lieferobjekten und deren jeweilige Abhängigkeiten beschreiben.

2.3.5 Ermittlung von Lieferobjektmengen

Nach Abschluß der Programmierung eines SIRs muss die Menge der von den Änderungen (Changeset) betroffenen Lieferobjekte bestimmt werden. Dazu war für die SAA-basierte Migration bereits weitgehend ein Skript (`getDeliverables4ChangeSetElements.pl`) erstellt worden. Dieses musste erstmalig eine Vielzahl von Setfiles (Frontend und Backend) sowie weiterer Steuerdateien einlesen. Obwohl es bereits in einem fortgeschrittenen Stadium der Implementierung (als monolithisches Skript) war, war dieses ein guter Kandidat, um es mit Teilen der Klassenbibliothek zu refaktorieren.

2.3.6 Migrations-Controller

Nicht alle Teilprozesse lassen sich direkt innerhalb der grafischen Oberfläche von ClearQuest verankern. Daher wurde ein weiteres Programm entwickelt, um bei den Schritten "Programmierung abschließen", "Migration durchführen" und "Migration abschließen" die Abläufe zu steuern (Kontrollabfragen vor und nach den jeweiligen Schritten) und zu visualisieren. Wegen der vermeintlich (gegenüber Perl) leichteren Nutzung von Komponenten zur Gestaltung der graphischen Oberfläche, wurde dieses Programm (Mig-Controller) als einziges in VisualBasic erstellt.

3 Anforderungen

An dieser Stelle soll nicht die detaillierte Anforderungsanalyse des gesamten KM-Projektes durchgegangen werden, sondern im wesentlichen die Requirements aufgezeigt werden, die für die Neuentwicklung der Klassenbibliothek maßgeblich waren. Dabei gehen wir — wie im Projekt — von Notwendigkeit aus, ein isoliertes Transferwerkzeug zu erstellen, und kommen dann etwas systematischer zu Anforderungen an die zu erstellenden Klassen.

3.1 Initiale Anforderungen an das Transferwerkzeug

Unabhängig von der Entscheidung, eine Klassenbibliothek in Perl aufzubauen, ging es zunächst darum, ein Transferwerkzeug zu erstellen. Dieses sollte ermöglichen, gezielt einzelne Dateien oder auch große Mengen von Dateien über beliebige Medien auf beliebige Zielsysteme zu transferieren.

Die Zielsysteme sind natürlich die Rechner bzw. die Umgebungen in Frontend und Backend. Diese sind im wesentlichen durch eine Adresse (im allgemeinen IP-Adresse, da FTP das bevorzugte Medium ist, s.u.) bestimmt, eine Benutzerkennung (und Authentifizierung) und initiale Pfade, in die geliefert wird. Von diesen Pfaden aus gibt es je nach Umgebung weitere Werkzeuge, die die Inbetriebnahme (Deployment) der gelieferten Objekte vornehmen, was aber nicht Gegenstand dieses Artikels ist. Da eine Umgebung aus mehreren gleichartigen Systemen (WLS-Cluster, Solaris-Domänen, ...) bestehen kann, muss das Werkzeug in der Lage sein, dieselbe Menge an Lieferobjekten sicher auf verschiedene gleichartige Systeme zu transferieren.

In der Regel werden die Lieferobjekte über FTP von der Entwicklungsumgebung zur Testumgebung transferiert. Das Transferwerkzeug sollte jedoch von diesem Transportmedium abstrahieren. Es musste grundsätzlich möglich sein, FTP nicht nur durch andere Transportprotokolle, wie z.B. sftp (eigentlich nur eine spezielle Erweiterung der Secure Shell/SSH) oder rsync (über SSH) zu ersetzen. Vielmehr sollte es auch z.B. möglich sein, die Dateien mit einem Archivierungswerkzeug (tar, zip/jar) zusammenzufassen, oder sie auf CD zu brennen. Tatsächlich wurde z.B. bis zur Einführung der SAA-gesteuerten Migration die Frontend-Lieferung in die weiteren Teststufen ab Systemtest durch Archivierung (zip) der gelieferten Dateien und Weitergabe dieser Dateimenge geregelt. Dieses Verfahren hatte den Nachteil, dass keine Qualitätssicherung mehr möglich ist, weil nicht mehr nachvollziehbar ist, welche Dateien bzw. Dateiversionen tatsächlich in die weiteren Teststufen oder gar in die Produktion geliefert wurden: Bei Problemen, die in einem komplexen Projekt immer auftreten, neigten Entwickler, Tester und Umgebungsverantwort-

liche leicht dazu, “mal eben schnell” eine Datei vor Ort anzupassen, zu verschieben, oder eine neuere Version manuell nachzuliefern.

Ein wichtiger Wunsch war, die immer gleiche Dateimenge auf eine Vielzahl von Systemen ausliefern zu können. Daneben war es aber auch wichtig, diese Mengen und Zielsysteme fast beliebig aufeinander zuzuschneiden. Im Extremfall sollten einzelne, individuelle Dateien auf einzelne Rechnerknoten geliefert werden können, allgemein sollten bestimmbare Teilmengen auf einen Teil oder die Gesamtheit der Zielsysteme geliefert werden. Die Forderung nach einzelnen Dateien betrifft beispielsweise Konfigurationsdateien, rechnerspezifische SSL-Schlüssel und -Zertifikate, oder Datenbankskripten, die auf den Zustand der jeweiligen Datenbank zugeschnitten sind.

3.2 Funktionale Anforderungen

Fachlich bestehen folgende Anforderungen an Transferwerkzeug und Klassenbibliothek.

Festlegung von Lieferobjekten und -mengen: Es sollte möglich sein, Lieferobjekte, ihre Abhängigkeit von Quellobjekten und Mengen von Lieferobjekten zu spezifizieren.

Festlegung von Lieferzielen: Für jedes Lieferobjekt sollte festgelegt werden können, wohin es geliefert werden muss. Dabei sollte eine Definition abgestuft nach

Migrationsstufe, den verschiedenen Teststufen, bzw. der Produktion

Servertyp, nach Backend, Applikationsserver/Webserver, Applikationsproxy, und nach

Rechner, für Lieferobjekte, die nur auf einzelne Rechner geliefert werden müssen

möglich sein.

Definition der Lieferziele: Jedes Lieferziel, also entweder ein einzelner Rechner, oder Gruppen von Rechnern nach Servertyp und Migrationsstufe, sollten spezifiziert werden können.

Berechnung von individuellen Liefermengen: Für einen Lieferprozess sollten die individuellen Liefermengen berechnet werden. Ausgehend von der (z.B. im Aufruf) spezifizierten Menge der Lieferobjekte sollte berechnet werden, welche davon auf die spezifizierte Menge der Zielsysteme geliefert werden muss.

3.3 Nicht-Funktionale Anforderungen

Das Transferwerkzeug sollte als Kommandozeilentool entwickelt werden, bei dem man über Parameter einfach die Lieferobjekte, die Lieferziele und ggf. das zu wählende Medium angibt. Die Klassenbibliothek sollte die gewünschten Funktionen unter besonderer Berücksichtigung folgender nicht-fachlicher Anforderungen bereitstellen.

Effizienz und Skalierbarkeit Eine effiziente Implementierung sollte vor allem folgende Gesichtspunkte berücksichtigen:

Objektmengen Für Lieferungen müssen u.U. sehr große Objektmengen bearbeitet werden, da sehr viele Dateien unter Versionskontrolle stehen, die nicht lieferrelevant sind. Die relevanten Dateien müssen effizient extrahiert werden können.

Einlesen von Konfigurationsdateien soll optimiert werden, so dass Dateien nicht mehrfach gelesen werden müssen.

Evaluierung von Mustern Die Evaluierung von Spezifikationen, beispielsweise um eine Menge von Dateien zu bestimmen, soll erst zu einem Zeitpunkt erfolgen, zu dem die Information auch tatsächlich benötigt wird (Lazy Evaluation).

Dateitransfer Die Dateitransfers sollen so durchgeführt werden, daß alle Dateien für einen Server auf einmal transferiert werden und somit zeitaufwändige Operationen wie Verbindungsaufbau und Benutzerauthentifikation minimiert werden.

Erweiterbarkeit Es war schon frühzeitig absehbar, dass sowohl Skript als auch Klassenbibliothek stark erweitert werden. Einerseits zeichneten sich bereits weitere Anforderungen wie die Erstellung von Java .war- und .ear-Dateien zum Deployment von Web- und Enterprise-Applikationen ab, andererseits sollten auf Dauer viele Funktionen aus den bisher vorhandenen Skripts durch die Klassenbibliothek ersetzt werden.

Flexibilität und Änderbarkeit Abstraktionen sollten so erfolgen, daß zugrundeliegende Implementierungen leicht ausgetauscht werden können, beispielsweise der auf `Net::FTP` aufbauende Dateitransfer.

3.4 Spätere Erweiterungen

Wie bereits mehrfach erwähnt, war schon während der initialen Implementierung klar, dass sowohl das eigentliche Transferwerkzeug, wie auch die Klassenbibliothek auf Dauer einigen Erweiterungen unterworfen werden sollte. Die Klassenbibliothek sollte so entworfen werden, dass derartige Erweiterungen funktional in der Regel durch zusätzliche Methoden oder Klassen abgedeckt werden können. Skripte bzw. Erweiterungen von Skripten sollten sich in der Regel auf eine Prüfung der Eingabeparameter, die Initialisierung entsprechender Objekte und den Aufruf geeigneter Methoden zur Ausführung der gewünschten Funktion beschränken.

4 Entwurf und Realisierung

Der objektorientierte Software-Entwurf sollte uns in die Lage versetzen, die Begriffe, die in der Analyse-Phase identifiziert wurden und zur Beschreibung des Problems und möglicher Lösungsansätze benutzt werden, in einem formalen Modell in entsprechende Klassen umzusetzen. Mit Hilfe der entworfenen Klassen sollte die angeforderte Funktion unter angemessener Berücksichtigung der nicht-funktionalen Nebenbedingungen umgesetzt werden können.

4.1 Objektorientierter Entwurf

Der Gesamtentwurf besteht aus einer Vielzahl von Klassen (vgl. Abb. 4) mit unterschiedlichsten Beziehungen. Im Folgenden seien nur die wichtigsten genannt.

4.2 Setfiles

Ein wesentlicher Begriff, der auch schon bei den “Altlasten” eine zentrale Rolle spielte, sind die Setfiles. Sie dienen dazu, Lieferobjekte zu spezifizieren. Im Backend handelt es sich fast ausschließlich um Executables, also Tuxedo-Services, Batches und sonstige Programme. Mit der aktuellen Release sollten im Backend auch Konfigurationsdateien, Skripts, Datenbank-Definitionsdateien und beliebige andere Dateien hinzukommen. Im Frontend handelt es sich hauptsächlich um Java jar-Dateien, sowie weitere Web-Inhalte (html-Dateien, gifs, JavaScript-, CSS-, Java Properties-Dateien etc.), die zum Teil in die jar-Files eingehen, zum Teil direkt geliefert werden mussten. Manche dieser Lieferobjekte (Executables und jar-Files) müssen zunächst aus anderen Objekten zusammengebaut werden. Ein Setfile enthält eine Menge von zusammenhängenden Lieferobjekten und maximal einem zu bauenden Lieferobjekt. Im Frontend sind das beispielsweise ein (zu bauendes) jar-File, das einen Use-Case implementiert und die dazugehörigen jsp-Dateien.

Ein typisches Frontend-Setfile zum Bauen eines Java jar-Files sieht folgendermaßen aus:

```
SERVERTYPE = webserver, appproxy
TARGET = itpta.jar
SOURCEBASEDIR = $(VIEWROOT)\TA\dev\java\src
SOURCEPATH = de\tlc\sez\itp
```

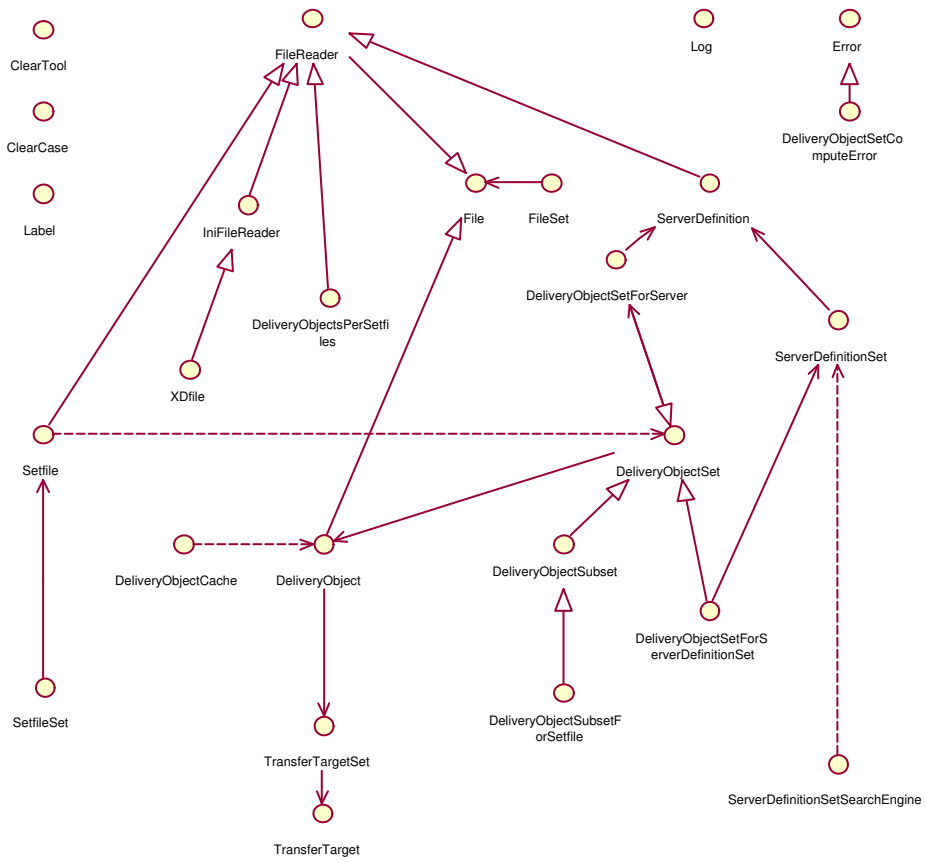

Abbildung 4: Objektorientierter Entwurf der Klassenbibliothek

```
EXCLUDEPATTERN =  \
          test* \
          dbks*.*
RECURSIVE_SOURCE_SEARCH = true
LIBS =  $(VIEWROOT)\TA\dev\java\lib\quartz.jar \
        $(VIEWROOT)\TA\dev\java\lib\cfw.jar \
        $(VIEWROOT)\TA\dev\java\lib\xerces.jar \
        $(VIEWROOT)\TA\dev\java\lib\ldapjdk.jar \
        $(VIEWROOT)\TA\dev\java\lib\weblogic.jar \
        ...
OUTPUTCLASSDIR = $(VIEWROOT)\TA\dev\java\classes
OUTPUTJARDIR = $(VIEWROOT)\TA\dev\java\build\tmp
TARGETCHECKINDIR = $(VIEWROOT)\TA\dev\java\build\lib
LOGCHECKINDIR = $(VIEWROOT)\PMX_DO\log
TRANSFER_1_SOURCEBASEDIR = $(TARGETCHECKINDIR)
TRANSFER_1_SOURCEPATH = .
TRANSFER_1_INCLUDEPATTERN = $(TARGET)
TRANSFER_1_TARGETPATH = WEB-INF/lib
TRANSFER_1_RECURSIVE = false
TRANSFER_1_MODE = binary
```

Ohne auf alle Makros einzugehen, soll damit das Jar-File `itpta.jar` gebaut werden, wobei rekursiv alle Java-Sourcen im Pfad `\TA\dev\java\src` benutzt werden sollen, mit Ausnahme der Dateien, deren Name mit `test` oder `dbks` anfangen. Es werden verschiedene Bibliotheken verwendet (`quartz.jar`, `weblogic.jar`, etc.).

Die letzten sechs Makros sind für den Transfer der Dateien relevant: Die eingecheckte TARGET-Datei soll ins Verzeichnis WEB-INF/lib des Zielservers transferiert werden. Wir bezeichnen diese Folge von Makros als "Transfergruppe" innerhalb des Setfiles. Die "eingebaute" Nummer (`TRANSFER_1_`) der Transfergruppe läßt bereits erahnen, dass es mehrere davon geben kann. Sollen beispielsweise noch jsp-Dateien mittransferiert werden, könnte es eine weitere Gruppe geben:

```
TRANSFER_2_SOURCEBASEDIR = \TA\dev\webserver\jsp
TRANSFER_2_SOURCEPATH = .
TRANSFER_2_INCLUDEPATTERN = *.jsp
TRANSFER_2_TARGETPATH = .
TRANSFER_2_RECURSIVE = true
TRANSFER_2_MODE = binary
```

Diese sorgt für den rekursiven Transfer (unter Beibehaltung der Dateibaumstruktur) der jsp-Dateien ins Zielverzeichnis. Das Zielverzeichnis ist in der Regel ein Zwischenverzeichnis der jeweiligen Umgebung von dem aus das eigentliche Deployment stattfindet.

Die Klasse `KM::Setfile` ist also ein zentraler Bestandteil des Entwurfs. Sie vereinheitlicht weitgehend die Makrosprache zwischen Frontend und Backend (bis auf wenige jeweils optionale Makros), kapselt den Parser, ermöglicht den Zugriff auf die (Makro-) Informationen über Abfrage-Methoden, z.B.

`miglevel` (optional) liefert die Migrationsstufe, in die die Lieferobjekte des Setfiles geliefert werden dürfen,

`servertype_list` (optional) liefert die Servertypen, auf die die Objekte geliefert werden können: Im Frontend unterscheiden wir noch weiter zwischen `WebServer` und `ApplicationProxy`,

`hostname` (optional) falls die Lieferobjekte auf genau einen Rechner geliefert werden sollen,

`libs` liefert die Pfade der Bibliotheken (`.a`- oder `.jar`-Dateien, nach Backend oder Frontend),

und implementiert einige zentrale Algorithmen, z.B.

matchesServerDefinition($serverDefinition) Prüft, ob ein Setfile Lieferobjekte für einen bestimmten Server spezifiziert.

deliveryObjectSet liefert ein Perl-Objekt, das die Lieferobjekte repräsentiert, die von diesem Setfile beschrieben werden.

contains hat als Parameter einen Dateinamen und gibt zurück, ob ein Element des `deliveryObjectSet` von dieser Datei abhängig ist (ohne zuvor das `deliveryObjectSet` zu berechnen!).

4.3 File

Viele Klassen der Bibliothek repräsentieren Daten, die in Files abgelegt werden bzw. die Dateien repräsentieren, beispielsweise als Lieferobjekte. Es liegt also nahe, eine Klasse `File` als Basisklasse zu definieren, die typische Attribute/Methoden bietet, z.B.

fullpath() für den kompletten Pfad ab der ClearCase-Wurzel (ViewRoot), oder

label($label,...) Labeln des Datei-Objekts in ClearCase gemäß einer bestimmten Migrationsstufe.

Zusätzlich definiert die Klasse aber zahlreiche weitere Methoden, die auf Datei-Objekten der Klassenbibliothek sinnvoll sind, z.B.

derivedObject([$flag]) zum Prüfen, oder Setzen, ob das Datei-Objekt von anderen Objekten abhängig ist (beispielsweise als Lieferobjekt),

dependsOn([$fileSet]) um die Dateimenge zu Setzen oder Abzufragen, von der das Datei-Objekt abhängig ist, also beispielsweise die verwendeten Bibliotheken oder Quell-Objekte eines Backend-Services, oder

usedBySetfiles() um die Menge der Setfiles, die das Objekt "aktiv" oder "passiv" enthalten, zu ermitteln.

4.4 Set-Klassen

Mehrere Typen der Klassenbibliothek können in Mengen auftreten. Für diese gibt es jeweils eine Mengen-Klasse, z.B. `DeliveryObjectSet`, welche kanonisch über den Namen der Element-Klasse mit einem angehängten `Set` benannt sind. Sie enthalten typischerweise Operationen, um Elemente hinzuzufügen oder die Menge als Array abzufragen. Hier ist eine Schwachstelle der Objektorientierung in Perl zu beklagen: Es ist leider nicht möglich, auf elegante Weise parametrisierbare Containerklassen zu definieren, wie das z.B. in C++ mit Templates möglich ist. Noch schöner wäre es, von einer parametrisierten Container-Klasse erben und weitere spezifische Methoden hinzufügen zu können.

Da die Container zumindest abstrakt eine gemeinsame Basis-Schnittstelle haben (Hinzufügen von Elementen, Abfragen/Iterieren über die Elemente), wäre vom Entwurf her prinzipiell eine gemeinsame Basisklasse sinnvoll gewesen. Allerdings weisen die verschiedenen Container in ihrer Implementierung (Repräsentation der Mengen) Unterschiede auf, so dass angesichts der relativ wenigen Set-Klassen auf eine gemeinsame Basisklasse verzichtet wurde. Zudem implementieren einige Set-Klassen spezifische Operationen, die auf der Menge der Elemente ausgeführt werden, z.B. das Labeln oder das Bauen aller Lieferobjekte.

Es sind derzeit folgende Set-Klassen definiert:

DeliveryObjectSet : Eine Menge von Lieferobjekten. Diese Klasse hat noch einige Unterklassen, die eine Filterung nach bestimmten Kriterien ermöglichen (vgl. Abschnitt 4.8).

DeliveryObjectSetForServerDefinitionSet : Eine Menge von Lieferobjekten, die auf eine Menge von zugeordneten Servern geliefert werden soll (vgl. Abschnitt 4.8).

FileSet : Eine Menge von Dateien (nicht nur Lieferobjekte, Basisklasse für andere Dateimengen).

ServerDefinitionSet : Eine Menge von Zielsystemen (vgl. Abschnitt 4.8).

SetfileSet : Eine Menge von Setfiles.

TransferTargetSet : Eine Menge von Transfer-Gruppen (vgl. Abschnitt 4.6).

4.5 DeliveryObject

Ein Setfile spezifiziert eine Menge von Lieferobjekten der Klasse `DeliveryObject`. Neben den Methoden, die von der Basisklasse `File` geerbt werden, werden u.a. folgende Methoden spezifiziert:

setfile([$newSetfile]) Fragt das Setfile, über das das Lieferobjekt definiert wurde ab, bzw. setzt es neu. Hierüber wird auch geprüft, ob ein Lieferobjekt von mehreren Setfiles spezifiziert wird: Über seinen (vollen) Dateinamen ist es eindeutig identifizierbar. Es erbt von der Klasse `File`. Alle `File`-Objekte wiederum werden durch ihren Konstruktor in einem zentralen Cache gehalten, der die Objekte über ihren vollen Pfadnamen adressiert, so dass jede Datei eindeutig durch ein Perl-Objekt repräsentiert wird. Bei der Initialisierung einer `DeliveryObject`-Instanz durch die `Setfile`-Klasse wird nicht nur der Konstruktor der Oberklasse bemüht, um das eindeutig identifizierte `File`-Objekt zu finden/initialisieren, sondern auch das `Setfile`-Attribut gesetzt. Sollte schon eines vorhanden sein, gibt es einen Konflikt mit einem anderen `Setfile`.

transferTargetSet() Gibt die Menge der Transfer-Gruppen zurück (vgl. Abschnitt 4.6). Prinzipiell kann das Lieferobjekt in mehreren Transfergruppen enthalten sein, also mehrfach an verschiedene Stellen des Zielsystems geliefert werden.

build() Baut das Objekt (zunächst nicht implementiert, da Konzentration auf Lieferprozess).

Zusätzlich gibt es noch Convenience-Methoden, die zur Vereinfachung des Codes bestimmte Funktionen auf vorhandene Attribute abbilden:

matchesServerDefinition($serverDefinition) Prüft über das `setfile()`-Attribut, ob ein Lieferobjekt auf einen bestimmten Server geliefert werden soll.

addTransferTarget(@newTransferTargets) Fügt eine neue Transfer-Gruppe hinzu.

Eine wichtige Funktion ist die Abfrage, ob das Lieferobjekt auf einen bestimmten Server geliefert werden soll, welche effektiv von der entsprechenden Methode der Klasse `Setfile` geleistet wird.

4.6 TransferTarget

Die Klasse `TransferTarget` müßte eigentlich `TransferGroup` heißen, da sie sämtliche Informationen einer Transfergruppe aus dem Setfile repräsentiert (vgl. Abschnitt 4.2).

4.7 ServerDefinition

Die Klassenbibliothek dient im Wesentlichen dazu, eine Menge von Lieferobjekten zu definieren, zu manipulieren (Bauen, Labeln, Einchecken etc.) und letztendlich auf eine Menge von Zielsystemen zu liefern. Die andere wichtige Abstraktion der Klassenbibliothek betrifft daher die konkrete Definition der Lieferziele über Server-Definitionen. Diese enthalten wenige Informationen, welche über geeignete Methoden abgefragt werden können.

```
IPADDRESS              = 172.29.97.252
TARGETBASEDIR          = /itp0/transfer/R03
SQLCATALOG             = itp01.itpcat
SQLDEFINITIONFILE      = $(VIEWROOT)\DA\...\TDS\TDSdefine_R03_ST
```

Die ersten beiden sprechen sicher für sich, die beiden optionalen SQL-Parameter sind für den SQL-Compiler im Backend nötig. Dessen Definitionen werden nicht vom Entwickler selbst gepflegt, sondern vom Datenbankteam, sodass sie in einem anderen Bereich des Dateibaums abgelegt werden.

Die ServerDefinition-Dateien sind zudem in einem Dateibaum abgelegt, der ab einem wohldefinierten Startverzeichnis (`$(VIEWROOT)`
`UMG`
`dev`
`migration`
`serverdefinitionfiles`) eine dreistufige Struktur der Form `ServerType`
`MigrationLevel`
`Hostname.sdf` aufweist, die implizit weitere Attribute für eine Server-Definition vorgibt:

- Typ (Backend, WebServer, AppProxy, ...),
- Teststufe/Produktion, und
- Rechnername (über den Dateinamen ohne Endung `.sdf`).

Die Dateien werden vom Umgebungsteam gepflegt, da sie auf die jeweiligen Zielrechner und deren Attribute verweisen, die vom Umgebungsteam eingerichtet und gewartet werden.

4.8 Filter: Von der änderungsbasierten Liefermenge zur rechnerorientierten Liefermenge

Primäres Ziel der Klassenbibliothek war, ein Transferwerkzeug zu implementieren, das konfigurierbare Dateimengen auf konfigurierbare Servermengen transferiert. Die Filter sind daher mit die interessantesten Klassen der Bibliothek, da sie genau diese Aufgabe unterstützen. Mit Hilfe der Klassen `DeliveryObjectSet` und `ServerDefinitionSet` lassen sich Mengen von Lieferobjekten und Mengen von Zielsystemen jeweils nach bestimmten Kriterien aufbauen, z.B. als Menge aller geänderten Lieferobjekte aus einer Menge von Setfiles, oder als Menge aller Frontend-Server im technischen Systemtest — die Filterklassen sorgen für die spezifische Sortierung der Mengen nach neuen Kriterien.

Um die Anforderung nach einer Optimierung des Dateitransfers zu erfüllen, müssen nun die Lieferobjekte nach den zu beliefernden Servern sortiert werden. Dazu gibt es eine Klasse `DeliveryObject\-SetForServerDefinitionSet`, deren Konstruktor zwei Objekte als Parameter erhält, die genau die beiden Basismengen als Klassen `DeliveryObjectSet` und `ServerDefinitionSet` repräsentieren. Der Filter sucht nun aus der Lieferobjektmenge alle Lieferobjekte heraus, die für die Server aus der Menge der Zielsysteme geliefert werden sollen. Diese Objekte werden als Mengen in einem Objekt der Klasse `DeliveryObjectSetForServer` zusammengefasst. Anschließend kann jede dieser Mengen zum jeweiligen Server transferiert werden, wobei nur eine einzige Verbindung aufgebaut werden muss.

Für eine RELP-Lieferung gibt es für das Release-Paket ein sogenanntes Ini-File, das von der Syntax her den Windows Ini-Dateien entspricht. Bestimmte Sektionen der Datei enthalten Listen von Setfiles (volle Pfadnamen), deren Lieferobjekte das Release-Paket ausmachen. Wird das Release-Paket geliefert, müssen alle diese Setfiles eingelesen werden, und die Gesamtheit der Lieferobjekte bestimmt werden.

Falls nur Teillieferungen im Rahmen eines SIRs erfolgen, war in einem der vorhergehenden Schritte (vgl. Abschnitt 2.3.5) bereits die Menge der Lieferobjekte mit den definierenden Setfiles bestimmt und in der ClearQuest-Datenbank abgelegt worden. Diese Information kann man sich beim Transfer zu nutze machen, um gezielt nur diese Setfiles zu bearbeiten. Sie müssen ohnehin eingelesen werden und die Gesamt-Lieferobjektmenge des Setfiles muss bestimmt werden. Da die Dateinamen der tatsächlich zu liefernden Objekte bekannt sind, wird mit Hilfe der Klasse `DeliveryObjectSubset`, die selbst eine Unterklasse von `DeliveryObjectSet` ist, und der daraus abgeleiteten Klasse `DeliveryObject\-SubsetForSetfile` die zuvor berechnete Dateimenge reduziert. Dann kann die Menge, die durch die Subklassen-Ableitung immer noch ein Objekt des Typs `DeliveryObjectSet`

5 Entwurfsmuster, Exceptions, Extensions, Erfahrungen, Etc.

5.1 Skripte

Nach Einführung der Klassenbibliothek wurden einige Skripte neu erstellt, andere wurden bereits teilweise umgestellt (vgl. Abschn. 5.5.1).

5.1.1 TransferTool

Nach Fertigstellung der Klassenbibliothek wurde das Transfer-Werkzeug gebaut. Ziel war es dabei, ein schlankes Kommandozeilenwerkzeug zu bauen, das nach Plausibilitätsprüfung der Parameter "ein paar Objekte initialisiert" und durch mächtige Methodenaufrufe auf diesen Objekten dann seine Aufgabe erfüllt. Im Kern ist es auch dabei geblieben: Jede erlaubte Kombination von Parametern wird auf Initialisierung und Methodenaufruf entsprechender Objekte abgebildet. Die Mächtigkeit des Werzeugs hat jedoch bereits zu einer aufwändigen Plausibilitätsprüfung der möglichen Aufruf-Parameter geführt. Daneben müssen manche angestoßene Vorgänge dennoch sauber voneinander getrennt werden, so dass es intern mittlerweile eine Reihe von Prozeduren gibt, die die Teilschritte voneinander trennen: Prinzipiell läßt sich das Skript beispielsweise mit mehreren Ini-Files (RELP) als Parameter aufrufen, oder mit mehreren Setfiles, deren Inhalte dann nacheinander transferiert werden. Im letzteren Fall sollte der Vorgang ohne zeitintensiven Aufbau von Gesamtmengen der Lieferobjekte durchgeführt werden, wobei sogar für jedes einzelne Setfile eine separate Log-Datei erzeugt wird. Somit hat das Skript mittlerweile auch schon über neunhundert Zeilen Code, ist aber durch modularen Aufbau dennoch recht übersichtlich.

5.1.2 Prüfen von Konfigurationsdateien

Kurze Zeit nach Umstieg wurde klar, dass die Entwickler und Migrateure bei Änderung von Ini- und Setfiles in der Lage sein müssen, eine schnelle Konsistenzprüfung durchzuführen. In manchen Migrationsschritten werden ein Vielzahl von diesen Dateien über alle Teams des Projekts geöffnet, um daraus die Lieferobjekte zu bestimmen. In diesen Migrationsschritten, z.B. in "ST_Mig_beantragen" sind die Aufgaben, mit denen man die Ini- und Setfiles ändern könnte, schon geschlossen. Falls sich in dieser Phase Fehler herausstellen, musste der Migrateur durch relativ aufwändige Zustandswechsel den SAA zunächst wieder in den Zustand "Prog_implementieren" versetzen, und in diesem dann eine Aufgabe anlegen oder neu eröffnen, um die Änderung vornehmen zu können. Dies war umso lästiger, wenn der Fehler in den Dateien eines anderen Teams lag, da man dann erst jemanden aus diesem Team finden muss, der die Fehler korrigiert. Somit wurde auf Basis der Klassenbibliothek ein Skript erstellt, das eine Konsistenzprüfung über Set- und Ini-Files durchführen kann. Dieses Skript ließ sich Dank der Klassenbibliothek effektiv in wenigen Zeilen Perl-Code und in nur drei Stunden inkl. Dokumentation implementieren.

5.2 Exceptions

Die Klassenbibliothek verwendet konsequent "Exceptions": Mit der Perl-Klasse `Error` bzw. mit Unterklassen davon und dazugehörigen Methodenaufrufen, lassen sich Dank der offenen Syntax von Perl (sub/eval-Blöcke lassen sich als try/catch-Blöcke "verschleiern") in einer Weise implementieren, die sich stark an bekannte Muster anderer OO-Sprachen anlehnen:

```
try {
        Log::del ($log);
} catch KM::Error with {
        my $err = shift;
        mywarn ($err);
} catch Error with {
        my $err = shift;
```

```
            my $warning
              = qq{Problems checking in Logfile \"$logfile\": $err\n};
            warn ($warning);
            mywarn ($err);
    };
```

Als problematisch hat sich hier erwiesen, dass der gesamte try/catch-Block mit einem Semikolon abgeschlossen werden muss, worauf die Dokumentation zu `Error` ausdrücklich hinweist, was jedoch leicht vergessen wird, da es keine Fehlermeldung auslöst. Effektiv handelt es sich um Code-Blöcke, die als Parameter an die Methoden `try` und `catch` übergeben werden — dieses Semikolon geht jedoch gelegentlich unter und führt zu äußerst schwer nachvollziehbarem Verhalten des Codes. Nur bei genauem Nachvollziehen der einzelnen Schritte im Debugger war häufig zu klären, wo das Problem lag. Mit der Zeit war natürlich die Erfahrung für diese neue Fehlerquelle gereift und bei Problemen wurde als Erstes die syntaktische Korrektheit des Exception-Handlings geprüft. Bezüglich der geworfenen Exceptions würde man sich manchmal eine strenge Typisierung und Prüfung durch den Perl-Interpreter wünschen.

Insgesamt hat der Einsatz von Exceptions allerdings nicht nur die Eleganz des Codes erhöht, sondern auch die Kompaktheit gefördert: Der Abbruch beliebig tief geschachtelter Aufrufketten und eine gezielte Problembehandlung wären nur durch äußerst aufwändige Konstrukte möglich gewesen, beispielsweise Abbruchroutinen mit komplexen Parametern (oder entsprechenden Objekten). Da dabei ein Wissen des Aufgerufenen über den Aufrufer hätte vorhanden sein müssen, wäre man über kurz oder lang wieder bei entsprechenden Abstraktionen, sprich zusätzlichen Klassen gelandet, hätte also ein eigenes Exception-Handling “erfunden”.

5.3 Tests

Eine weitere wichtige Vorgabe bei der Entwicklung der Klassenbibliothek war die parallele Erstellung von Testfällen, insbesondere von automatisierten Tests. Bei den zuvor entwickelten Skripten war durch geringe Modularisierung und den Aufbau komplexer Zwischenzustände der Test sehr aufwändig. Die Skripte konnten nur durch “normalen” Aufruf getestet werden, was nicht nur lange dauert, da nicht nur intern erstmal entsprechende Zustände aufgebaut werden müssen, sondern auch externe Prozesse angestoßen (Transaktionen in ClearCase und ClearQuest, make/javac/jar) werden müssen. Im Fehlerfall oder auch im Erfolgsfall müssen die Zustände auch wieder zurückgesetzt werden. Spätestens bei der Analyse komplexer Fehlersituationen mit dem Debugger war der Test hier auch extrem unzuverlässig, da z.B. Timeouts auftreten, die zu nicht-deterministischem Verhalten führen. Vollständige Regressionstests nach Refaktorierungen blieben zudem aus, oder wurden nur in bestimmten langfristigen Zyklen durchgeführt, die eine gezielte Reaktion auf durch einzelne Änderungen entstandene Fehler unmöglich machten.

5.3.1 Unit-Tests

Bei einer Entwicklung dedizierter Klassen bietet es sich an, für jede der Klassen eine Testklasse bzw. ein Testskript (Unit-Test) und zugehörige Testfälle (Eingabedaten und erwartete Ausgabedaten) zu erzeugen. Damit ist ein automatischer Regressionstest sehr einfach und gezielt möglich. Insgesamt wurden 20 einzelne Test-Skripte mit `Test::More`, sowie ein Gesamt-Testskript auf Basis von `Test::Harness` entwickelt, die insgesamt mehr als 140 Testfälle abdecken. Es werden jedoch nur die Basisfunktionen der Klassen getestet, es könnten sicherlich insgesamt deutlich mehr Testfälle definiert und automatisiert abgedeckt werden. Nichtsdestotrotz läßt sich damit in wenigen Minuten ein automatischer Regressionstest aller Klassen durchführen, der zumindest die vom `TransferTool` benötigten Funktionen abdeckt und die hohe Qualität der Klassenbibliothek weitgehend sicherstellt. Auftretende Fehler lassen sich mit Hilfe der Testfälle in der Regel schnell eingrenzen.

5.3.2 Modulgruppen-Tests

Ein automatisierter Modulgruppentest, also ein Test auf korrektes Zusammenspiel aller Klassen läßt sich in diesem Umfeld nur mit großem Aufwand realisieren, da große Mengen von Objekten komplex bestimmt und verarbeitet werden müssen. Zur Verarbeitung zählt dabei der Transfer von Dateien auf andere Rechner. Tests wurden daher hier auf Basis von "Referenz-Übertragungen" der Dateimengen realisiert: Nach jedem Modulgruppentest wurde (manuell gestartet) ein Abgleich der tatsächlich übertragenen Menge an Dateien auf den verschiedenen Rechnern mit der Referenz-Mengen verglichen.

5.4 Optimierungen und Entwurfsmuster

Bereits beim Entwurf wurden verschiedene Entwurfsmuster [GHJV95] verwendet, später bei der Implementierung weitere ergänzt. Die Verwendung von Entwurfsmustern ermöglicht die Wiederverwendung eines bekannten Verfahrens in der Software-Entwicklung und entlastet den Designer/Entwickler vom Entwurf einer eigenen Lösung für Standard-Probleme. Gleichzeitig erhöht sich die Wiederverwendbarkeit der Implementierung für vergleichbare Teilprobleme.

5.4.1 Sichtbares Caching per Singleton/Factory

Ein weit verbreitetes Muster ist die Verwendung von Singletons, vielleicht vergleichbar mit "globalen" Variablen in Non-OO Programmen. Objekte, die nur einmal im Programm vorhanden sein dürfen, werden als Klasse gekapselt. Der Zugriff auf das Objekt oder seine Teilobjekte über Methoden der Klasse, anstelle eines direkten Zugriffs auf seine Attribute, ermöglichen eine Abstraktion und damit Austauschbarkeit der Klasse bzw. des Singletons. Wir verwenden Singletons beispielsweise zur Implementierung von Caches, und damit zur Optimierung.

`DeliveryObjectCache` ist eine derartige Klasse. Sie bietet als einzige Methode eine sogenannte Factory-Methode `retrieve($fullpath)` an, mit der ein Objekt der Klasse `DeliveryObject` "generiert" wird. Tatsächlich wird nur in einem globalen internen Objekt (einem Hash) nachgesehen, ob das Objekt bereits existiert. Wenn ja, wird dieses zurückgegeben, wenn nein wird per Konstruktor ein neues erstellt. Der Aufrufer verwendet also nicht den Konstruktor selbst, sondern bemüht die Factory-Methode der Singleton-Klasse, um das Objekt zu "erzeugen". Anschließend setzt der Aufrufer Attribute des Objektes, z.B. das dazugehörige `Setfile`. Dies ermöglicht zusätzlich die einfache Prüfung, ob `DeliveryObject`s nur von maximal einem `Setfile` definiert werden: Falls das Object schon ein gesetztes `Setfile`-Attribut hat, wird ein `Error` geworfen. Die (globalen!) `DeliveryObject`s werden beim Berechnen von Setfile-Objekten in verschiedenen Berechnungsdurchläufen mit zusätzlichen Informationen über Transfergruppen ergänzt. Die Verwendung des Caches erspart der Setfile-Klasse zudem ein eigenes Cache/Retrieve, was unter anderem die Setfile-Objekte im Speicher sehr unübersichtlich gemacht hätte (Debugging!).

5.4.2 Hintergründiges Caching per Konstruktor

Auch die Klasse `Setfile` verwendet ein Singleton mit einem globalen Cache für Objekte der Klasse, allerdings nicht nach außen über eine Methode exportiert. Vielmehr werden Objekte der Klasse beim Aufruf des Konstruktors (`newSetfile(...)` zunächst im "globalen" Cache der Klasse gesucht und ggf. wiederverwendet. Dies bringt einen enormen Performance-Gewinn, da `Setfile`-Objekte zum Teil äußerst aufwändig aus den `Setfile`-Dateien zu berechnen sind.

5.4.3 Vereinheitlichung von Datei-basierten Initialisierungen

Die Klasse `FileReader` implementiert in ihrem Konstruktor einen einheitlichen Algorithmus zum Einlesen der zugrundeliegenden Datei. Der Konstruktor ist somit eine "Template Method". Jede abgeleitete Klasse (immerhin vier direkte Abkömmlinge und ein indirekter!) definiert entweder keinen

eigenen Konstruktor, oder ruft in ihrem eigenen Konstruktor den der Oberklasse auf, um diesen Algorithmus wiederzuverwenden. Hierbei kommt eine der Stärken von Perl zum Tragen, nämlich dass Parameterlisten nicht typisiert sind. Jede Unterklasse kann zu ihrer Konstruktion weitere Attribute erwarten, die als Array, Hash oder Hash-Referenz an den Konstruktor übergeben werden. Diese werden vom `FileReader`-Konstruktor generisch als Attribute der (abgeleiteten) Klasse gesetzt. Anschließend wird die Methode `verify` aufgerufen, um ggf. die Attribute zu prüfen. Daneben ruft der Konstruktor `load` und `verifyAfterLoad` auf, um die Datei zu laden bzw. um den geladenen Inhalt zu prüfen. Mehrere abgeleitete Klassen definieren die `verify`-Methoden nicht neu. In diesem Fall wird die leere Implementierung der Basisklasse `FileReader` verwendet. Einzig `load` muss von jedem Erben implementiert werden. Die Methode der Basisklasse wirft bei Aufruf eine Exception.

5.4.4 Labeln per "Transaktion"

Im Laufe der Implementierung und ersten Tests hat sich das Labeln von Dateien als Flaschenhals herausgestellt: Wird einer Datei (Klasse `File` und Erben) ein Label gesetzt (Methode `label`), muss auch allen Verzeichnis-Objekten im Pfad der Datei bis zur Wurzel des Dateibaums das gleiche Label gesetzt werden, da die Datei sonst in Label-basierten ClearCase-Views nicht sichtbar ist. Die Methode musste dabei für jedes Verzeichnis zunächst das entsprechende ClearCase-Objekt initialisieren — eine relativ zeitaufwändige Methode — um anschließend in sämtlichen ClearCase-Objekten das Label zu setzen. Da beim Labeln viele Objekte gemeinsame Pfad-Komponenten haben, wurde vielen der Verzeichnis-Objekte immer wieder das gleiche Label gesetzt und sie mußten vor allem immer wieder als ClearCase-Objekt initialisiert werden.

Hier bot sich ebenfalls die Verwendung eines Singletons als Cache an. `File::label` setzt also das Label nicht direkt, sondern trägt die Datei bzw. ihren Pfad in einen Cache (Singleton-Klasse `Label`) ein, sammelt also bei den Aufrufen nur alle Dateinamen, die gelabelt werden müssen. Tatsächlich werden nicht nur die Pfade gesammelt, sondern auch gleich in Pfadkomponenten zerlegt, die in einen Hash eingetragen werden. Somit konnten die Verfahren, die ein Label setzen, weiterhin die Methode `label` auf den von ihnen genutzten `File`-Objekten verwenden. Nun gibt es natürlich die Schwierigkeit, dass zu irgendeinem Zeitpunkt tatsächlich gelabelt werden muss. Daher implementiert die Klasse `Label` ein Transaktionsmuster: Das Sammeln der Dateien wird durch die Methode `startTransaction` (ist implizit beim Laden der Klasse angenommen) und `commitTransaction` eingerahmt. Letztere geht den Cache durch, initialisiert pro Eintrag einmal das dazugehörige ClearCase-Objekt und labelt dieses genau einmal. Selbstverständlich gibt es auch eine Methode `abortTransaction`, die das Sammeln der Label-Objekte abbricht, was beispielsweise in Exception-Handlern genutzt wird.

Gegenüber der bisherigen Label-Implementierung (in den alten Skripts, nicht in der ersten Version der Label-Klasse) konnte somit die Laufzeit um über 60% verkürzt werden. Bei bestimmten Übergängen der Testphasen, z.B. von der Systemtestphase zur IAT-Phase, bei der die Migration aller systemgetesten Lieferobjekte auf die IAT-Systeme durch Setzen eines Labels vorbereitet wird, ließen sich so einige Stunden einsparen. Die Definition eines klaren Transaktions-Zustandes ermöglicht zudem seine Überwachung in den `END`-Blöcken der Perl-Klasse: Ist die Transaktion am Ende des Programms nicht korrekt abgeschlossen oder definiert abgebrochen worden, gibt es zumindest ein Logging dieses Problems und bei Bedarf auch eine weitergehende Fehlerbehandlung, was zur Sicherheit der Skripte beiträgt.

5.5 Anpassungen und Erweiterungen

5.5.1 Anpassung bestehender Skripte

Neben der Neuimplementierung u.a. des Transfer-Skripts sollte die Klassenbibliothek auch eine Refaktorierung des bestehenden Codes ermöglichen. Ein wichtiger Punkt war hier die Vereinheitlichung von Setfiles und die Wiederverwendung gemeinsamer Algorithmen, wie z.B. das Labeln. Diese beiden Punkte wurden mit als Erstes nach Fertigstellung in Angriff genommen: Der bestehende Code wurde allein hier um einiges schlanker und es gibt nun vor allem eine einheitliche Semantik von Setfiles. Auch das

vereinheitlichte Labeln auf Basis des Transaktions-Musters (s.o.) hat eine klare Laufzeitverbesserung der bestehenden Skripte gebracht.

5.5.2 Liefern von Java WAR-Files

Eine wichtige Erweiterungsanforderung nach Fertigstellung des Designs und von Teilen der Implementierung war der Wunsch, Java WAR-Files (Web-Application aRchive) oder auch EAR-Files (Enterprise Application aRchive) liefern zu können. Damit kann einem Java Applikationsserver eine komplette Web-Applikation oder Enterprise-Applikation "in einem Stück" übergeben werden. Ein WAR/EAR-File ist ein Jar-File, das Bibliotheken (Jar-Files) und andere verwendete Objekte (HTML-Dateien, Images, JSPs, Enterprise Beans, ...) sowie einige weitere Informationen (Manifest) enthält.

Aus Sicht der Klassenbibliothek stellt eine solche Erweiterung kein grosses Problem dar: Statt Lieferobjekte direkt per FTP zuzustellen, werden sie zu einem WAR-File zusammengefasst und dieses geliefert. Diese Änderungen konnten recht einfach als Refaktorierungen und Erweiterungen bestehenden Codes implementiert werden. Zusätzlich wurde eine neue Klasse `XDfile` (WAR/EAR-Definition file) eingeführt, die Beschreibungsdateien für WAR/EAR-Files verarbeiten kann. Diese beerbt zum einen `IniFileReader`, weil der Aufbau der Beschreibungsdatei einem Ini-File entspricht. Zum anderen ist sie von `Setfile` abgeleitet, da das sonstige Verhalten einem Setfile entspricht: Es muss ein (einziges) Lieferobjekt bereitgestellt werden, dieses muß transferiert werden, es besteht aus einer Menge von Objekten (andere Lieferobjekte) etc. Durch die Mehrfachvererbung ist eine übersichtliche Implementierung (366 Zeilen gegenüber 1727 Zeilen in `Setfile.pm` und 154 Zeilen in `IniFileReader.pm`) unter Wiederverwendung zahlreicher Methoden (10 neue/überschriebene vs. 64 + 4 Methoden in den Basisklassen) gelungen.

5.5.3 Verwendung weiterer Entwurfsmuster

Schwieriger war die Erweiterung des Steuerungsskriptes `transfertool.pl`. Dieses erlaubte ohnehin schon zahlreiche Parameter für die verschiedenen Übergabearten von `DeliveryObjectSet`-Definitionen (Setfiles, Inifiles, ...) und `ServerDefinitionSet`-Definitionen, die prinzipiell beliebig miteinander kombiniert werden können (vgl. Abschn. 5.1.1). Nun kamen weitere hinzu, um auch XDfiles als Parameter zuzulassen. Faktisch implementiert ein großer Teil des Skriptes ein "Command"-Entwurfsmuster. Konsequenterweise sollte dieses auf Dauer durch eine entsprechende Klasse gekapselt sein, und durch "Template"-Methoden refaktoriert werden, da bei der Auswertung der einzelnen Eingabedateien auch immer wieder die gleichen Verarbeitungsmuster zum Tragen kommen.

5.6 Unabhängige Module

Neben den direkt miteinander verzahnten Klassen entstanden einige "Module" (oder Singleton-Klassen), die nicht direkt zur Implementierung der geforderten Funktionen nötig waren. Trotzdem vereinfachen sie die Implementierung erheblich und sollen hier nicht unerwähnt bleiben.

5.6.1 Logging + Tracing

Bei der Nutzung der Skripts, aber auch beim Test und Debugging oder dem Nachstellen von Fehlern im Produktionsbetrieb stellen die Kontrollausgaben auf dem Bildschirm und das Schreiben von Log-Dateien einen wichtigen und schwierigen Bereich dar. Der Anwender will zum einen möglichst wenig an Ausgaben sehen: Im Wesentlichen, ob die Anwendung funktioniert hat, oder im Falle von Problemen die mögliche Ursache. Zur Qualitätssicherung soll aber ein Protokoll vorliegen, das alle "wesentlichen" Aktionen mitprotokolliert. Bei Bedarf sollen sowohl die Bildschirm-Ausgaben, wie auch die Tiefe der Log-Informationen beim Start der Skripts in mehreren Stufen verfeinert werden. Zur Revisionssicherheit muss zudem das "Standard"-Logfile in ClearCase eingecheckt werden. Es wurde daher eine bestehende Log-Klassenbibliothek des Autors, die zentrales Logging auf verschiedenen Ausgabemedien erlaubt,

erweitert. Jede Ausgabe im Programm wird dabei als Aufruf einer statischen Methode der Logklasse mit jeweils einem Log-Level (Debug, Info, Notice, Warning, Error, Critical, ...) verbunden. Dieser zentralen Logklasse können verschiedene Observer (Entwurfsmuster!) als Datensenken dynamisch hinzugefügt werden, z.B. die Konsole, Logdateien für den Gesamtablauf oder bestimmte Schritte wie den Transfer auf die einzelnen Server, Systemmanagement-Tools wie syslog, Mail-Empfänger etc. Jedem Observer kann man ein Loglevel mitgeben, auf bzw. ab dem dieses die Logmeldungen mitprotokolliert.

Windows Command-Fenster ermöglichen beispielsweise mehrfarbige Ausgaben für die verschiedenen Loglevel, was die Übersichtlichkeit für den Anwender erhöht. In der Klassenbibliothek selbst wurde eine Klasse `Log` erstellt, die von der allgemeinen (Observer-) Log-Klasse erbt, und die Logdateien beim Schliessen automatisch in ClearCase eincheckt.

5.6.2 ClearCase

ClearCase bringt eigene Perl-Klassen [RAT] mit, die unter Windows im wesentlichen der Kapselung von COM/OLE-Objekten dienen. Diese bieten dann zahlreiche Methoden, beispielsweise um Operationen auf den verschiedenen Versionen der einzelnen Dateien durchzuführen. Wir haben diese um zwei Klassen ergänzen, die projektspezifische Schnittstellen hinzufügen, beispielsweise zur Berechnung der Release-Nummer des Entwicklungsprojekts aus ClearCase-View- und -Projekt-Bezeichnern. Es gibt auch Methoden, die mächtige Funktionen auf die Basis-Funktionen von ClearCase zurückführen, beispielsweise zum checkin von Dateien, bei dem in ClearCase verschiedene Schritte durchgeführt werden müssen, je nach dem ob die Datei/Version schon existierte (neue Version erstellen und einchecken) oder noch nicht (Verzeichnis auschecken, neues Element hinzufügen, neues Element einchecken, Verzeichnis einchecken). Diese Methoden wurden außerdem um Projekt-spezifisches Exception-Handling ergänzt.

6 Zusammenfassung und Ausblick

Die implementierte Klassenbibliothek hat die Realisierung eines eigenständigen Transfer-Werkzeugs ermöglicht, das komplexe Softwarelieferungen in optimierter Weise erlaubt. Sie ist damit ein wichtiger Bestandteil eines kontrollierten, teilweise automatisierten und sicheren Lieferprozesses, der einen wesentlichen Beitrag zur Qualitätssicherung der entwickelten Software bei deren Test und Deployment leistet. Die bestehenden Werkzeuge konnten deutlich verschlankt und vereinheitlicht werden. Weiterhin konnten neue Anforderungen mit wenig Aufwand und in kurzer Zeit umgesetzt werden (vgl. Abschn. 5.1.2 und 5.5.2);

6.1 Einsatz der Werkzeuge

Insgesamt kann festgehalten werden, dass der Transferprozess, aber auch anderer Teilprozesse der Softwarelieferung, im Rahmen der SAA-basierten Migration ohne Neuentwicklung der Klassenbibliothek kaum beherrschbar gewesen wäre. Hinsichtlich des Durchsatzes gab es zunächst einen Einbruch, wobei ein Vergleich durch die unterschiedliche Komplexität sehr schwierig ist:

- Bei der vorherigen Implementierung waren die Inhalte einzelner Setfiles in (zahlreichen) manuell gesteuerten Einzelschritten transferiert worden,
- die SAA-basierte Migration erlaubt hingegen unter Kontrolle des Change-Management-Prozesses das automatische Starten von einzelnen Transfers , was die Qualität der Ergebnisse deutlich erhöht. Die dazu notwendigen Berechnungen und Analysen der Abhängigkeiten ist ungleich aufwändiger.

Durch die Implementierung als übersichtliche Einzel-Klassen war jedoch eine gezielte Analyse der Durchsatzprobleme möglich, die letztendlich zu klaren Optimierungen (Caching, transaktionsbasiertes Labeln) geführt hat, sodass jetzt die Durchlaufzeiten wieder im Rahmen des für Migrateure Erträglichen liegen.

6.2 Weitere Refaktorierung bestehender Skripte

Im Rahmen des Projektes wurde eine neue Klassenbibliothek im Umfang von über 7500 Zeilen Perl-Code erstellt. Dazu kommen 915 Zeilen für `transfertool.pl` und 350 Zeilen für `chkIniSetfi\-les.pl`. Die "Altlasten" haben derzeit (nach Refaktorierung und Entfernung redundanten Codes) einen Umfang von über 13000 Zeilen. Dieser Umfang und damit die Komplexität kann durch weitere Refaktorierung noch stark reduziert werden, da beispielsweise das `build` der Lieferobjekte vereinheitlicht und in die Klassenbibliothek integriert werden kann. Idealerweise sollte am Ende des Refaktorierungsprozesses der gesamte Funktionsumfang objektorientiert in der Klassenbibliothek implementiert sein. Die Skripte und ggf. angeschlossene Module sollten nur noch der Analyse und der Konsistenzprüfung der Aufrufparameter dienen. Anschließend sollten sie Objekte aus der Klassenbibliothek initialisieren und deren Methoden aufrufen.

Literatur

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[RAT] RATIONAL SOFTWARE CORPORATION (Hrsg.). *ClearCase Automation Library*

[RAT99a] *Administering Rational ClearQuest*. 4.0. Rational Software Corporation, 1999

[RAT99b] *Building Software with ClearCase*. 4.0. Rational Software Corporation, 1999

Autoreninfo

Gerd Aschemann engagiert sich als freiberuflicher Consultant in den Schwerpunktbereichen Systemarchitekturen, Objektorientierte Softwareentwicklung und Konfigurationsmanagement. Daneben verfügt er über langjährige Erfahrungen aus der Administration verteilter Systeme und Netze.